Short Contents

Table of Contents

1 perl

1.1 NAME

perl - The Perl 5 language interpreter

1.2 SYNOPSIS

perl [**-sTtuUWX**] [**-hv**] [**-V**[:*configvar*]] [**-cw**] [**-d**[**t**][:*debugger*]] [**-D**[*number/list*]]
[**-pna**] [**-F***pattern*] [**-l**[*octal*]] [**-0**[*octal/hexadecimal*]] [**-I***dir*] [**-m**[**-**]*module*] [**-M**[**-**]*'module...'*] [**-f**]
[**-C** [*number/list*]] [**-S**] [**-x**[*dir*]] [**-i**[*extension*]] [[**-e**|**-E**] *'command'*] [−] [*programfile*] [*argument*]...

For more information on these options, you can run `perldoc perlrun`.

1.3 GETTING HELP

The `perldoc` program gives you access to all the documentation that comes with Perl. You can get more documentation, tutorials and community support online at `http://www.perl.org/`.

If you're new to Perl, you should start by running `perldoc perlintro`, which is a general intro for beginners and provides some background to help you navigate the rest of Perl's extensive documentation. Run `perldoc perldoc` to learn more things you can do with `perldoc`.

For ease of access, the Perl manual has been split up into several sections.

1.3.1 Overview

```
    perl                Perl overview (this section)
    perlintro           Perl introduction for beginners
    perlrun             Perl execution and options
    perltoc             Perl documentation table of contents
```

1.3.2 Tutorials

```
    perlreftut          Perl references short introduction
    perldsc             Perl data structures intro
    perllol             Perl data structures: arrays of arrays

    perlrequick         Perl regular expressions quick start
    perlretut           Perl regular expressions tutorial

    perlootut           Perl OO tutorial for beginners

    perlperf            Perl Performance and Optimization Techniques

    perlstyle           Perl style guide

    perlcheat           Perl cheat sheet
    perltrap            Perl traps for the unwary
```

perldebtut	Perl debugging tutorial

perlfaq	Perl frequently asked questions
perlfaq1	General Questions About Perl
perlfaq2	Obtaining and Learning about Perl
perlfaq3	Programming Tools
perlfaq4	Data Manipulation
perlfaq5	Files and Formats
perlfaq6	Regexes
perlfaq7	Perl Language Issues
perlfaq8	System Interaction
perlfaq9	Networking

1.3.3 Reference Manual

perlsyn	Perl syntax
perldata	Perl data structures
perlop	Perl operators and precedence
perlsub	Perl subroutines
perlfunc	Perl built-in functions
perlopentut	Perl open() tutorial
perlpacktut	Perl pack() and unpack() tutorial
perlpod	Perl plain old documentation
perlpodspec	Perl plain old documentation format specification
perlpodstyle	Perl POD style guide
perldiag	Perl diagnostic messages
perllexwarn	Perl warnings and their control
perldebug	Perl debugging
perlvar	Perl predefined variables
perlre	Perl regular expressions, the rest of the story
perlrebackslash	Perl regular expression backslash sequences
perlrecharclass	Perl regular expression character classes
perlreref	Perl regular expressions quick reference
perlref	Perl references, the rest of the story
perlform	Perl formats
perlobj	Perl objects
perltie	Perl objects hidden behind simple variables
perldbmfilter	Perl DBM filters

perlipc	Perl interprocess communication
perlfork	Perl fork() information
perlnumber	Perl number semantics

perlthrtut	Perl threads tutorial

perlport	Perl portability guide
perllocale	Perl locale support

perluniintro	Perl Unicode introduction
perlunicode	Perl Unicode support
perlunicook	Perl Unicode cookbook
perlunifaq	Perl Unicode FAQ
perluniprops	Index of Unicode properties in Perl
perlunitut	Perl Unicode tutorial
perlebcdic	Considerations for running Perl on EBCDIC platforms
perlsec	Perl security
perlmod	Perl modules: how they work
perlmodlib	Perl modules: how to write and use
perlmodstyle	Perl modules: how to write modules with style
perlmodinstall	Perl modules: how to install from CPAN
perlnewmod	Perl modules: preparing a new module for distribution
perlpragma	Perl modules: writing a user pragma
perlutil	utilities packaged with the Perl distribution
perlfilter	Perl source filters
perldtrace	Perl's support for DTrace
perlglossary	Perl Glossary

1.3.4 Internals and C Language Interface

perlembed	Perl ways to embed perl in your C or C++ application
perldebguts	Perl debugging guts and tips
perlxstut	Perl XS tutorial
perlxs	Perl XS application programming interface
perlxstypemap	Perl XS C/Perl type conversion tools
perlclib	Internal replacements for standard C library functions
perlguts	Perl internal functions for those doing extensions
perlcall	Perl calling conventions from C
perlmroapi	Perl method resolution plugin interface
perlreapi	Perl regular expression plugin interface
perlreguts	Perl regular expression engine internals
perlapi	Perl API listing (autogenerated)
perlintern	Perl internal functions (autogenerated)
perliol	C API for Perl's implementation of IO in Layers
perlapio	Perl internal IO abstraction interface
perlhack	Perl hackers guide
perlsource	Guide to the Perl source tree
perlinterp	Overview of the Perl interpreter source and how it works

perlhacktut	Walk through the creation of a simple C code patch
perlhacktips	Tips for Perl core C code hacking
perlpolicy	Perl development policies
perlgit	Using git with the Perl repository

1.3.5 Miscellaneous

perlbook	Perl book information
perlcommunity	Perl community information
perldoc	Look up Perl documentation in Pod format
perlhist	Perl history records
perldelta	Perl changes since previous version
perl52111delta	Perl changes in version 5.21.11
perl52110delta	Perl changes in version 5.21.10
perl5219delta	Perl changes in version 5.21.9
perl5218delta	Perl changes in version 5.21.8
perl5217delta	Perl changes in version 5.21.7
perl5216delta	Perl changes in version 5.21.6
perl5215delta	Perl changes in version 5.21.5
perl5214delta	Perl changes in version 5.21.4
perl5213delta	Perl changes in version 5.21.3
perl5212delta	Perl changes in version 5.21.2
perl5211delta	Perl changes in version 5.21.1
perl5210delta	Perl changes in version 5.21.0
perl5202delta	Perl changes in version 5.20.2
perl5201delta	Perl changes in version 5.20.1
perl5200delta	Perl changes in version 5.20.0
perl5184delta	Perl changes in version 5.18.4
perl5182delta	Perl changes in version 5.18.2
perl5181delta	Perl changes in version 5.18.1
perl5180delta	Perl changes in version 5.18.0
perl5163delta	Perl changes in version 5.16.3
perl5162delta	Perl changes in version 5.16.2
perl5161delta	Perl changes in version 5.16.1
perl5160delta	Perl changes in version 5.16.0
perl5144delta	Perl changes in version 5.14.4
perl5143delta	Perl changes in version 5.14.3
perl5142delta	Perl changes in version 5.14.2
perl5141delta	Perl changes in version 5.14.1
perl5140delta	Perl changes in version 5.14.0
perl5125delta	Perl changes in version 5.12.5
perl5124delta	Perl changes in version 5.12.4
perl5123delta	Perl changes in version 5.12.3
perl5122delta	Perl changes in version 5.12.2
perl5121delta	Perl changes in version 5.12.1

perl5120delta	Perl changes in version 5.12.0
perl5101delta	Perl changes in version 5.10.1
perl5100delta	Perl changes in version 5.10.0
perl589delta	Perl changes in version 5.8.9
perl588delta	Perl changes in version 5.8.8
perl587delta	Perl changes in version 5.8.7
perl586delta	Perl changes in version 5.8.6
perl585delta	Perl changes in version 5.8.5
perl584delta	Perl changes in version 5.8.4
perl583delta	Perl changes in version 5.8.3
perl582delta	Perl changes in version 5.8.2
perl581delta	Perl changes in version 5.8.1
perl58delta	Perl changes in version 5.8.0
perl561delta	Perl changes in version 5.6.1
perl56delta	Perl changes in version 5.6
perl5005delta	Perl changes in version 5.005
perl5004delta	Perl changes in version 5.004

perlexperiment	A listing of experimental features in Perl

perlartistic	Perl Artistic License
perlgpl	GNU General Public License

1.3.6 Language-Specific

perlcn	Perl for Simplified Chinese (in EUC-CN)
perljp	Perl for Japanese (in EUC-JP)
perlko	Perl for Korean (in EUC-KR)
perltw	Perl for Traditional Chinese (in Big5)

1.3.7 Platform-Specific

perlaix	Perl notes for AIX
perlamiga	Perl notes for AmigaOS
perlandroid	Perl notes for Android
perlbs2000	Perl notes for POSIX-BC BS2000
perlce	Perl notes for WinCE
perlcygwin	Perl notes for Cygwin
perldos	Perl notes for DOS
perlfreebsd	Perl notes for FreeBSD
perlhaiku	Perl notes for Haiku
perlhpux	Perl notes for HP-UX
perlhurd	Perl notes for Hurd
perlirix	Perl notes for Irix
perllinux	Perl notes for Linux
perlmacos	Perl notes for Mac OS (Classic)
perlmacosx	Perl notes for Mac OS X
perlnetware	Perl notes for NetWare

```
perlopenbsd         Perl notes for OpenBSD
perlos2             Perl notes for OS/2
perlos390           Perl notes for OS/390
perlos400           Perl notes for OS/400
perlplan9           Perl notes for Plan 9
perlqnx             Perl notes for QNX
perlriscos          Perl notes for RISC OS
perlsolaris         Perl notes for Solaris
perlsymbian         Perl notes for Symbian
perlsynology        Perl notes for Synology
perltru64           Perl notes for Tru64
perlvms             Perl notes for VMS
perlvos             Perl notes for Stratus VOS
perlwin32           Perl notes for Windows
```

1.3.8 Stubs for Deleted Documents

```
perlboot
perlbot
perlrepository
perltodo
perltooc
perltoot
```

On a Unix-like system, these documentation files will usually also be available as manpages for use with the `man` program.

Some documentation is not available as man pages, so if a cross-reference is not found by man, try it with `perldoc`. Perldoc can also take you directly to documentation for functions (with the **-f** switch). See `perldoc --help` (or `perldoc perldoc` or `man perldoc`) for other helpful options `perldoc` has to offer.

In general, if something strange has gone wrong with your program and you're not sure where you should look for help, try making your code comply with **use strict** and **use warnings**. These will often point out exactly where the trouble is.

1.4 DESCRIPTION

Perl officially stands for Practical Extraction and Report Language, except when it doesn't.

Perl was originally a language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It quickly became a good language for many system management tasks. Over the years, Perl has grown into a general-purpose programming language. It's widely used for everything from quick `"one-liners"` to full-scale application development.

The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). It combines (in the author's opinion, anyway) some of the best features of **sed**, **awk**, and **sh**, making it familiar and easy to use for Unix users to whip up quick solutions to annoying problems. Its general-purpose programming facilities support procedural, functional, and object-oriented programming paradigms, making Perl a comfortable language for the long haul on major projects, whatever your bent.

Perl's roots in text processing haven't been forgotten over the years. It still boasts some of the most powerful regular expressions to be found anywhere, and its support for Unicode text is world-class. It handles all kinds of structured text, too, through an extensive collection of extensions. Those libraries, collected in the CPAN, provide ready-made solutions to an astounding array of problems. When they haven't set the standard themselves, they steal from the best – just like Perl itself.

1.5 AVAILABILITY

Perl is available for most operating systems, including virtually all Unix-like platforms. See Section 56.7 [perlport Supported Platforms], page 980 for a listing.

1.6 ENVIRONMENT

See Section 69.1 [perlrun NAME], page 1176.

1.7 AUTHOR

Larry Wall <larry@wall.org>, with the help of oodles of other folks.

If your Perl success stories and testimonials may be of help to others who wish to advocate the use of Perl in their applications, or if you wish to simply express your gratitude to Larry and the Perl developers, please write to perl-thanks@perl.org .

1.8 FILES

```
 "@INC"                 locations of perl libraries
```

1.9 SEE ALSO

```
 http://www.perl.org/       the Perl homepage
 http://www.perl.com/       Perl articles (O'Reilly)
 http://www.cpan.org/       the Comprehensive Perl Archive
 http://www.pm.org/         the Perl Mongers
```

1.10 DIAGNOSTICS

Using the `use strict` pragma ensures that all variables are properly declared and prevents other misuses of legacy Perl features.

The `use warnings` pragma produces some lovely diagnostics. One can also use the **`-w`** flag, but its use is normally discouraged, because it gets applied to all executed Perl code, including that not under your control.

See Section 16.1 [perldiag NAME], page 137 for explanations of all Perl's diagnostics. The `use diagnostics` pragma automatically turns Perl's normally terse warnings and errors into these longer forms.

Compilation errors will tell you the line number of the error, with an indication of the next token or token type that was to be examined. (In a script passed to Perl via **`-e`** switches, each **`-e`** is counted as one line.)

Setuid scripts have additional constraints that can produce error messages such as "Insecure dependency". See Section 70.1 [perlsec NAME], page 1198.

Did we mention that you should definitely consider using the **use warnings** pragma?

1.11 BUGS

The behavior implied by the **use warnings** pragma is not mandatory.

Perl is at the mercy of your machine's definitions of various operations such as type casting, atof(), and floating-point output with sprintf().

If your stdio requires a seek or eof between reads and writes on a particular stream, so does Perl. (This doesn't apply to sysread() and syswrite().)

While none of the built-in data types have any arbitrary size limits (apart from memory size), there are still a few arbitrary limits: a given variable name may not be longer than 251 characters. Line numbers displayed by diagnostics are internally stored as short integers, so they are limited to a maximum of 65535 (higher numbers usually being affected by wraparound).

You may mail your bug reports (be sure to include full configuration information as output by the myconfig program in the perl source tree, or by `perl -V`) to perlbug@perl.org . If you've succeeded in compiling perl, the `perlbug` script in the `utils/` subdirectory can be used to help mail in a bug report.

Perl actually stands for Pathologically Eclectic Rubbish Lister, but don't tell anyone I said that.

1.12 NOTES

The Perl motto is "There's more than one way to do it." Divining how many more is left as an exercise to the reader.

The three principal virtues of a programmer are Laziness, Impatience, and Hubris. See the Camel Book for why.

2 perlapio

2.1 NAME

perlapio - perl's IO abstraction interface.

2.2 SYNOPSIS

```
#define PERLIO_NOT_STDIO 0    /* For co-existence with stdio only */
#include <perlio.h>           /* Usually via #include <perl.h> */

PerlIO *PerlIO_stdin(void);
PerlIO *PerlIO_stdout(void);
PerlIO *PerlIO_stderr(void);

PerlIO *PerlIO_open(const char *path,const char *mode);
PerlIO *PerlIO_fdopen(int fd, const char *mode);
PerlIO *PerlIO_reopen(const char *path, const char *mode, PerlIO *old);  /* deprecated
int     PerlIO_close(PerlIO *f);

int     PerlIO_stdoutf(const char *fmt,...)
int     PerlIO_puts(PerlIO *f,const char *string);
int     PerlIO_putc(PerlIO *f,int ch);
SSize_t PerlIO_write(PerlIO *f,const void *buf,size_t numbytes);
int     PerlIO_printf(PerlIO *f, const char *fmt,...);
int     PerlIO_vprintf(PerlIO *f, const char *fmt, va_list args);
int     PerlIO_flush(PerlIO *f);

int     PerlIO_eof(PerlIO *f);
int     PerlIO_error(PerlIO *f);
void    PerlIO_clearerr(PerlIO *f);

int     PerlIO_getc(PerlIO *d);
int     PerlIO_ungetc(PerlIO *f,int ch);
SSize_t PerlIO_read(PerlIO *f, void *buf, size_t numbytes);

int     PerlIO_fileno(PerlIO *f);

void    PerlIO_setlinebuf(PerlIO *f);

Off_t   PerlIO_tell(PerlIO *f);
int     PerlIO_seek(PerlIO *f, Off_t offset, int whence);
void    PerlIO_rewind(PerlIO *f);

int     PerlIO_getpos(PerlIO *f, SV *save);        /* prototype changed */
int     PerlIO_setpos(PerlIO *f, SV *saved);       /* prototype changed */
int     PerlIO_fast_gets(PerlIO *f);
int     PerlIO_has_cntptr(PerlIO *f);
SSize_t PerlIO_get_cnt(PerlIO *f);
char   *PerlIO_get_ptr(PerlIO *f);
void    PerlIO_set_ptrcnt(PerlIO *f, char *ptr, SSize_t count);

int     PerlIO_canset_cnt(PerlIO *f);              /* deprecated */
void    PerlIO_set_cnt(PerlIO *f, int count);      /* deprecated */

int     PerlIO_has_base(PerlIO *f);
char   *PerlIO_get_base(PerlIO *f);
SSize_t PerlIO_get_bufsiz(PerlIO *f);

PerlIO *PerlIO_importFILE(FILE *stdio, const char *mode);
FILE   *PerlIO_exportFILE(PerlIO *f, int flags);
FILE   *PerlIO_findFILE(PerlIO *f);
void    PerlIO_releaseFILE(PerlIO *f,FILE *stdio);

int     PerlIO_apply_layers(PerlIO *f, const char *mode, const char *layers);
int     PerlIO_binmode(PerlIO *f, int ptype, int imode, const char *layers);
void    PerlIO_debug(const char *fmt,...)
```

2.3 DESCRIPTION

Perl's source code, and extensions that want maximum portability, should use the above functions instead of those defined in ANSI C's *stdio.h*. The perl headers (in particular "perlio.h") will `#define` them to the I/O mechanism selected at Configure time.

The functions are modeled on those in *stdio.h*, but parameter order has been "tidied up a little".

`PerlIO *` takes the place of FILE *. Like FILE * it should be treated as opaque (it is probably safe to assume it is a pointer to something).

There are currently three implementations:

1. USE_STDIO

   All above are #define'd to stdio functions or are trivial wrapper functions which call stdio. In this case *only* PerlIO * is a FILE *. This has been the default implementation since the abstraction was introduced in perl5.003_02.

2. USE_PERLIO

   Introduced just after perl5.7.0, this is a re-implementation of the above abstraction which allows perl more control over how IO is done as it decouples IO from the way the operating system and C library choose to do things. For USE_PERLIO PerlIO * has an extra layer of indirection - it is a pointer-to-a-pointer. This allows the PerlIO * to remain with a known value while swapping the implementation around underneath *at run time*. In this case all the above are true (but very simple) functions which call the underlying implementation.

   This is the only implementation for which `PerlIO_apply_layers()` does anything "interesting".

The USE_PERLIO implementation is described in Section 35.1 [perliol NAME], page 651.

Because "perlio.h" is a thin layer (for efficiency) the semantics of these functions are somewhat dependent on the underlying implementation. Where these variations are understood they are noted below.

Unless otherwise noted, functions return 0 on success, or a negative value (usually `EOF` which is usually -1) and set `errno` on error.

PerlIO_stdin(), **PerlIO_stdout()**, **PerlIO_stderr()**

Use these rather than `stdin`, `stdout`, `stderr`. They are written to look like "function calls" rather than variables because this makes it easier to *make them* function calls if platform cannot export data to loaded modules, or if (say) different "threads" might have different values.

PerlIO_open(path, mode), **PerlIO_fdopen(fd,mode)**

These correspond to fopen()/fdopen() and the arguments are the same. Return `NULL` and set `errno` if there is an error. There may be an implementation limit on the number of open handles, which may be lower than the limit on the number of open files - `errno` may not be set when `NULL` is returned if this limit is exceeded.

PerlIO_reopen(path,mode,f)

While this currently exists in all three implementations perl itself does not use it. *As perl does not use it, it is not well tested.*

Perl prefers to `dup` the new low-level descriptor to the descriptor used by the existing PerlIO. This may become the behaviour of this function in the future.

PerlIO_printf(f,fmt,...), **PerlIO_vprintf(f,fmt,a)**

These are fprintf()/vfprintf() equivalents.

PerlIO_stdoutf(fmt,...)

This is printf() equivalent. printf is #defined to this function, so it is (currently) legal to use `printf(fmt,...)` in perl sources.

PerlIO_read(f,buf,count), **PerlIO_write(f,buf,count)**

These correspond functionally to fread() and fwrite() but the arguments and return values are different. The PerlIO_read() and PerlIO_write() signatures have been modeled on the more sane low level read() and write() functions instead: The "file" argument is passed first, there is only one "count", and the return value can distinguish between error and `EOF`.

Returns a byte count if successful (which may be zero or positive), returns negative value and sets `errno` on error. Depending on implementation `errno` may be `EINTR` if operation was interrupted by a signal.

PerlIO_close(f)

Depending on implementation `errno` may be `EINTR` if operation was interrupted by a signal.

PerlIO_puts(f,s), **PerlIO_putc(f,c)**

These correspond to fputs() and fputc(). Note that arguments have been revised to have "file" first.

PerlIO_ungetc(f,c)

This corresponds to ungetc(). Note that arguments have been revised to have "file" first. Arranges that next read operation will return the byte **c**. Despite the implied "character" in the name only values in the range 0..0xFF are defined. Returns the byte **c** on success or -1 (`EOF`) on error. The number of bytes that can be "pushed back" may vary, only 1 character is certain, and then only if it is the last character that was read from the handle.

PerlIO_getc(f)

This corresponds to getc(). Despite the c in the name only byte range 0..0xFF is supported. Returns the character read or -1 (`EOF`) on error.

PerlIO_eof(f)

This corresponds to feof(). Returns a true/false indication of whether the handle is at end of file. For terminal devices this may or may not be "sticky" depending on the implementation. The flag is cleared by PerlIO_seek(), or PerlIO_rewind().

PerlIO_error(f)

This corresponds to ferror(). Returns a true/false indication of whether there has been an IO error on the handle.

PerlIO_fileno(f)

This corresponds to fileno(), note that on some platforms, the meaning of "fileno" may not match Unix. Returns -1 if the handle has no open descriptor associated with it.

PerlIO_clearerr(f)

This corresponds to clearerr(), i.e., clears 'error' and (usually) 'eof' flags for the "stream". Does not return a value.

PerlIO_flush(f)

This corresponds to fflush(). Sends any buffered write data to the underlying file. If called with `NULL` this may flush all open streams (or core dump with some USE_STDIO implementations). Calling on a handle open for read only, or on which last operation was a read of some kind may lead to undefined behaviour on some USE_STDIO implementations. The USE_PERLIO (layers) implementation tries to behave better: it flushes all open streams when passed `NULL`, and attempts to retain data on read streams either in the buffer or by seeking the handle to the current logical position.

PerlIO_seek(f,offset,whence)

This corresponds to fseek(). Sends buffered write data to the underlying file, or discards any buffered read data, then positions the file descriptor as specified by **offset** and **whence** (sic). This is the correct thing to do when switching between read and write on the same handle (see issues with PerlIO_flush() above). Offset is of type `Off_t` which is a perl Configure value which may not be same as stdio's `off_t`.

PerlIO_tell(f)

This corresponds to ftell(). Returns the current file position, or (Off_t) -1 on error. May just return value system "knows" without making a system call or checking the underlying file descriptor (so use on shared file descriptors is not safe without a PerlIO_seek()). Return value is of type `Off_t` which is a perl Configure value which may not be same as stdio's `off_t`.

PerlIO_getpos(f,p), **PerlIO_setpos(f,p)**

These correspond (loosely) to fgetpos() and fsetpos(). Rather than stdio's Fpos_t they expect a "Perl Scalar Value" to be passed. What is stored there should be considered opaque. The layout of the data may vary from handle to handle. When not using stdio or if platform does not have the stdio calls then they are implemented in terms of PerlIO_tell() and PerlIO_seek().

PerlIO_rewind(f)

This corresponds to rewind(). It is usually defined as being

```
PerlIO_seek(f,(Off_t)0L, SEEK_SET);
PerlIO_clearerr(f);
```

PerlIO_tmpfile()

This corresponds to tmpfile(), i.e., returns an anonymous PerlIO or NULL on error. The system will attempt to automatically delete the file when closed. On Unix the file is usually `unlink`-ed just after it is created so it does not matter how it gets closed. On other systems the file may only be deleted if closed via PerlIO_close() and/or the program exits via `exit`. Depending on the implementation there may be "race conditions" which allow other processes access to the file, though in general it will be safer in this regard than ad. hoc. schemes.

PerlIO_setlinebuf(f)

This corresponds to setlinebuf(). Does not return a value. What constitutes a "line" is implementation dependent but usually means that writing "\n" flushes the buffer. What happens with things like "this\nthat" is uncertain. (Perl core uses it *only* when "dumping"; it has nothing to do with $| auto-flush.)

2.3.1 Co-existence with stdio

There is outline support for co-existence of PerlIO with stdio. Obviously if PerlIO is implemented in terms of stdio there is no problem. However in other cases then mechanisms must exist to create a FILE * which can be passed to library code which is going to use stdio calls.

The first step is to add this line:

```
#define PERLIO_NOT_STDIO 0
```

before including any perl header files. (This will probably become the default at some point). That prevents "perlio.h" from attempting to #define stdio functions onto PerlIO functions.

XS code is probably better using "typemap" if it expects FILE * arguments. The standard typemap will be adjusted to comprehend any changes in this area.

PerlIO_importFILE(f,mode)

Used to get a PerlIO * from a FILE *.

The mode argument should be a string as would be passed to fopen/PerlIO_open. If it is NULL then - for legacy support - the code will (depending upon the platform and the implementation) either attempt to empirically determine the mode in which *f* is open, or use `"r+"` to indicate a read/write stream.

Once called the FILE * should *ONLY* be closed by calling `PerlIO_close()` on the returned PerlIO *.

The PerlIO is set to textmode. Use PerlIO_binmode if this is not the desired mode.

This is **not** the reverse of PerlIO_exportFILE().

PerlIO_exportFILE(f,mode)

Given a PerlIO * create a 'native' FILE * suitable for passing to code expecting to be compiled and linked with ANSI C *stdio.h*. The mode argument should be a string as would be passed to fopen/PerlIO_open. If it is NULL then - for legacy support - the FILE * is opened in same mode as the PerlIO *.

The fact that such a FILE * has been 'exported' is recorded, (normally by pushing a new :stdio `"layer"` onto the PerlIO *), which may affect future PerlIO operations on the original PerlIO *. You should not call `fclose()` on the file unless you call `PerlIO_releaseFILE()` to disassociate it from the PerlIO *. (Do not use PerlIO_importFILE() for doing the disassociation.)

Calling this function repeatedly will create a FILE * on each call (and will push an :stdio layer each time as well).

PerlIO_releaseFILE(p,f)

Calling PerlIO_releaseFILE informs PerlIO that all use of FILE * is complete. It is removed from the list of 'exported' FILE *s, and the associated PerlIO * should revert to its original behaviour.

Use this to disassociate a file from a PerlIO * that was associated using PerlIO_exportFILE().

PerlIO_findFILE(f)

Returns a native FILE * used by a stdio layer. If there is none, it will create one with PerlIO_exportFILE. In either case the FILE * should be considered as belonging to PerlIO subsystem and should only be closed by calling `PerlIO_close()`.

2.3.2 "Fast gets" Functions

In addition to standard-like API defined so far above there is an `"implementation"` interface which allows perl to get at internals of PerlIO. The following calls correspond to the various FILE_xxx macros determined by Configure - or their equivalent in other implementations. This section is really of interest to only those concerned with detailed perl-core behaviour, implementing a PerlIO mapping or writing code which can make use of the `"read ahead"` that has been done by the IO system in the same way perl does. Note that any code that uses these interfaces must be prepared to do things the traditional way if a handle does not support them.

PerlIO_fast_gets(f)

Returns true if implementation has all the interfaces required to allow perl's `sv_gets` to "bypass" normal IO mechanism. This can vary from handle to handle.

```
  PerlIO_fast_gets(f) = PerlIO_has_cntptr(f) && \
                        PerlIO_canset_cnt(f) && \
                        'Can set pointer into buffer'
```

PerlIO_has_cntptr(f)

Implementation can return pointer to current position in the "buffer" and a count of bytes available in the buffer. Do not use this - use PerlIO_fast_gets.

PerlIO_get_cnt(f)

Return count of readable bytes in the buffer. Zero or negative return means no more bytes available.

PerlIO_get_ptr(f)

Return pointer to next readable byte in buffer, accessing via the pointer (dereferencing) is only safe if PerlIO_get_cnt() has returned a positive value. Only positive offsets up to value returned by PerlIO_get_cnt() are allowed.

PerlIO_set_ptrcnt(f,p,c)

Set pointer into buffer, and a count of bytes still in the buffer. Should be used only to set pointer to within range implied by previous calls to `PerlIO_get_ptr` and `PerlIO_get_cnt`. The two values *must* be consistent with each other (implementation may only use one or the other or may require both).

PerlIO_canset_cnt(f)

Implementation can adjust its idea of number of bytes in the buffer. Do not use this - use PerlIO_fast_gets.

PerlIO_set_cnt(f,c)

Obscure - set count of bytes in the buffer. Deprecated. Only usable if PerlIO_canset_cnt() returns true. Currently used in only doio.c to force count less than -1 to -1. Perhaps should be PerlIO_set_empty or similar. This call may actually do nothing if "count" is deduced from pointer and a "limit". Do not use this - use PerlIO_set_ptrcnt().

PerlIO_has_base(f)

Returns true if implementation has a buffer, and can return pointer to whole buffer and its size. Used by perl for **-T** / **-B** tests. Other uses would be very obscure...

PerlIO_get_base(f)

Return *start* of buffer. Access only positive offsets in the buffer up to the value returned by PerlIO_get_bufsiz().

PerlIO_get_bufsiz(f)

Return the *total number of bytes* in the buffer, this is neither the number that can be read, nor the amount of memory allocated to the buffer. Rather it is what the operating system and/or implementation happened to `read()` (or whatever) last time IO was requested.

2.3.3 Other Functions

PerlIO_apply_layers(f,mode,layers)

The new interface to the USE_PERLIO implementation. The layers ":crlf" and ":raw" are only ones allowed for other implementations and those are silently ignored. (As of perl5.8 ":raw" is deprecated.) Use PerlIO_binmode() below for the portable case.

PerlIO_binmode(f,ptype,imode,layers)

The hook used by perl's `binmode` operator. **ptype** is perl's character for the kind of IO:

'<' read

'>' write

'+' read/write

imode is `O_BINARY` or `O_TEXT`.

layers is a string of layers to apply, only ":crlf" makes sense in the non USE_PERLIO case. (As of perl5.8 ":raw" is deprecated in favour of passing NULL.)

Portable cases are:

```
    PerlIO_binmode(f,ptype,O_BINARY,NULL);
and
    PerlIO_binmode(f,ptype,O_TEXT,":crlf");
```

On Unix these calls probably have no effect whatsoever. Elsewhere they alter "\n" to CR,LF translation and possibly cause a special text "end of file" indicator to be written or honoured on read. The effect of making the call after doing any IO to the handle depends on the implementation. (It may be ignored, affect any data which is already buffered as well, or only apply to subsequent data.)

PerlIO_debug(fmt,...)

PerlIO_debug is a printf()-like function which can be used for debugging. No return value. Its main use is inside PerlIO where using real printf, warn() etc. would recursively call PerlIO and be a problem.

PerlIO_debug writes to the file named by $ENV{'PERLIO_DEBUG'} typical use might be

```
  Bourne shells (sh, ksh, bash, zsh, ash, ...):
   PERLIO_DEBUG=/dev/tty ./perl somescript some args

  Csh/Tcsh:
   setenv PERLIO_DEBUG /dev/tty
   ./perl somescript some args

  If you have the "env" utility:
   env PERLIO_DEBUG=/dev/tty ./perl somescript some args

  Win32:
```

```
set PERLIO_DEBUG=CON
perl somescript some args
```

If $ENV{'PERLIO_DEBUG'} is not set PerlIO_debug() is a no-op.

3 perlartistic

3.1 NAME

perlartistic - the Perl Artistic License

3.2 SYNOPSIS

```
 You can refer to this document in Pod via "L<perlartistic>"
 Or you can see this document by entering "perldoc perlartistic"
```

3.3 DESCRIPTION

Perl is free software; you can redistribute it and/or modify it under the terms of either:

```
        a) the GNU General Public License as published by the Free
        Software Foundation; either version 1, or (at your option) any
        later version, or

        b) the "Artistic License" which comes with this Kit.
```

This is "**The Artistic License**". It's here so that modules, programs, etc., that want to declare this as their distribution license can link to it.

For the GNU General Public License, see Section 27.1 [perlgpl NAME], page 506.

3.4 The "Artistic License"

3.4.1 Preamble

The intent of this document is to state the conditions under which a Package may be copied, such that the Copyright Holder maintains some semblance of artistic control over the development of the package, while giving the users of the package the right to use and distribute the Package in a more-or-less customary fashion, plus the right to make reasonable modifications.

3.4.2 Definitions

"Package"

refers to the collection of files distributed by the Copyright Holder, and derivatives of that collection of files created through textual modification.

"Standard Version"

refers to such a Package if it has not been modified, or has been modified in accordance with the wishes of the Copyright Holder as specified below.

"Copyright Holder"

is whoever is named in the copyright or copyrights for the package.

"You"

is you, if you're thinking about copying or distributing this Package.

"Reasonable copying fee"

is whatever you can justify on the basis of media cost, duplication charges, time of people involved, and so on. (You will not be required to justify it to the Copyright Holder, but only to the computing community at large as a market that must bear the fee.)

"Freely Available"

means that no fee is charged for the item itself, though there may be fees involved in handling the item. It also means that recipients of the item may redistribute it under the same conditions they received it.

3.4.3 Conditions

1. You may make and give away verbatim copies of the source form of the Standard Version of this Package without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.
2. You may apply bug fixes, portability fixes and other modifications derived from the Public Domain or from the Copyright Holder. A Package modified in such a way shall still be considered the Standard Version.
3. You may otherwise modify your copy of this Package in any way, provided that you insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:

   a)

   place your modifications in the Public Domain or otherwise make them Freely Available, such as by posting said modifications to Usenet or an equivalent medium, or placing the modifications on a major archive site such as uunet.uu.net, or by allowing the Copyright Holder to include your modifications in the Standard Version of the Package.

   b)

   use the modified Package only within your corporation or organization.

   c)

   rename any non-standard executables so the names do not conflict with standard executables, which must also be provided, and provide a separate manual page for each non-standard executable that clearly documents how it differs from the Standard Version.

   d)

   make other distribution arrangements with the Copyright Holder.
4. You may distribute the programs of this Package in object code or executable form, provided that you do at least ONE of the following:

   a)

   distribute a Standard Version of the executables and library files, together with instructions (in the manual page or equivalent) on where to get the Standard Version.

b) accompany the distribution with the machine-readable source of the Package with your modifications.

c) give non-standard executables non-standard names, and clearly document the differences in manual pages (or equivalent), together with instructions on where to get the Standard Version.

d) make other distribution arrangements with the Copyright Holder.

5. You may charge a reasonable copying fee for any distribution of this Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Package as a product of your own. You may embed this Package's interpreter within an executable of yours (by linking); this shall be construed as a mere form of aggregation, provided that the complete Standard Version of the interpreter is so embedded.
6. The scripts and library files supplied as input to or produced as output from the programs of this Package do not automatically fall under the copyright of this Package, but belong to whoever generated them, and may be sold commercially, and may be aggregated with this Package. If such scripts or library files are aggregated with this Package via the so-called "undump" or "unexec" methods of producing a binary executable image, then distribution of such an image shall neither be construed as a distribution of this Package nor shall it fall under the restrictions of Paragraphs 3 and 4, provided that you do not represent such an executable image as a Standard Version of this Package.
7. C subroutines (or comparably compiled subroutines in other languages) supplied by you and linked into this Package in order to emulate subroutines and variables of the language defined by this Package shall not be considered part of this Package, but are the equivalent of input as in Paragraph 6, provided these subroutines do not change the language in any way that would cause it to fail the regression tests for the language.
8. Aggregation of this Package with a commercial distribution is always permitted provided that the use of this Package is embedded; that is, when no overt attempt is made to make this Package's interfaces visible to the end user of the commercial distribution. Such use shall not be construed as a distribution of this Package.
9. The name of the Copyright Holder may not be used to endorse or promote products derived from this software without specific prior written permission.
10. THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

The End

4 perlbook

4.1 NAME

perlbook - Books about and related to Perl

4.2 DESCRIPTION

There are many books on Perl and Perl-related. A few of these are good, some are OK, but many aren't worth your money. There is a list of these books, some with extensive reviews, at http://books.perl.org/ . We list some of the books here, and while listing a book implies our endorsement, don't think that not including a book means anything.

Most of these books are available online through Safari Books Online (http://safaribooksonline.com/).

4.2.1 The most popular books

The major reference book on Perl, written by the creator of Perl, is *Programming Perl*:

Programming Perl (the "Camel Book"):

```
by Tom Christiansen, brian d foy, Larry Wall with Jon Orwant
ISBN 978-0-596-00492-7 [4th edition February 2012]
ISBN 978-1-4493-9890-3 [ebook]
http://oreilly.com/catalog/9780596004927
```

The Ram is a cookbook with hundreds of examples of using Perl to accomplish specific tasks:

The Perl Cookbook (the "Ram Book"):

```
by Tom Christiansen and Nathan Torkington,
    with Foreword by Larry Wall
ISBN 978-0-596-00313-5 [2nd Edition August 2003]
ISBN 978-0-596-15888-0 [ebook]
http://oreilly.com/catalog/9780596003135/
```

If you want to learn the basics of Perl, you might start with the Llama book, which assumes that you already know a little about programming:

Learning Perl (the "Llama Book")

```
by Randal L. Schwartz, Tom Phoenix, and brian d foy
ISBN 978-1-4493-0358-7 [6th edition June 2011]
ISBN 978-1-4493-0458-4 [ebook]
http://www.learning-perl.com/
```

The tutorial started in the Llama continues in the Alpaca, which introduces the intermediate features of references, data structures, object-oriented programming, and modules:

Intermediate Perl (the "Alpaca Book")

```
by Randal L. Schwartz and brian d foy, with Tom Phoenix
        foreword by Damian Conway
ISBN 978-1-4493-9309-0 [2nd edition August 2012]
ISBN 978-1-4493-0459-1 [ebook]
http://www.intermediateperl.com/
```

4.2.2 References

You might want to keep these desktop references close by your keyboard:

Perl 5 Pocket Reference

```
by Johan Vromans
ISBN 978-1-4493-0370-9 [5th edition July 2011]
ISBN 978-1-4493-0813-1 [ebook]
http://oreilly.com/catalog/0636920018476/
```

Perl Debugger Pocket Reference

```
by Richard Foley
ISBN 978-0-596-00503-0 [1st edition January 2004]
ISBN 978-0-596-55625-9 [ebook]
http://oreilly.com/catalog/9780596005030/
```

Regular Expression Pocket Reference

```
by Tony Stubblebine
ISBN 978-0-596-51427-3 [2nd edition July 2007]
ISBN 978-0-596-55782-9 [ebook]
http://oreilly.com/catalog/9780596514273/
```

4.2.3 Tutorials

Beginning Perl

```
by James Lee
ISBN 1-59059-391-X [3rd edition April 2010 & ebook]
http://www.apress.com/9781430227939
```

Learning Perl (the "Llama Book")

```
by Randal L. Schwartz, Tom Phoenix, and brian d foy
ISBN 978-1-4493-0358-7 [6th edition June 2011]
ISBN 978-1-4493-0458-4 [ebook]
http://www.learning-perl.com/
```

Intermediate Perl (the "Alpaca Book")

```
by Randal L. Schwartz and brian d foy, with Tom Phoenix
        foreword by Damian Conway
    ISBN 978-1-4493-9309-0 [2nd edition August 2012]
    ISBN 978-1-4493-0459-1 [ebook]
    http://www.intermediateperl.com/
```

Mastering Perl

```
by brian d foy
    ISBN 9978-1-4493-9311-3 [2st edition January 2014]
    ISBN 978-1-4493-6487-8 [ebook]
    http://www.masteringperl.org/
```

Effective Perl Programming

```
by Joseph N. Hall, Joshua A. McAdams, brian d foy
```

ISBN 0-321-49694-9 [2nd edition 2010]
http://www.effectiveperlprogramming.com/

4.2.4 Task-Oriented

Writing Perl Modules for CPAN
by Sam Tregar
ISBN 1-59059-018-X [1st edition August 2002 & ebook]
http://www.apress.com/9781590590188

The Perl Cookbook
by Tom Christiansen and Nathan Torkington,
with Foreword by Larry Wall
ISBN 978-0-596-00313-5 [2nd Edition August 2003]
ISBN 978-0-596-15888-0 [ebook]
http://oreilly.com/catalog/9780596003135/

Automating System Administration with Perl
by David N. Blank-Edelman
ISBN 978-0-596-00639-6 [2nd edition May 2009]
ISBN 978-0-596-80251-6 [ebook]
http://oreilly.com/catalog/9780596006396

Real World SQL Server Administration with Perl
by Linchi Shea
ISBN 1-59059-097-X [1st edition July 2003 & ebook]
http://www.apress.com/9781590590973

4.2.5 Special Topics

Regular Expressions Cookbook
by Jan Goyvaerts and Steven Levithan
ISBN 978-1-4493-1943-4 [2nd edition August 2012]
ISBN 978-1-4493-2747-7 [ebook]
http://shop.oreilly.com/product/0636920023630.do

Programming the Perl DBI
by Tim Bunce and Alligator Descartes
ISBN 978-1-56592-699-8 [February 2000]
ISBN 978-1-4493-8670-2 [ebook]
http://oreilly.com/catalog/9781565926998

Perl Best Practices
by Damian Conway
ISBN 978-0-596-00173-5 [1st edition July 2005]
ISBN 978-0-596-15900-9 [ebook]
http://oreilly.com/catalog/9780596001735

Higher-Order Perl
by Mark-Jason Dominus
ISBN 1-55860-701-3 [1st edition March 2005]
free ebook http://hop.perl.plover.com/book/

```
http://hop.perl.plover.com/
```

Mastering Regular Expressions

```
by Jeffrey E. F. Friedl
ISBN 978-0-596-52812-6 [3rd edition August 2006]
ISBN 978-0-596-55899-4 [ebook]
http://oreilly.com/catalog/9780596528126
```

Network Programming with Perl

```
by Lincoln Stein
ISBN 0-201-61571-1 [1st edition 2001]
http://www.pearsonhighered.com/educator/product/Network-Programming-with-
```

Perl Template Toolkit

```
by Darren Chamberlain, Dave Cross, and Andy Wardley
ISBN 978-0-596-00476-7 [December 2003]
ISBN 978-1-4493-8647-4 [ebook]
http://oreilly.com/catalog/9780596004767
```

Object Oriented Perl

```
by Damian Conway
    with foreword by Randal L. Schwartz
ISBN 1-884777-79-1 [1st edition August 1999 & ebook]
http://www.manning.com/conway/
```

Data Munging with Perl

```
by Dave Cross
ISBN 1-930110-00-6 [1st edition 2001 & ebook]
http://www.manning.com/cross
```

Mastering Perl/Tk

```
by Steve Lidie and Nancy Walsh
ISBN 978-1-56592-716-2 [1st edition January 2002]
ISBN 978-0-596-10344-6 [ebook]
http://oreilly.com/catalog/9781565927162
```

Extending and Embedding Perl

```
by Tim Jenness and Simon Cozens
ISBN 1-930110-82-0 [1st edition August 2002 & ebook]
http://www.manning.com/jenness
```

Pro Perl Debugging

```
by Richard Foley with Andy Lester
ISBN 1-59059-454-1 [1st edition July 2005 & ebook]
http://www.apress.com/9781590594544
```

4.2.6 Free (as in beer) books

Some of these books are available as free downloads.

Higher-Order Perl: http://hop.perl.plover.com/

4.2.7 Other interesting, non-Perl books

You might notice several familiar Perl concepts in this collection of ACM columns from Jon Bentley. The similarity to the title of the major Perl book (which came later) is not completely accidental:

Programming Pearls

```
by Jon Bentley
ISBN 978-0-201-65788-3 [2 edition, October 1999]
```

More Programming Pearls

```
by Jon Bentley
ISBN 0-201-11889-0 [January 1988]
```

4.2.8 A note on freshness

Each version of Perl comes with the documentation that was current at the time of release. This poses a problem for content such as book lists. There are probably very nice books published after this list was included in your Perl release, and you can check the latest released version at http://perldoc.perl.org/perlbook.html .

Some of the books we've listed appear almost ancient in internet scale, but we've included those books because they still describe the current way of doing things. Not everything in Perl changes every day. Many of the beginner-level books, too, go over basic features and techniques that are still valid today. In general though, we try to limit this list to books published in the past five years.

4.2.9 Get your book listed

If your Perl book isn't listed and you think it should be, let us know.

5 perlboot

5.1 NAME

perlboot - Links to information on object-oriented programming in Perl

5.2 DESCRIPTION

For information on OO programming with Perl, please see Section 47.1 [perlootut NAME], page 786 and Section 46.1 [perlobj NAME], page 769.

(The above documents supersede the tutorial that was formerly here in perlboot.)

6 perlbot

6.1 NAME

perlbot - Links to information on object-oriented programming in Perl

6.2 DESCRIPTION

For information on OO programming with Perl, please see Section 47.1 [perlootut NAME], page 786 and Section 46.1 [perlobj NAME], page 769.

(The above documents supersede the collection of tricks that was formerly here in perlbot.)

7 perlcall

7.1 NAME

perlcall - Perl calling conventions from C

7.2 DESCRIPTION

The purpose of this document is to show you how to call Perl subroutines directly from C, i.e., how to write *callbacks*.

Apart from discussing the C interface provided by Perl for writing callbacks the document uses a series of examples to show how the interface actually works in practice. In addition some techniques for coding callbacks are covered.

Examples where callbacks are necessary include

- An Error Handler

  You have created an XSUB interface to an application's C API.

  A fairly common feature in applications is to allow you to define a C function that will be called whenever something nasty occurs. What we would like is to be able to specify a Perl subroutine that will be called instead.

- An Event-Driven Program

  The classic example of where callbacks are used is when writing an event driven program, such as for an X11 application. In this case you register functions to be called whenever specific events occur, e.g., a mouse button is pressed, the cursor moves into a window or a menu item is selected.

Although the techniques described here are applicable when embedding Perl in a C program, this is not the primary goal of this document. There are other details that must be considered and are specific to embedding Perl. For details on embedding Perl in C refer to Section 20.1 [perlembed NAME], page 301.

Before you launch yourself head first into the rest of this document, it would be a good idea to have read the following two documents–`perlxs` and Section 28.1 [perlguts NAME], page 512.

7.3 THE CALL_ FUNCTIONS

Although this stuff is easier to explain using examples, you first need be aware of a few important definitions.

Perl has a number of C functions that allow you to call Perl subroutines. They are

```
I32 call_sv(SV* sv, I32 flags);
I32 call_pv(char *subname, I32 flags);
I32 call_method(char *methname, I32 flags);
I32 call_argv(char *subname, I32 flags, char **argv);
```

The key function is *call_sv*. All the other functions are fairly simple wrappers which make it easier to call Perl subroutines in special cases. At the end of the day they will all call *call_sv* to invoke the Perl subroutine.

All the *call_** functions have a `flags` parameter which is used to pass a bit mask of options to Perl. This bit mask operates identically for each of the functions. The settings available in the bit mask are discussed in Section 7.4 [FLAG VALUES], page 29.

Each of the functions will now be discussed in turn.

call_sv

> *call_sv* takes two parameters. The first, `sv`, is an SV*. This allows you to specify the Perl subroutine to be called either as a C string (which has first been converted to an SV) or a reference to a subroutine. The section, *Using call_sv*, shows how you can make use of *call_sv*.

call_pv

> The function, *call_pv*, is similar to *call_sv* except it expects its first parameter to be a C char* which identifies the Perl subroutine you want to call, e.g., `call_pv("fred", 0)`. If the subroutine you want to call is in another package, just include the package name in the string, e.g., `"pkg::fred"`.

call_method

> The function *call_method* is used to call a method from a Perl class. The parameter `methname` corresponds to the name of the method to be called. Note that the class that the method belongs to is passed on the Perl stack rather than in the parameter list. This class can be either the name of the class (for a static method) or a reference to an object (for a virtual method). See Section 46.1 [perlobj NAME], page 769 for more information on static and virtual methods and Section 7.5.11 [Using call_method], page 45 for an example of using *call_method*.

call_argv

> *call_argv* calls the Perl subroutine specified by the C string stored in the `subname` parameter. It also takes the usual `flags` parameter. The final parameter, `argv`, consists of a NULL-terminated list of C strings to be passed as parameters to the Perl subroutine. See *Using call_argv*.

All the functions return an integer. This is a count of the number of items returned by the Perl subroutine. The actual items returned by the subroutine are stored on the Perl stack.

As a general rule you should *always* check the return value from these functions. Even if you are expecting only a particular number of values to be returned from the Perl subroutine, there is nothing to stop someone from doing something unexpected–don't say you haven't been warned.

7.4 FLAG VALUES

The `flags` parameter in all the *call_** functions is one of G_VOID, G_SCALAR, or G_ARRAY, which indicate the call context, OR'ed together with a bit mask of any combination of the other G_* symbols defined below.

7.4.1 G_VOID

Calls the Perl subroutine in a void context.

This flag has 2 effects:

1. It indicates to the subroutine being called that it is executing in a void context (if it executes *wantarray* the result will be the undefined value).
2. It ensures that nothing is actually returned from the subroutine.

The value returned by the *call_** function indicates how many items have been returned by the Perl subroutine–in this case it will be 0.

7.4.2 G_SCALAR

Calls the Perl subroutine in a scalar context. This is the default context flag setting for all the *call_** functions.

This flag has 2 effects:

1. It indicates to the subroutine being called that it is executing in a scalar context (if it executes *wantarray* the result will be false).
2. It ensures that only a scalar is actually returned from the subroutine. The subroutine can, of course, ignore the *wantarray* and return a list anyway. If so, then only the last element of the list will be returned.

The value returned by the *call_** function indicates how many items have been returned by the Perl subroutine - in this case it will be either 0 or 1.

If 0, then you have specified the G_DISCARD flag.

If 1, then the item actually returned by the Perl subroutine will be stored on the Perl stack - the section *Returning a Scalar* shows how to access this value on the stack. Remember that regardless of how many items the Perl subroutine returns, only the last one will be accessible from the stack - think of the case where only one value is returned as being a list with only one element. Any other items that were returned will not exist by the time control returns from the *call_** function. The section *Returning a list in a scalar context* shows an example of this behavior.

7.4.3 G_ARRAY

Calls the Perl subroutine in a list context.

As with G_SCALAR, this flag has 2 effects:

1. It indicates to the subroutine being called that it is executing in a list context (if it executes *wantarray* the result will be true).
2. It ensures that all items returned from the subroutine will be accessible when control returns from the *call_** function.

The value returned by the *call_** function indicates how many items have been returned by the Perl subroutine.

If 0, then you have specified the G_DISCARD flag.

If not 0, then it will be a count of the number of items returned by the subroutine. These items will be stored on the Perl stack. The section *Returning a list of values* gives an example of using the G_ARRAY flag and the mechanics of accessing the returned items from the Perl stack.

7.4.4 G_DISCARD

By default, the *call_** functions place the items returned from by the Perl subroutine on the stack. If you are not interested in these items, then setting this flag will make Perl get rid of them automatically for you. Note that it is still possible to indicate a context to the Perl subroutine by using either G_SCALAR or G_ARRAY.

If you do not set this flag then it is *very* important that you make sure that any temporaries (i.e., parameters passed to the Perl subroutine and values returned from the subroutine) are disposed of yourself. The section *Returning a Scalar* gives details of how to dispose of these temporaries explicitly and the section *Using Perl to dispose of temporaries* discusses the specific circumstances where you can ignore the problem and let Perl deal with it for you.

7.4.5 G_NOARGS

Whenever a Perl subroutine is called using one of the *call_** functions, it is assumed by default that parameters are to be passed to the subroutine. If you are not passing any parameters to the Perl subroutine, you can save a bit of time by setting this flag. It has the effect of not creating the `@_` array for the Perl subroutine.

Although the functionality provided by this flag may seem straightforward, it should be used only if there is a good reason to do so. The reason for being cautious is that, even if you have specified the G_NOARGS flag, it is still possible for the Perl subroutine that has been called to think that you have passed it parameters.

In fact, what can happen is that the Perl subroutine you have called can access the `@_` array from a previous Perl subroutine. This will occur when the code that is executing the *call_** function has itself been called from another Perl subroutine. The code below illustrates this

```
sub fred
  { print "@_\n"  }

sub joe
  { &fred }

&joe(1,2,3);
```

This will print

```
1 2 3
```

What has happened is that `fred` accesses the `@_` array which belongs to `joe`.

7.4.6 G_EVAL

It is possible for the Perl subroutine you are calling to terminate abnormally, e.g., by calling *die* explicitly or by not actually existing. By default, when either of these events occurs, the process will terminate immediately. If you want to trap this type of event, specify the G_EVAL flag. It will put an *eval { }* around the subroutine call.

Whenever control returns from the *call_** function you need to check the `$@` variable as you would in a normal Perl script.

The value returned from the *call_** function is dependent on what other flags have been specified and whether an error has occurred. Here are all the different cases that can occur:

- If the *call_** function returns normally, then the value returned is as specified in the previous sections.
- If G_DISCARD is specified, the return value will always be 0.
- If G_ARRAY is specified *and* an error has occurred, the return value will always be 0.
- If G_SCALAR is specified *and* an error has occurred, the return value will be 1 and the value on the top of the stack will be *undef*. This means that if you have already detected the error by checking `$@` and you want the program to continue, you must remember to pop the *undef* from the stack.

See *Using G_EVAL* for details on using G_EVAL.

7.4.7 G_KEEPERR

Using the G_EVAL flag described above will always set `$@`: clearing it if there was no error, and setting it to describe the error if there was an error in the called code. This is what you want if your intention is to handle possible errors, but sometimes you just want to trap errors and stop them interfering with the rest of the program.

This scenario will mostly be applicable to code that is meant to be called from within destructors, asynchronous callbacks, and signal handlers. In such situations, where the code being called has little relation to the surrounding dynamic context, the main program needs to be insulated from errors in the called code, even if they can't be handled intelligently. It may also be useful to do this with code for `__DIE__` or `__WARN__` hooks, and `tie` functions.

The G_KEEPERR flag is meant to be used in conjunction with G_EVAL in *call_** functions that are used to implement such code, or with `eval_sv`. This flag has no effect on the `call_*` functions when G_EVAL is not used.

When G_KEEPERR is used, any error in the called code will terminate the call as usual, and the error will not propagate beyond the call (as usual for G_EVAL), but it will not go into `$@`. Instead the error will be converted into a warning, prefixed with the string `"\t(in cleanup)"`. This can be disabled using `no warnings 'misc'`. If there is no error, `$@` will not be cleared.

Note that the G_KEEPERR flag does not propagate into inner evals; these may still set `$@`.

The G_KEEPERR flag was introduced in Perl version 5.002.

See *Using G_KEEPERR* for an example of a situation that warrants the use of this flag.

7.4.8 Determining the Context

As mentioned above, you can determine the context of the currently executing subroutine in Perl with *wantarray*. The equivalent test can be made in C by using the `GIMME_V` macro, which returns `G_ARRAY` if you have been called in a list context, `G_SCALAR` if in a scalar context, or `G_VOID` if in a void context (i.e., the return value will not be used). An older version of this macro is called `GIMME`; in a void context it returns `G_SCALAR` instead of `G_VOID`. An example of using the `GIMME_V` macro is shown in section *Using GIMME_V*.

7.5 EXAMPLES

Enough of the definition talk! Let's have a few examples.

Perl provides many macros to assist in accessing the Perl stack. Wherever possible, these macros should always be used when interfacing to Perl internals. We hope this should make the code less vulnerable to any changes made to Perl in the future.

Another point worth noting is that in the first series of examples I have made use of only the *call_pv* function. This has been done to keep the code simpler and ease you into the topic. Wherever possible, if the choice is between using *call_pv* and *call_sv*, you should always try to use *call_sv*. See *Using call_sv* for details.

7.5.1 No Parameters, Nothing Returned

This first trivial example will call a Perl subroutine, *PrintUID*, to print out the UID of the process.

```
sub PrintUID
{
    print "UID is $<\n";
}
```

and here is a C function to call it

```
static void
call_PrintUID()
{
    dSP;

    PUSHMARK(SP);
    call_pv("PrintUID", G_DISCARD|G_NOARGS);
}
```

Simple, eh?

A few points to note about this example:

1. Ignore `dSP` and `PUSHMARK(SP)` for now. They will be discussed in the next example.
2. We aren't passing any parameters to *PrintUID* so G_NOARGS can be specified.
3. We aren't interested in anything returned from *PrintUID*, so G_DISCARD is specified. Even if *PrintUID* was changed to return some value(s), having specified G_DISCARD will mean that they will be wiped by the time control returns from *call_pv*.
4. As *call_pv* is being used, the Perl subroutine is specified as a C string. In this case the subroutine name has been 'hard-wired' into the code.
5. Because we specified G_DISCARD, it is not necessary to check the value returned from *call_pv*. It will always be 0.

7.5.2 Passing Parameters

Now let's make a slightly more complex example. This time we want to call a Perl subroutine, `LeftString`, which will take 2 parameters–a string ($s) and an integer ($n). The subroutine will simply print the first $n characters of the string.

So the Perl subroutine would look like this:

```
sub LeftString
{
    my($s, $n) = @_;
    print substr($s, 0, $n), "\n";
}
```

The C function required to call *LeftString* would look like this:

```
static void
call_LeftString(a, b)
char * a;
int b;
{
    dSP;

    ENTER;
    SAVETMPS;

    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSVpv(a, 0)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK;

    call_pv("LeftString", G_DISCARD);

    FREETMPS;
    LEAVE;
}
```

Here are a few notes on the C function *call_LeftString*.

1. Parameters are passed to the Perl subroutine using the Perl stack. This is the purpose of the code beginning with the line `dSP` and ending with the line `PUTBACK`. The `dSP` declares a local copy of the stack pointer. This local copy should **always** be accessed as `SP`.

2. If you are going to put something onto the Perl stack, you need to know where to put it. This is the purpose of the macro `dSP`–it declares and initializes a *local* copy of the Perl stack pointer.

   All the other macros which will be used in this example require you to have used this macro.

   The exception to this rule is if you are calling a Perl subroutine directly from an XSUB function. In this case it is not necessary to use the `dSP` macro explicitly–it will be declared for you automatically.

3. Any parameters to be pushed onto the stack should be bracketed by the `PUSHMARK` and `PUTBACK` macros. The purpose of these two macros, in this context, is to count the number of parameters you are pushing automatically. Then whenever Perl is creating the `@_` array for the subroutine, it knows how big to make it.

   The `PUSHMARK` macro tells Perl to make a mental note of the current stack pointer. Even if you aren't passing any parameters (like the example shown in the section *No Parameters, Nothing Returned*) you must still call the `PUSHMARK` macro before you can call any of the *call_** functions–Perl still needs to know that there are no parameters.

The `PUTBACK` macro sets the global copy of the stack pointer to be the same as our local copy. If we didn't do this, *call_pv* wouldn't know where the two parameters we pushed were–remember that up to now all the stack pointer manipulation we have done is with our local copy, *not* the global copy.

4. Next, we come to XPUSHs. This is where the parameters actually get pushed onto the stack. In this case we are pushing a string and an integer.

   See Section 28.4.1 [perlguts XSUBs and the Argument Stack], page 537 for details on how the XPUSH macros work.

5. Because we created temporary values (by means of sv_2mortal() calls) we will have to tidy up the Perl stack and dispose of mortal SVs.

   This is the purpose of

   ```
   ENTER;
   SAVETMPS;
   ```

   at the start of the function, and

   ```
   FREETMPS;
   LEAVE;
   ```

   at the end. The `ENTER/SAVETMPS` pair creates a boundary for any temporaries we create. This means that the temporaries we get rid of will be limited to those which were created after these calls.

   The `FREETMPS/LEAVE` pair will get rid of any values returned by the Perl subroutine (see next example), plus it will also dump the mortal SVs we have created. Having `ENTER/SAVETMPS` at the beginning of the code makes sure that no other mortals are destroyed.

   Think of these macros as working a bit like `{` and `}` in Perl to limit the scope of local variables.

   See the section *Using Perl to Dispose of Temporaries* for details of an alternative to using these macros.

6. Finally, *LeftString* can now be called via the *call_pv* function. The only flag specified this time is G_DISCARD. Because we are passing 2 parameters to the Perl subroutine this time, we have not specified G_NOARGS.

7.5.3 Returning a Scalar

Now for an example of dealing with the items returned from a Perl subroutine.

Here is a Perl subroutine, *Adder*, that takes 2 integer parameters and simply returns their sum.

```
sub Adder
{
    my($a, $b) = @_;
    $a + $b;
}
```

Because we are now concerned with the return value from *Adder*, the C function required to call it is now a bit more complex.

```
static void
```

```
call_Adder(a, b)
int a;
int b;
{
    dSP;
    int count;

    ENTER;
    SAVETMPS;

    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSViv(a)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK;

    count = call_pv("Adder", G_SCALAR);

    SPAGAIN;

    if (count != 1)
        croak("Big trouble\n");

    printf ("The sum of %d and %d is %d\n", a, b, POPi);

    PUTBACK;
    FREETMPS;
    LEAVE;
}
```

Points to note this time are

1. The only flag specified this time was G_SCALAR. That means that the `@_` array will be created and that the value returned by *Adder* will still exist after the call to *call_pv*.
2. The purpose of the macro `SPAGAIN` is to refresh the local copy of the stack pointer. This is necessary because it is possible that the memory allocated to the Perl stack has been reallocated during the *call_pv* call.

   If you are making use of the Perl stack pointer in your code you must always refresh the local copy using SPAGAIN whenever you make use of the *call_** functions or any other Perl internal function.
3. Although only a single value was expected to be returned from *Adder*, it is still good practice to check the return code from *call_pv* anyway.

   Expecting a single value is not quite the same as knowing that there will be one. If someone modified *Adder* to return a list and we didn't check for that possibility and take appropriate action the Perl stack would end up in an inconsistent state. That is something you *really* don't want to happen ever.
4. The `POPi` macro is used here to pop the return value from the stack. In this case we wanted an integer, so `POPi` was used.

Here is the complete list of POP macros available, along with the types they return.

POPs	SV
POPp	pointer
POPn	double
POPi	integer
POPl	long

5. The final PUTBACK is used to leave the Perl stack in a consistent state before exiting the function. This is necessary because when we popped the return value from the stack with POPi it updated only our local copy of the stack pointer. Remember, PUTBACK sets the global stack pointer to be the same as our local copy.

7.5.4 Returning a List of Values

Now, let's extend the previous example to return both the sum of the parameters and the difference.

Here is the Perl subroutine

```
sub AddSubtract
{
   my($a, $b) = @_;
   ($a+$b, $a-$b);
}
```

and this is the C function

```
static void
call_AddSubtract(a, b)
int a;
int b;
{
    dSP;
    int count;

    ENTER;
    SAVETMPS;

    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSViv(a)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK;

    count = call_pv("AddSubtract", G_ARRAY);

    SPAGAIN;

    if (count != 2)
        croak("Big trouble\n");

    printf ("%d - %d = %d\n", a, b, POPi);
```

```
        printf ("%d + %d = %d\n", a, b, POPi);

        PUTBACK;
        FREETMPS;
        LEAVE;
    }
```

If *call_AddSubtract* is called like this

```
call_AddSubtract(7, 4);
```

then here is the output

```
7 - 4 = 3
7 + 4 = 11
```

Notes

1. We wanted list context, so G_ARRAY was used.
2. Not surprisingly `POPi` is used twice this time because we were retrieving 2 values from the stack. The important thing to note is that when using the `POP*` macros they come off the stack in *reverse* order.

7.5.5 Returning a List in a Scalar Context

Say the Perl subroutine in the previous section was called in a scalar context, like this

```
static void
call_AddSubScalar(a, b)
int a;
int b;
{
    dSP;
    int count;
    int i;

    ENTER;
    SAVETMPS;

    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSViv(a)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK;

    count = call_pv("AddSubtract", G_SCALAR);

    SPAGAIN;

    printf ("Items Returned = %d\n", count);

    for (i = 1; i <= count; ++i)
        printf ("Value %d = %d\n", i, POPi);
    PUTBACK;
    FREETMPS;
    LEAVE;
}
```

The other modification made is that *call_AddSubScalar* will print the number of items returned from the Perl subroutine and their value (for simplicity it assumes that they are integer). So if *call_AddSubScalar* is called

```
call_AddSubScalar(7, 4);
```

then the output will be

```
Items Returned = 1
Value 1 = 3
```

In this case the main point to note is that only the last item in the list is returned from the subroutine. *AddSubtract* actually made it back to *call_AddSubScalar*.

7.5.6 Returning Data from Perl via the Parameter List

It is also possible to return values directly via the parameter list–whether it is actually desirable to do it is another matter entirely.

The Perl subroutine, *Inc*, below takes 2 parameters and increments each directly.

```
sub Inc
{
    ++ $_[0];
    ++ $_[1];
}
```

and here is a C function to call it.

```
static void
call_Inc(a, b)
int a;
int b;
{
    dSP;
    int count;
    SV * sva;
    SV * svb;

    ENTER;
    SAVETMPS;

    sva = sv_2mortal(newSViv(a));
    svb = sv_2mortal(newSViv(b));

    PUSHMARK(SP);
    XPUSHs(sva);
    XPUSHs(svb);
    PUTBACK;
```

```
        count = call_pv("Inc", G_DISCARD);

        if (count != 0)
            croak ("call_Inc: expected 0 values from 'Inc', got %d\n",
                   count);

        printf ("%d + 1 = %d\n", a, SvIV(sva));
        printf ("%d + 1 = %d\n", b, SvIV(svb));

        FREETMPS;
        LEAVE;
    }
```

To be able to access the two parameters that were pushed onto the stack after they return from *call_pv* it is necessary to make a note of their addresses–thus the two variables `sva` and `svb`.

The reason this is necessary is that the area of the Perl stack which held them will very likely have been overwritten by something else by the time control returns from *call_pv*.

7.5.7 Using G_EVAL

Now an example using G_EVAL. Below is a Perl subroutine which computes the difference of its 2 parameters. If this would result in a negative result, the subroutine calls *die*.

```
    sub Subtract
    {
        my ($a, $b) = @_;

        die "death can be fatal\n" if $a < $b;

        $a - $b;
    }
```

and some C to call it

```
    static void
    call_Subtract(a, b)
    int a;
    int b;
    {
        dSP;
        int count;

        ENTER;
        SAVETMPS;

        PUSHMARK(SP);
        XPUSHs(sv_2mortal(newSViv(a)));
        XPUSHs(sv_2mortal(newSViv(b)));
        PUTBACK;
        count = call_pv("Subtract", G_EVAL|G_SCALAR);

        SPAGAIN;

        /* Check the eval first */
        if (SvTRUE(ERRSV))
        {
            printf ("Uh oh - %s\n", SvPV_nolen(ERRSV));
            POPs;
        }
        else
        {
            if (count != 1)
               croak("call_Subtract: wanted 1 value from 'Subtract', got %d\n",
                        count);

            printf ("%d - %d = %d\n", a, b, POPi);
        }

        PUTBACK;
        FREETMPS;
        LEAVE;
    }
```

If *call_Subtract* is called thus

```
call_Subtract(4, 5)
```

the following will be printed

```
Uh oh - death can be fatal
```

Notes

1. We want to be able to catch the *die* so we have used the G_EVAL flag. Not specifying this flag would mean that the program would terminate immediately at the *die* statement in the subroutine *Subtract*.
2. The code

   ```
       if (SvTRUE(ERRSV))
       {
           printf ("Uh oh - %s\n", SvPV_nolen(ERRSV));
           POPs;
       }
   ```

   is the direct equivalent of this bit of Perl

   ```
       print "Uh oh - $@\n" if $@;
   ```

   `PL_errgv` is a perl global of type `GV *` that points to the symbol table entry containing the error. `ERRSV` therefore refers to the C equivalent of `$@`.
3. Note that the stack is popped using `POPs` in the block where `SvTRUE(ERRSV)` is true. This is necessary because whenever a *call_** function invoked with G_EVAL|G_SCALAR returns an error, the top of the stack holds the value *undef*.

Because we want the program to continue after detecting this error, it is essential that the stack be tidied up by removing the *undef*.

7.5.8 Using G_KEEPERR

Consider this rather facetious example, where we have used an XS version of the call_Subtract example above inside a destructor:

```
package Foo;
sub new { bless {}, $_[0] }
sub Subtract {
    my($a,$b) = @_;
    die "death can be fatal" if $a < $b;
    $a - $b;
}
sub DESTROY { call_Subtract(5, 4); }
sub foo { die "foo dies"; }

package main;
{
    my $foo = Foo->new;
    eval { $foo->foo };
}
print "Saw: $@" if $@;             # should be, but isn't
```

This example will fail to recognize that an error occurred inside the `eval {}`. Here's why: the call_Subtract code got executed while perl was cleaning up temporaries when exiting the outer braced block, and because call_Subtract is implemented with *call_pv* using the G_EVAL flag, it promptly reset `$@`. This results in the failure of the outermost test for `$@`, and thereby the failure of the error trap.

Appending the G_KEEPERR flag, so that the *call_pv* call in call_Subtract reads:

```
    count = call_pv("Subtract", G_EVAL|G_SCALAR|G_KEEPERR);
```

will preserve the error and restore reliable error handling.

7.5.9 Using call_sv

In all the previous examples I have 'hard-wired' the name of the Perl subroutine to be called from C. Most of the time though, it is more convenient to be able to specify the name of the Perl subroutine from within the Perl script.

Consider the Perl code below

```
sub fred
{
    print "Hello there\n";
}

CallSubPV("fred");
```

Here is a snippet of XSUB which defines *CallSubPV*.

```
void
CallSubPV(name)
```

```
        char *  name
        CODE:
        PUSHMARK(SP);
        call_pv(name, G_DISCARD|G_NOARGS);
```

That is fine as far as it goes. The thing is, the Perl subroutine can be specified as only a string, however, Perl allows references to subroutines and anonymous subroutines. This is where *call_sv* is useful.

The code below for *CallSubSV* is identical to *CallSubPV* except that the `name` parameter is now defined as an SV* and we use *call_sv* instead of *call_pv*.

```
    void
    CallSubSV(name)
        SV *    name
        CODE:
        PUSHMARK(SP);
        call_sv(name, G_DISCARD|G_NOARGS);
```

Because we are using an SV to call *fred* the following can all be used:

```
    CallSubSV("fred");
    CallSubSV(\&fred);
    $ref = \&fred;
    CallSubSV($ref);
    CallSubSV( sub { print "Hello there\n" } );
```

As you can see, *call_sv* gives you much greater flexibility in how you can specify the Perl subroutine.

You should note that, if it is necessary to store the SV (`name` in the example above) which corresponds to the Perl subroutine so that it can be used later in the program, it not enough just to store a copy of the pointer to the SV. Say the code above had been like this:

```
    static SV * rememberSub;

    void
    SaveSub1(name)
        SV *    name
        CODE:
        rememberSub = name;

    void
    CallSavedSub1()
        CODE:
        PUSHMARK(SP);
        call_sv(rememberSub, G_DISCARD|G_NOARGS);
```

The reason this is wrong is that, by the time you come to use the pointer `rememberSub` in `CallSavedSub1`, it may or may not still refer to the Perl subroutine that was recorded in `SaveSub1`. This is particularly true for these cases:

```
    SaveSub1(\&fred);
    CallSavedSub1();
```

```
SaveSub1( sub { print "Hello there\n" } );
CallSavedSub1();
```

By the time each of the SaveSub1 statements above has been executed, the SV*s which corresponded to the parameters will no longer exist. Expect an error message from Perl of the form

```
Can't use an undefined value as a subroutine reference at ...
```

for each of the CallSavedSub1 lines.

Similarly, with this code

```
$ref = \&fred;
SaveSub1($ref);
$ref = 47;
CallSavedSub1();
```

you can expect one of these messages (which you actually get is dependent on the version of Perl you are using)

```
Not a CODE reference at ...
Undefined subroutine &main::47 called ...
```

The variable $ref may have referred to the subroutine fred whenever the call to SaveSub1 was made but by the time CallSavedSub1 gets called it now holds the number 47. Because we saved only a pointer to the original SV in SaveSub1, any changes to $ref will be tracked by the pointer rememberSub. This means that whenever CallSavedSub1 gets called, it will attempt to execute the code which is referenced by the SV* rememberSub. In this case though, it now refers to the integer 47, so expect Perl to complain loudly.

A similar but more subtle problem is illustrated with this code:

```
$ref = \&fred;
SaveSub1($ref);
$ref = \&joe;
CallSavedSub1();
```

This time whenever CallSavedSub1 gets called it will execute the Perl subroutine joe (assuming it exists) rather than fred as was originally requested in the call to SaveSub1.

To get around these problems it is necessary to take a full copy of the SV. The code below shows SaveSub2 modified to do that.

```
static SV * keepSub = (SV*)NULL;

void
SaveSub2(name)
    SV *    name
    CODE:
    /* Take a copy of the callback */
    if (keepSub == (SV*)NULL)
        /* First time, so create a new SV */
        keepSub = newSVsv(name);
    else
        /* Been here before, so overwrite */
        SvSetSV(keepSub, name);
```

```
void
CallSavedSub2()
    CODE:
    PUSHMARK(SP);
    call_sv(keepSub, G_DISCARD|G_NOARGS);
```

To avoid creating a new SV every time `SaveSub2` is called, the function first checks to see if it has been called before. If not, then space for a new SV is allocated and the reference to the Perl subroutine `name` is copied to the variable `keepSub` in one operation using `newSVsv`. Thereafter, whenever `SaveSub2` is called, the existing SV, `keepSub`, is overwritten with the new value using `SvSetSV`.

7.5.10 Using call_argv

Here is a Perl subroutine which prints whatever parameters are passed to it.

```
sub PrintList
{
    my(@list) = @_;

    foreach (@list) { print "$_\n" }
}
```

And here is an example of *call_argv* which will call *PrintList*.

```
static char * words[] = {"alpha", "beta", "gamma", "delta", NULL};

static void
call_PrintList()
{
    dSP;

    call_argv("PrintList", G_DISCARD, words);
}
```

Note that it is not necessary to call `PUSHMARK` in this instance. This is because *call_argv* will do it for you.

7.5.11 Using call_method

Consider the following Perl code:

```
{
    package Mine;

    sub new
    {
        my($type) = shift;
        bless [@_]
    }

    sub Display
    {
        my ($self, $index) = @_;
        print "$index: $$self[$index]\n";
    }

    sub PrintID
    {
        my($class) = @_;
        print "This is Class $class version 1.0\n";
    }
}
```

It implements just a very simple class to manage an array. Apart from the constructor, `new`, it declares methods, one static and one virtual. The static method, `PrintID`, prints out simply the class name and a version number. The virtual method, `Display`, prints out a single element of the array. Here is an all-Perl example of using it.

```
$a = Mine->new('red', 'green', 'blue');
$a->Display(1);
Mine->PrintID;
```

will print

```
1: green
This is Class Mine version 1.0
```

Calling a Perl method from C is fairly straightforward. The following things are required:

- A reference to the object for a virtual method or the name of the class for a static method
- The name of the method
- Any other parameters specific to the method

Here is a simple XSUB which illustrates the mechanics of calling both the `PrintID` and `Display` methods from C.

```
void
call_Method(ref, method, index)
    SV *    ref
    char *  method
    int             index
    CODE:
    PUSHMARK(SP);
    XPUSHs(ref);
    XPUSHs(sv_2mortal(newSViv(index)));
    PUTBACK;

    call_method(method, G_DISCARD);

void
call_PrintID(class, method)
    char *  class
```

```
        char *  method
        CODE:
        PUSHMARK(SP);
        XPUSHs(sv_2mortal(newSVpv(class, 0)));
        PUTBACK;

        call_method(method, G_DISCARD);
```

So the methods PrintID and Display can be invoked like this:

```
    $a = Mine->new('red', 'green', 'blue');
    call_Method($a, 'Display', 1);
    call_PrintID('Mine', 'PrintID');
```

The only thing to note is that, in both the static and virtual methods, the method name is not passed via the stack–it is used as the first parameter to *call_method*.

7.5.12 Using GIMME_V

Here is a trivial XSUB which prints the context in which it is currently executing.

```
    void
    PrintContext()
        CODE:
        I32 gimme = GIMME_V;
        if (gimme == G_VOID)
            printf ("Context is Void\n");
        else if (gimme == G_SCALAR)
            printf ("Context is Scalar\n");
        else
            printf ("Context is Array\n");
```

And here is some Perl to test it.

```
    PrintContext;
    $a = PrintContext;
    @a = PrintContext;
```

The output from that will be

```
    Context is Void
    Context is Scalar
    Context is Array
```

7.5.13 Using Perl to Dispose of Temporaries

In the examples given to date, any temporaries created in the callback (i.e., parameters passed on the stack to the *call_** function or values returned via the stack) have been freed by one of these methods:

- Specifying the G_DISCARD flag with *call_**
- Explicitly using the ENTER/SAVETMPS–FREETMPS/LEAVE pairing

There is another method which can be used, namely letting Perl do it for you automatically whenever it regains control after the callback has terminated. This is done by simply not using the

```
ENTER;
SAVETMPS;
...
FREETMPS;
LEAVE;
```

sequence in the callback (and not, of course, specifying the G_DISCARD flag).

If you are going to use this method you have to be aware of a possible memory leak which can arise under very specific circumstances. To explain these circumstances you need to know a bit about the flow of control between Perl and the callback routine.

The examples given at the start of the document (an error handler and an event driven program) are typical of the two main sorts of flow control that you are likely to encounter with callbacks. There is a very important distinction between them, so pay attention.

In the first example, an error handler, the flow of control could be as follows. You have created an interface to an external library. Control can reach the external library like this

```
perl --> XSUB --> external library
```

Whilst control is in the library, an error condition occurs. You have previously set up a Perl callback to handle this situation, so it will get executed. Once the callback has finished, control will drop back to Perl again. Here is what the flow of control will be like in that situation

```
perl --> XSUB --> external library
                  ...
                  error occurs
                  ...
                  external library --> call_* --> perl
                                                      |
perl <-- XSUB <-- external library <-- call_* <----+
```

After processing of the error using *call_** is completed, control reverts back to Perl more or less immediately.

In the diagram, the further right you go the more deeply nested the scope is. It is only when control is back with perl on the extreme left of the diagram that you will have dropped back to the enclosing scope and any temporaries you have left hanging around will be freed.

In the second example, an event driven program, the flow of control will be more like this

```
perl --> XSUB --> event handler
                  ...
                  event handler --> call_* --> perl
                                                   |
                  event handler <-- call_* <----+
                  ...
                  event handler --> call_* --> perl
                                                   |
                  event handler <-- call_* <----+
                  ...
                  event handler --> call_* --> perl
```

```
                                                    |
                  event handler <-- call_* <----+
```

In this case the flow of control can consist of only the repeated sequence

```
event handler --> call_* --> perl
```

for practically the complete duration of the program. This means that control may *never* drop back to the surrounding scope in Perl at the extreme left.

So what is the big problem? Well, if you are expecting Perl to tidy up those temporaries for you, you might be in for a long wait. For Perl to dispose of your temporaries, control must drop back to the enclosing scope at some stage. In the event driven scenario that may never happen. This means that, as time goes on, your program will create more and more temporaries, none of which will ever be freed. As each of these temporaries consumes some memory your program will eventually consume all the available memory in your system–kapow!

So here is the bottom line–if you are sure that control will revert back to the enclosing Perl scope fairly quickly after the end of your callback, then it isn't absolutely necessary to dispose explicitly of any temporaries you may have created. Mind you, if you are at all uncertain about what to do, it doesn't do any harm to tidy up anyway.

7.5.14 Strategies for Storing Callback Context Information

Potentially one of the trickiest problems to overcome when designing a callback interface can be figuring out how to store the mapping between the C callback function and the Perl equivalent.

To help understand why this can be a real problem first consider how a callback is set up in an all C environment. Typically a C API will provide a function to register a callback. This will expect a pointer to a function as one of its parameters. Below is a call to a hypothetical function `register_fatal` which registers the C function to get called when a fatal error occurs.

```
register_fatal(cb1);
```

The single parameter `cb1` is a pointer to a function, so you must have defined `cb1` in your code, say something like this

```
static void
cb1()
{
    printf ("Fatal Error\n");
    exit(1);
}
```

Now change that to call a Perl subroutine instead

```
static SV * callback = (SV*)NULL;

static void
cb1()
{
    dSP;
```

```
        PUSHMARK(SP);

        /* Call the Perl sub to process the callback */
        call_sv(callback, G_DISCARD);
    }


    void
    register_fatal(fn)
        SV *    fn
        CODE:
        /* Remember the Perl sub */
        if (callback == (SV*)NULL)
            callback = newSVsv(fn);
        else
            SvSetSV(callback, fn);

        /* register the callback with the external library */
        register_fatal(cb1);
```

where the Perl equivalent of `register_fatal` and the callback it registers, `pcb1`, might look like this

```
    # Register the sub pcb1
    register_fatal(\&pcb1);

    sub pcb1
    {
        die "I'm dying...\n";
    }
```

The mapping between the C callback and the Perl equivalent is stored in the global variable `callback`.

This will be adequate if you ever need to have only one callback registered at any time. An example could be an error handler like the code sketched out above. Remember though, repeated calls to `register_fatal` will replace the previously registered callback function with the new one.

Say for example you want to interface to a library which allows asynchronous file i/o. In this case you may be able to register a callback whenever a read operation has completed. To be of any use we want to be able to call separate Perl subroutines for each file that is opened. As it stands, the error handler example above would not be adequate as it allows only a single callback to be defined at any time. What we require is a means of storing the mapping between the opened file and the Perl subroutine we want to be called for that file.

Say the i/o library has a function `asynch_read` which associates a C function `ProcessRead` with a file handle `fh`–this assumes that it has also provided some routine to open the file and so obtain the file handle.

```
    asynch_read(fh, ProcessRead)
```

This may expect the C *ProcessRead* function of this form

```
void
ProcessRead(fh, buffer)
int fh;
char *      buffer;
{
     ...
}
```

To provide a Perl interface to this library we need to be able to map between the fh parameter and the Perl subroutine we want called. A hash is a convenient mechanism for storing this mapping. The code below shows a possible implementation

```
static HV * Mapping = (HV*)NULL;

void
asynch_read(fh, callback)
    int     fh
    SV *    callback
    CODE:
    /* If the hash doesn't already exist, create it */
    if (Mapping == (HV*)NULL)
        Mapping = newHV();

    /* Save the fh -> callback mapping */
    hv_store(Mapping, (char*)&fh, sizeof(fh), newSVsv(callback), 0);

    /* Register with the C Library */
    asynch_read(fh, asynch_read_if);
```

and asynch_read_if could look like this

```
static void
asynch_read_if(fh, buffer)
int fh;
char *      buffer;
{
    dSP;
    SV ** sv;

    /* Get the callback associated with fh */
    sv =  hv_fetch(Mapping, (char*)&fh , sizeof(fh), FALSE);
    if (sv == (SV**)NULL)
        croak("Internal error...\n");

    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSViv(fh)));
    XPUSHs(sv_2mortal(newSVpv(buffer, 0)));
    PUTBACK;

    /* Call the Perl sub */
```

```
        call_sv(*sv, G_DISCARD);
    }
```

For completeness, here is asynch_close. This shows how to remove the entry from the hash Mapping.

```
    void
    asynch_close(fh)
        int     fh
        CODE:
        /* Remove the entry from the hash */
        (void) hv_delete(Mapping, (char*)&fh, sizeof(fh), G_DISCARD);

        /* Now call the real asynch_close */
        asynch_close(fh);
```

So the Perl interface would look like this

```
    sub callback1
    {
        my($handle, $buffer) = @_;
    }

    # Register the Perl callback
    asynch_read($fh, \&callback1);

    asynch_close($fh);
```

The mapping between the C callback and Perl is stored in the global hash Mapping this time. Using a hash has the distinct advantage that it allows an unlimited number of callbacks to be registered.

What if the interface provided by the C callback doesn't contain a parameter which allows the file handle to Perl subroutine mapping? Say in the asynchronous i/o package, the callback function gets passed only the buffer parameter like this

```
    void
    ProcessRead(buffer)
    char *      buffer;
    {
        ...
    }
```

Without the file handle there is no straightforward way to map from the C callback to the Perl subroutine.

In this case a possible way around this problem is to predefine a series of C functions to act as the interface to Perl, thus

```
    #define MAX_CB              3
    #define NULL_HANDLE -1
    typedef void (*FnMap)();

    struct MapStruct {
        FnMap    Function;
        SV *     PerlSub;
        int      Handle;
      };

    static void  fn1();
    static void  fn2();
    static void  fn3();

    static struct MapStruct Map [MAX_CB] =
        {
            { fn1, NULL, NULL_HANDLE },
            { fn2, NULL, NULL_HANDLE },
            { fn3, NULL, NULL_HANDLE }
        };

    static void
    Pcb(index, buffer)
    int index;
    char * buffer;
    {
        dSP;

        PUSHMARK(SP);
        XPUSHs(sv_2mortal(newSVpv(buffer, 0)));
        PUTBACK;

        /* Call the Perl sub */
        call_sv(Map[index].PerlSub, G_DISCARD);
    }

    static void
    fn1(buffer)
    char * buffer;
    {
        Pcb(0, buffer);
    }

    static void
    fn2(buffer)
    char * buffer;
    {
        Pcb(1, buffer);
    }

    static void
    fn3(buffer)
    char * buffer;
```

```
{
    Pcb(2, buffer);
}

void
array_asynch_read(fh, callback)
    int             fh
    SV *    callback
    CODE:
    int index;
    int null_index = MAX_CB;

    /* Find the same handle or an empty entry */
    for (index = 0; index < MAX_CB; ++index)
    {
        if (Map[index].Handle == fh)
            break;

        if (Map[index].Handle == NULL_HANDLE)
            null_index = index;
    }

    if (index == MAX_CB && null_index == MAX_CB)
        croak ("Too many callback functions registered\n");

    if (index == MAX_CB)
        index = null_index;

    /* Save the file handle */
    Map[index].Handle = fh;

    /* Remember the Perl sub */
    if (Map[index].PerlSub == (SV*)NULL)
        Map[index].PerlSub = newSVsv(callback);
    else
        SvSetSV(Map[index].PerlSub, callback);

    asynch_read(fh, Map[index].Function);

void
array_asynch_close(fh)
    int     fh
    CODE:
    int index;

    /* Find the file handle */
    for (index = 0; index < MAX_CB; ++ index)
```

```
        if (Map[index].Handle == fh)
            break;

    if (index == MAX_CB)
        croak ("could not close fh %d\n", fh);

    Map[index].Handle = NULL_HANDLE;
    SvREFCNT_dec(Map[index].PerlSub);
    Map[index].PerlSub = (SV*)NULL;

    asynch_close(fh);
```

In this case the functions `fn1`, `fn2`, and `fn3` are used to remember the Perl subroutine to be called. Each of the functions holds a separate hard-wired index which is used in the function `Pcb` to access the `Map` array and actually call the Perl subroutine.

There are some obvious disadvantages with this technique.

Firstly, the code is considerably more complex than with the previous example.

Secondly, there is a hard-wired limit (in this case 3) to the number of callbacks that can exist simultaneously. The only way to increase the limit is by modifying the code to add more functions and then recompiling. None the less, as long as the number of functions is chosen with some care, it is still a workable solution and in some cases is the only one available.

To summarize, here are a number of possible methods for you to consider for storing the mapping between C and the Perl callback

1. Ignore the problem - Allow only 1 callback

   For a lot of situations, like interfacing to an error handler, this may be a perfectly adequate solution.

2. Create a sequence of callbacks - hard wired limit

   If it is impossible to tell from the parameters passed back from the C callback what the context is, then you may need to create a sequence of C callback interface functions, and store pointers to each in an array.

3. Use a parameter to map to the Perl callback

   A hash is an ideal mechanism to store the mapping between C and Perl.

7.5.15 Alternate Stack Manipulation

Although I have made use of only the `POP*` macros to access values returned from Perl subroutines, it is also possible to bypass these macros and read the stack using the `ST` macro (See `perlxs` for a full description of the `ST` macro).

Most of the time the `POP*` macros should be adequate; the main problem with them is that they force you to process the returned values in sequence. This may not be the most suitable way to process the values in some cases. What we want is to be able to access the stack in a random order. The `ST` macro as used when coding an XSUB is ideal for this purpose.

The code below is the example given in the section *Returning a List of Values* recoded to use `ST` instead of `POP*`.

```
static void
call_AddSubtract2(a, b)
int a;
int b;
{
    dSP;
    I32 ax;
    int count;

    ENTER;
    SAVETMPS;

    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSViv(a)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK;

    count = call_pv("AddSubtract", G_ARRAY);

    SPAGAIN;
    SP -= count;
    ax = (SP - PL_stack_base) + 1;

    if (count != 2)
        croak("Big trouble\n");

    printf ("%d + %d = %d\n", a, b, SvIV(ST(0)));
    printf ("%d - %d = %d\n", a, b, SvIV(ST(1)));

    PUTBACK;
    FREETMPS;
    LEAVE;
}
```

Notes

1. Notice that it was necessary to define the variable `ax`. This is because the `ST` macro expects it to exist. If we were in an XSUB it would not be necessary to define `ax` as it is already defined for us.

2. The code

   ```
   SPAGAIN;
   SP -= count;
   ax = (SP - PL_stack_base) + 1;
   ```

   sets the stack up so that we can use the `ST` macro.

3. Unlike the original coding of this example, the returned values are not accessed in reverse order. So `ST(0)` refers to the first value returned by the Perl subroutine and `ST(count-1)` refers to the last.

7.5.16 Creating and Calling an Anonymous Subroutine in C

As we've already shown, `call_sv` can be used to invoke an anonymous subroutine. However, our example showed a Perl script invoking an XSUB to perform this operation. Let's see how it can be done inside our C code:

```
 ...

 SV *cvrv = eval_pv("sub { print 'You will not find me cluttering any namespace!' }", TRUE)

 ...

 call_sv(cvrv, G_VOID|G_NOARGS);
```

`eval_pv` is used to compile the anonymous subroutine, which will be the return value as well (read more about `eval_pv` in Section "eval_pv" in `perlapi`). Once this code reference is in hand, it can be mixed in with all the previous examples we've shown.

7.6 LIGHTWEIGHT CALLBACKS

Sometimes you need to invoke the same subroutine repeatedly. This usually happens with a function that acts on a list of values, such as Perl's built-in sort(). You can pass a comparison function to sort(), which will then be invoked for every pair of values that needs to be compared. The first() and reduce() functions from `List-Util` follow a similar pattern.

In this case it is possible to speed up the routine (often quite substantially) by using the lightweight callback API. The idea is that the calling context only needs to be created and destroyed once, and the sub can be called arbitrarily many times in between.

It is usual to pass parameters using global variables (typically $_ for one parameter, or $a and $b for two parameters) rather than via @_. (It is possible to use the @_ mechanism if you know what you're doing, though there is as yet no supported API for it. It's also inherently slower.)

The pattern of macro calls is like this:

```
    dMULTICALL;                 /* Declare local variables */
    I32 gimme = G_SCALAR;       /* context of the call: G_SCALAR,
                                 * G_ARRAY, or G_VOID */

    PUSH_MULTICALL(cv);         /* Set up the context for calling cv,
                                   and set local vars appropriately */

    /* loop */ {
        /* set the value(s) af your parameter variables */
        MULTICALL;              /* Make the actual call */
    } /* end of loop */

    POP_MULTICALL;              /* Tear down the calling context */
```

For some concrete examples, see the implementation of the first() and reduce() functions of List::Util 1.18. There you will also find a header file that emulates the multicall API on older versions of perl.

7.7 SEE ALSO

`perlxs`, Section 28.1 [perlguts NAME], page 512, Section 20.1 [perlembed NAME], page 301

7.8 AUTHOR

Paul Marquess

Special thanks to the following people who assisted in the creation of the document.

Jeff Okamoto, Tim Bunce, Nick Gianniotis, Steve Kelem, Gurusamy Sarathy and Larry Wall.

7.9 DATE

Version 1.3, 14th Apr 1997

8 perlcheat

8.1 NAME

perlcheat - Perl 5 Cheat Sheet

8.2 DESCRIPTION

This ’cheat sheet’ is a handy reference, meant for beginning Perl programmers. Not everything is mentioned, but 195 features may already be overwhelming.

8.2.1 The sheet

```
  CONTEXTS  SIGILS  ref        ARRAYS        HASHES
  void      $scalar SCALAR     @array        %hash
  scalar    @array  ARRAY      @array[0, 2]  @hash{’a’, ’b’}
  list      %hash   HASH       $array[0]     $hash{’a’}
            &sub    CODE
            *glob   GLOB       SCALAR VALUES
                    FORMAT     number, string, ref, glob, undef
  REFERENCES
  \      reference       $$foo[1]       aka $foo->[1]
  $@%&*  dereference     $$foo{bar}     aka $foo->{bar}
  []     anon. arrayref  ${$$foo[1]}[2] aka $foo->[1]->[2]
  {}     anon. hashref   ${$$foo[1]}[2] aka $foo->[1][2]
  \()    list of refs
                         SYNTAX
  OPERATOR PRECEDENCE    foreach (LIST) { }     for (a;b;c) { }
  ->                     while   (e) { }        until (e)   { }
  ++ --                  if      (e) { } elsif (e) { } else { }
  **                     unless  (e) { } elsif (e) { } else { }
  ! ~ \ u+ u-            given   (e) { when (e) {} default {} }
  =~ !~
  * / % x                 NUMBERS vs STRINGS  FALSE vs TRUE
  + - .                   =          =        undef, "", 0, "0"
  << >>                   +          .        anything else
  named uops              == !=      eq ne
  < > <= >= lt gt le ge   < > <= >=  lt gt le ge
  == != <=> eq ne cmp ~~  <=>        cmp
  &
  | ^             REGEX MODIFIERS       REGEX METACHARS
  &&              /i case insensitive   ^      string begin
  || //           /m line based ^$      $      str end (bfr \n)
  .. ...          /s . includes \n      +      one or more
  ?:              /x ignore wh.space    *      zero or more
  = += last goto  /p preserve           ?      zero or one
  , =>            /a ASCII    /aa safe  {3,7}  repeat in range
  list ops        /l locale   /d  dual  |      alternation
```

```
not             /u Unicode            []     character class
and             /e evaluate /ee rpts  \b     boundary
or xor          /g global             \z     string end
                /o compile pat once   ()     capture
DEBUG                                 (?:p)  no capture
-MO=Deparse     REGEX CHARCLASSES     (?#t)  comment
-MO=Terse       .   [^\n]             (?=p)  ZW pos ahead
-D##            \s  whitespace        (?!p)  ZW neg ahead
-d:Trace        \w  word chars        (?<=p) ZW pos behind \K
                \d  digits            (?<!p) ZW neg behind
CONFIGURATION   \pP named property    (?>p)  no backtrack
perl -V:ivsize  \h  horiz.wh.space    (?|p|p)branch reset
                \R  linebreak         (?<n>p)named capture
                \S \W \D \H negate    \g{n}  ref to named cap
                                      \K     keep left part
FUNCTION RETURN LISTS
stat      localtime    caller         SPECIAL VARIABLES
 0 dev    0 second      0 package     $_    default variable
 1 ino    1 minute      1 filename    $0    program name
 2 mode   2 hour        2 line        $/    input separator
 3 nlink  3 day         3 subroutine  $\    output separator
 4 uid    4 month-1     4 hasargs     $|    autoflush
 5 gid    5 year-1900   5 wantarray   $!    sys/libcall error
 6 rdev   6 weekday     6 evaltext    $@    eval error
 7 size   7 yearday     7 is_require  $$    process ID
 8 atime  8 is_dst      8 hints       $.    line number
 9 mtime                9 bitmask     @ARGV command line args
10 ctime               10 hinthash    @INC  include paths
11 blksz               3..10 only     @_    subroutine args
12 blcks               with EXPR      %ENV  environment
```

8.3 ACKNOWLEDGEMENTS

The first version of this document appeared on Perl Monks, where several people had useful suggestions. Thank you, Perl Monks.

A special thanks to Damian Conway, who didn't only suggest important changes, but also took the time to count the number of listed features and make a Perl 6 version to show that Perl will stay Perl.

8.4 AUTHOR

Juerd Waalboer <#####@juerd.nl>, with the help of many Perl Monks.

8.5 SEE ALSO

- `http://perlmonks.org/?node_id=216602` - the original PM post
- `http://perlmonks.org/?node_id=238031` - Damian Conway's Perl 6 version

- `http://juerd.nl/site.plp/perlcheat` - home of the Perl Cheat Sheet

9 perlclib

9.1 NAME

perlclib - Internal replacements for standard C library functions

9.2 DESCRIPTION

One thing Perl porters should note is that `perl` doesn't tend to use that much of the C standard library internally; you'll see very little use of, for example, the `ctype.h` functions in there. This is because Perl tends to reimplement or abstract standard library functions, so that we know exactly how they're going to operate.

This is a reference card for people who are familiar with the C library and who want to do things the Perl way; to tell them which functions they ought to use instead of the more normal C functions.

9.2.1 Conventions

In the following tables:

`t`

is a type.

`p`

is a pointer.

`n`

is a number.

`s`

is a string.

`sv`, `av`, `hv`, etc. represent variables of their respective types.

9.2.2 File Operations

Instead of the `stdio.h` functions, you should use the Perl abstraction layer. Instead of `FILE*` types, you need to be handling `PerlIO*` types. Don't forget that with the new PerlIO layered I/O abstraction `FILE*` types may not even be available. See also the `perlapio` documentation for more information about the following functions:

Instead Of:	Use:
stdin	PerlIO_stdin()
stdout	PerlIO_stdout()
stderr	PerlIO_stderr()
fopen(fn, mode)	PerlIO_open(fn, mode)
freopen(fn, mode, stream)	PerlIO_reopen(fn, mode, perlio) (Deprecated)
fflush(stream)	PerlIO_flush(perlio)
fclose(stream)	PerlIO_close(perlio)

9.2.3 File Input and Output

```
Instead Of:                 Use:

fprintf(stream, fmt, ...)   PerlIO_printf(perlio, fmt, ...)

[f]getc(stream)             PerlIO_getc(perlio)
[f]putc(stream, n)          PerlIO_putc(perlio, n)
ungetc(n, stream)           PerlIO_ungetc(perlio, n)
```

Note that the PerlIO equivalents of `fread` and `fwrite` are slightly different from their C library counterparts:

```
fread(p, size, n, stream)   PerlIO_read(perlio, buf, numbytes)
fwrite(p, size, n, stream)  PerlIO_write(perlio, buf, numbytes)

fputs(s, stream)            PerlIO_puts(perlio, s)
```

There is no equivalent to `fgets`; one should use `sv_gets` instead:

```
fgets(s, n, stream)         sv_gets(sv, perlio, append)
```

9.2.4 File Positioning

```
Instead Of:                 Use:

feof(stream)                PerlIO_eof(perlio)
fseek(stream, n, whence)    PerlIO_seek(perlio, n, whence)
rewind(stream)              PerlIO_rewind(perlio)

fgetpos(stream, p)          PerlIO_getpos(perlio, sv)
fsetpos(stream, p)          PerlIO_setpos(perlio, sv)

ferror(stream)              PerlIO_error(perlio)
clearerr(stream)            PerlIO_clearerr(perlio)
```

9.2.5 Memory Management and String Handling

```
Instead Of:                    Use:

t* p = malloc(n)               Newx(p, n, t)
t* p = calloc(n, s)            Newxz(p, n, t)
p = realloc(p, n)              Renew(p, n, t)
memcpy(dst, src, n)            Copy(src, dst, n, t)
memmove(dst, src, n)           Move(src, dst, n, t)
memcpy(dst, src, sizeof(t))    StructCopy(src, dst, t)
memset(dst, 0, n * sizeof(t))  Zero(dst, n, t)
memzero(dst, 0)                Zero(dst, n, char)
free(p)                        Safefree(p)

strdup(p)                      savepv(p)
strndup(p, n)                  savepvn(p, n) (Hey, strndup doesn't
                                              exist!)

strstr(big, little)          instr(big, little)
strcmp(s1, s2)               strLE(s1, s2) / strEQ(s1, s2)
                                             / strGT(s1,s2)
strncmp(s1, s2, n)           strnNE(s1, s2, n) / strnEQ(s1, s2, n)

memcmp(p1, p2, n)            memNE(p1, p2, n)
!memcmp(p1, p2, n)           memEQ(p1, p2, n)
```

Notice the different order of arguments to `Copy` and `Move` than used in `memcpy` and `memmove`.

Most of the time, though, you'll want to be dealing with SVs internally instead of raw `char *` strings:

```
 strlen(s)                 sv_len(sv)
 strcpy(dt, src)           sv_setpv(sv, s)
 strncpy(dt, src, n)       sv_setpvn(sv, s, n)
 strcat(dt, src)           sv_catpv(sv, s)
 strncat(dt, src)          sv_catpvn(sv, s)
 sprintf(s, fmt, ...)      sv_setpvf(sv, fmt, ...)
```

Note also the existence of `sv_catpvf` and `sv_vcatpvfn`, combining concatenation with formatting.

Sometimes instead of zeroing the allocated heap by using Newxz() you should consider "poisoning" the data. This means writing a bit pattern into it that should be illegal as pointers (and floating point numbers), and also hopefully surprising enough as integers, so that any code attempting to use the data without forethought will break sooner rather than later. Poisoning can be done using the Poison() macros, which have similar arguments to Zero():

```
 PoisonWith(dst, n, t, b)    scribble memory with byte b
 PoisonNew(dst, n, t)        equal to PoisonWith(dst, n, t, 0xAB)
 PoisonFree(dst, n, t)       equal to PoisonWith(dst, n, t, 0xEF)
 Poison(dst, n, t)           equal to PoisonFree(dst, n, t)
```

9.2.6 Character Class Tests

There are several types of character class tests that Perl implements. The only ones described here are those that directly correspond to C library functions that operate on 8-bit characters, but there are equivalents that operate on wide characters, and UTF-8 encoded strings. All are more fully described in Section "Character classification" in `perlapi` and Section "Character case changing" in `perlapi`.

The C library routines listed in the table below return values based on the current locale. Use the entries in the final column for that functionality. The other two columns always assume a POSIX (or C) locale. The entries in the ASCII column are only meaningful for ASCII inputs, returning FALSE for anything else. Use these only when you **know** that is what you want. The entries in the Latin1 column assume that the non-ASCII 8-bit characters are as Unicode defines, them, the same as ISO-8859-1, often called Latin 1.

```
 Instead Of:  Use for ASCII:   Use for Latin1:      Use for locale:
```

isalnum(c)	isALPHANUMERIC(c)	isALPHANUMERIC_L1(c)	isALPHANUMERIC_LC(c)
isalpha(c)	isALPHA(c)	isALPHA_L1(c)	isALPHA_LC(u)
isascii(c)	isASCII(c)		isASCII_LC(c)
isblank(c)	isBLANK(c)	isBLANK_L1(c)	isBLANK_LC(c)
iscntrl(c)	isCNTRL(c)	isCNTRL_L1(c)	isCNTRL_LC(c)
isdigit(c)	isDIGIT(c)	isDIGIT_L1(c)	isDIGIT_LC(c)
isgraph(c)	isGRAPH(c)	isGRAPH_L1(c)	isGRAPH_LC(c)
islower(c)	isLOWER(c)	isLOWER_L1(c)	isLOWER_LC(c)
isprint(c)	isPRINT(c)	isPRINT_L1(c)	isPRINT_LC(c)
ispunct(c)	isPUNCT(c)	isPUNCT_L1(c)	isPUNCT_LC(c)
isspace(c)	isSPACE(c)	isSPACE_L1(c)	isSPACE_LC(c)
isupper(c)	isUPPER(c)	isUPPER_L1(c)	isUPPER_LC(c)
isxdigit(c)	isXDIGIT(c)	isXDIGIT_L1(c)	isXDIGIT_LC(c)
tolower(c)	toLOWER(c)	toLOWER_L1(c)	toLOWER_LC(c)
toupper(c)	toUPPER(c)		toUPPER_LC(c)

To emphasize that you are operating only on ASCII characters, you can append `_A` to each of the macros in the ASCII column: `isALPHA_A`, `isDIGIT_A`, and so on.

(There is no entry in the Latin1 column for `isascii` even though there is an `isASCII_L1`, which is identical to `isASCII`; the latter name is clearer. There is no entry in the Latin1 column for `toupper` because the result can be non-Latin1. You have to use `toUPPER_uni`, as described in Section "Character case changing" in `perlapi`.)

9.2.7 `stdlib.h` functions

Instead Of:	Use:
atof(s)	Atof(s)
atoi(s)	grok_atoUV(s, &uv, &e)
atol(s)	grok_atoUV(s, &uv, &e)
strtod(s, &p)	Nothing. Just don't use it.
strtol(s, &p, n)	grok_atoUV(s, &uv, &e)
strtoul(s, &p, n)	grok_atoUV(s, &uv, &e)

Typical use is to do range checks on `uv` before casting:

```
int i; UV uv; char* end_ptr;
if (grok_atoUV(input, &uv, &end_ptr)
    && uv <= INT_MAX)
  i = (int)uv;
  ... /* continue parsing from end_ptr */
} else {
  ... /* parse error: not a decimal integer in range 0 .. MAX_IV */
}
```

Notice also the `grok_bin`, `grok_hex`, and `grok_oct` functions in `numeric.c` for converting strings representing numbers in the respective bases into `NV`s. Note that grok_atoUV() doesn't handle negative inputs, or leading whitespace (being purposefully strict).

Note that strtol() and strtoul() may be disguised as Strtol(), Strtoul(), Atol(), Atoul(). Avoid those, too.

In theory `Strtol` and `Strtoul` may not be defined if the machine perl is built on doesn't actually have strtol and strtoul. But as those 2 functions are part of the 1989 ANSI C spec we suspect you'll find them everywhere by now.

```
int rand()                  double Drand01()
srand(n)                    { seedDrand01((Rand_seed_t)n);
                              PL_srand_called = TRUE; }

exit(n)                     my_exit(n)
system(s)                   Don't. Look at pp_system or use my_popen.

getenv(s)                   PerlEnv_getenv(s)
setenv(s, val)              my_setenv(s, val)
```

9.2.8 Miscellaneous functions

You should not even **want** to use `setjmp.h` functions, but if you think you do, use the `JMPENV` stack in `scope.h` instead.

For `signal/sigaction`, use `rsignal(signo, handler)`.

9.3 SEE ALSO

`perlapi`, Section 2.1 [perlapio NAME], page 9, Section 28.1 [perlguts NAME], page 512

10 perlcommunity

10.1 NAME

perlcommunity - a brief overview of the Perl community

10.2 DESCRIPTION

This document aims to provide an overview of the vast perl community, which is far too large and diverse to provide a detailed listing. If any specific niche has been forgotten, it is not meant as an insult but an omission for the sake of brevity.

The Perl community is as diverse as Perl, and there is a large amount of evidence that the Perl users apply TMTOWTDI to all endeavors, not just programming. From websites, to IRC, to mailing lists, there is more than one way to get involved in the community.

10.2.1 Where to Find the Community

There is a central directory for the Perl community: `http://perl.org` maintained by the Perl Foundation (`http://www.perlfoundation.org/`), which tracks and provides services for a variety of other community sites.

10.2.2 Mailing Lists and Newsgroups

Perl runs on e-mail; there is no doubt about it. The Camel book was originally written mostly over e-mail and today Perl's development is co-ordinated through mailing lists. The largest repository of Perl mailing lists is located at `http://lists.perl.org`.

Most Perl-related projects set up mailing lists for both users and contributors. If you don't see a certain project listed at `http://lists.perl.org`, check the particular website for that project. Most mailing lists are archived at `http://nntp.perl.org/`.

There are also plenty of Perl related newsgroups located under `comp.lang.perl.*`.

10.2.3 IRC

The Perl community has a rather large IRC presence. For starters, it has its own IRC network, `irc://irc.perl.org`. General (not help-oriented) chat can be found at `irc://irc.perl.org/#perl`. Many other more specific chats are also hosted on the network. Information about irc.perl.org is located on the network's website: `http://www.irc.perl.org`. For a more help-oriented #perl, check out `irc://irc.freenode.net/#perl`. Perl 6 development also has a presence in `irc://irc.freenode.net/#perl6`. Most Perl-related channels will be kind enough to point you in the right direction if you ask nicely.

Any large IRC network (Dalnet, EFnet) is also likely to have a #perl channel, with varying activity levels.

10.2.4 Websites

Perl websites come in a variety of forms, but they fit into two large categories: forums and news websites. There are many Perl-related websites, so only a few of the community's largest are mentioned here.

10.2.4.1 News sites

`http://perl.com/`
: Run by O'Reilly Media (the publisher of Section 4.1 [the Camel Book], page 21, among other Perl-related literature), perl.com provides current Perl news, articles, and resources for Perl developers as well as a directory of other useful websites.

`http://blogs.perl.org/`
: Many members of the community have a Perl-related blog on this site. If you'd like to join them, you can sign up for free.

`http://use.perl.org/`
: use Perl; used to provide a slashdot-style news/blog website covering all things Perl, from minutes of the meetings of the Perl 6 Design team to conference announcements with (ir)relevant discussion. It no longer accepts updates, but you can still use the site to read old entries and comments.

10.2.4.2 Forums

`http://www.perlmonks.org/`
: PerlMonks is one of the largest Perl forums, and describes itself as "A place for individuals to polish, improve, and showcase their Perl skills." and "A community which allows everyone to grow and learn from each other."

`http://stackoverflow.com/`
: Stack Overflow is a free question-and-answer site for programmers. It's not focussed solely on Perl, but it does have an active group of users who do their best to help people with their Perl programming questions.

10.2.5 User Groups

Many cities around the world have local Perl Mongers chapters. A Perl Mongers chapter is a local user group which typically holds regular in-person meetings, both social and technical; helps organize local conferences, workshops, and hackathons; and provides a mailing list or other continual contact method for its members to keep in touch.

To find your local Perl Mongers (or PM as they're commonly abbreviated) group check the international Perl Mongers directory at `http://www.pm.org/`.

10.2.6 Workshops

Perl workshops are, as the name might suggest, workshops where Perl is taught in a variety of ways. At the workshops, subjects range from a beginner's introduction (such as the Pittsburgh Perl Workshop's "Zero To Perl") to much more advanced subjects.

There are several great resources for locating workshops: the Section 10.2.4 [websites], page 67 mentioned above, the Section 10.2.9 [calendar], page 69 mentioned below, and the YAPC Europe website, `http://www.yapceurope.org/`, which is probably the best resource for European Perl events.

10.2.7 Hackathons

Hackathons are a very different kind of gathering where Perl hackers gather to do just that, hack nonstop for an extended (several day) period on a specific project or projects. Information about hackathons can be located in the same place as information about Section 10.2.6 [workshops], page 68 as well as in `irc://irc.perl.org/#perl`.

If you have never been to a hackathon, here are a few basic things you need to know before attending: have a working laptop and know how to use it; check out the involved projects beforehand; have the necessary version control client; and bring backup equipment (an extra LAN cable, additional power strips, etc.) because someone will forget.

10.2.8 Conventions

Perl has two major annual conventions: The Perl Conference (now part of OSCON), put on by O'Reilly, and Yet Another Perl Conference or YAPC (pronounced yap-see), which is localized into several regional YAPCs (North America, Europe, Asia) in a stunning grassroots display by the Perl community. For more information about either conference, check out their respective web pages: OSCON `http://conferences.oreillynet.com/`; YAPC `http://www.yapc.org`.

A relatively new conference franchise with a large Perl portion is the Open Source Developers Conference or OSDC. First held in Australia it has recently also spread to Israel and France. More information can be found at: `http://www.osdc.com.au/` for Australia, `http://www.osdc.org.il` for Israel, and `http://www.osdc.fr/` for France.

10.2.9 Calendar of Perl Events

The Perl Review, `http://www.theperlreview.com` maintains a website and Google calendar (`http://www.theperlreview.com/community_calendar`) for tracking workshops, hackathons, Perl Mongers meetings, and other events. Views of this calendar are at `http://www.perl.org/events.html` and `http://www.yapc.org`.

Not every event or Perl Mongers group is on that calendar, so don't lose heart if you don't see yours posted. To have your event or group listed, contact brian d foy (brian@theperlreview.com).

10.3 AUTHOR

Edgar "Trizor" Bering <trizor@gmail.com>

11 perldata

11.1 NAME

perldata - Perl data types

11.2 DESCRIPTION

11.2.1 Variable names

Perl has three built-in data types: scalars, arrays of scalars, and associative arrays of scalars, known as "hashes". A scalar is a single string (of any size, limited only by the available memory), number, or a reference to something (which will be discussed in Section 62.1 [perlref NAME], page 1077). Normal arrays are ordered lists of scalars indexed by number, starting with 0. Hashes are unordered collections of scalar values indexed by their associated string key.

Values are usually referred to by name, or through a named reference. The first character of the name tells you to what sort of data structure it refers. The rest of the name tells you the particular value to which it refers. Usually this name is a single *identifier*, that is, a string beginning with a letter or underscore, and containing letters, underscores, and digits. In some cases, it may be a chain of identifiers, separated by `::` (or by the slightly archaic `'`); all but the last are interpreted as names of packages, to locate the namespace in which to look up the final identifier (see Section 40.2.2 [perlmod Packages], page 732 for details). For a more in-depth discussion on identifiers, see Section 11.2.2 [Identifier parsing], page 71. It's possible to substitute for a simple identifier, an expression that produces a reference to the value at runtime. This is described in more detail below and in Section 62.1 [perlref NAME], page 1077.

Perl also has its own built-in variables whose names don't follow these rules. They have strange names so they don't accidentally collide with one of your normal variables. Strings that match parenthesized parts of a regular expression are saved under names containing only digits after the `$` (see Section 48.1 [perlop NAME], page 798 and Section 58.1 [perlre NAME], page 989). In addition, several special variables that provide windows into the inner working of Perl have names containing punctuation characters and control characters. These are documented in Section 86.1 [perlvar NAME], page 1375.

Scalar values are always named with '$', even when referring to a scalar that is part of an array or a hash. The '$' symbol works semantically like the English word "the" in that it indicates a single value is expected.

```
    $days               # the simple scalar value "days"
    $days[28]           # the 29th element of array @days
    $days{'Feb'}        # the 'Feb' value from hash %days
    $#days              # the last index of array @days
```

Entire arrays (and slices of arrays and hashes) are denoted by '@', which works much as the word "these" or "those" does in English, in that it indicates multiple values are expected.

```
    @days               # ($days[0], $days[1],... $days[n])
```

```
@days[3,4,5]        # same as ($days[3],$days[4],$days[5])
@days{'a','c'}      # same as ($days{'a'},$days{'c'})
```

Entire hashes are denoted by '%':

```
%days               # (key1, val1, key2, val2 ...)
```

In addition, subroutines are named with an initial '&', though this is optional when unambiguous, just as the word "do" is often redundant in English. Symbol table entries can be named with an initial '*', but you don't really care about that yet (if ever :-).

Every variable type has its own namespace, as do several non-variable identifiers. This means that you can, without fear of conflict, use the same name for a scalar variable, an array, or a hash–or, for that matter, for a filehandle, a directory handle, a subroutine name, a format name, or a label. This means that $foo and @foo are two different variables. It also means that `$foo[1]` is a part of @foo, not a part of $foo. This may seem a bit weird, but that's okay, because it is weird.

Because variable references always start with '$', '@', or '%', the "reserved" words aren't in fact reserved with respect to variable names. They *are* reserved with respect to labels and filehandles, however, which don't have an initial special character. You can't have a filehandle named "log", for instance. Hint: you could say `open(LOG,'logfile')` rather than `open(log,'logfile')`. Using uppercase filehandles also improves readability and protects you from conflict with future reserved words. Case *is* significant–"FOO", "Foo", and "foo" are all different names. Names that start with a letter or underscore may also contain digits and underscores.

It is possible to replace such an alphanumeric name with an expression that returns a reference to the appropriate type. For a description of this, see Section 62.1 [perlref NAME], page 1077.

Names that start with a digit may contain only more digits. Names that do not start with a letter, underscore, digit or a caret (i.e. a control character) are limited to one character, e.g., `$%` or `$$`. (Most of these one character names have a predefined significance to Perl. For instance, `$$` is the current process id.)

11.2.2 Identifier parsing

Up until Perl 5.18, the actual rules of what a valid identifier was were a bit fuzzy. However, in general, anything defined here should work on previous versions of Perl, while the opposite – edge cases that work in previous versions, but aren't defined here – probably won't work on newer versions. As an important side note, please note that the following only applies to bareword identifiers as found in Perl source code, not identifiers introduced through symbolic references, which have much fewer restrictions. If working under the effect of the `use utf8;` pragma, the following rules apply:

```
/ (?[ ( \p{Word} & \p{XID_Start} ) + [_] ])
  (?[ ( \p{Word} & \p{XID_Continue} ) ]) *    /x
```

That is, a "start" character followed by any number of "continue" characters. Perl requires every character in an identifier to also match `\w` (this prevents some problematic cases); and Perl additionally accepts identfier names beginning with an underscore.

If not under `use utf8`, the source is treated as ASCII `+` 128 extra controls, and identifiers should match

```
/ (?aa) (?!\d) \w+ /x
```

That is, any word character in the ASCII range, as long as the first character is not a digit.

There are two package separators in Perl: A double colon (`::`) and a single quote (`'`). Normal identifiers can start or end with a double colon, and can contain several parts delimited by double colons. Single quotes have similar rules, but with the exception that they are not legal at the end of an identifier: That is, `$'foo` and `$foo'bar` are legal, but `$foo'bar'` is not.

Additionally, if the identifier is preceded by a sigil – that is, if the identifier is part of a variable name – it may optionally be enclosed in braces.

While you can mix double colons with singles quotes, the quotes must come after the colons: `$::::'foo` and `$foo::'bar` are legal, but `$::'::foo` and `$foo'::bar` are not.

Put together, a grammar to match a basic identifier becomes

```
 /
  (?(DEFINE)
      (?<variable>
          (?&sigil)
          (?:
                  (?&normal_identifier)
              |   \{ \s* (?&normal_identifier) \s* \}
          )
      )
      (?<normal_identifier>
          (?: :: )* '?
           (?&basic_identifier)
           (?: (?= (?: :: )+ '? | (?: :: )* ' ) (?&normal_identifier) )?
          (?: :: )*
      )
      (?<basic_identifier>
        # is use utf8 on?
          (?(?{ (caller(0))[8] & $utf8::hint_bits })
              (?&Perl_XIDS) (?&Perl_XIDC)*
            | (?aa) (?!\d) \w+
          )
      )
      (?<sigil> [&*\$\@\%])
      (?<Perl_XIDS> (?[ ( \p{Word} & \p{XID_Start} ) + [_] ]) )
      (?<Perl_XIDC> (?[ \p{Word} & \p{XID_Continue} ]) )
  )
 /x
```

Meanwhile, special identifiers don't follow the above rules; For the most part, all of the identifiers in this category have a special meaning given by Perl. Because they have special parsing rules, these generally can't be fully-qualified. They come in four forms:

- A sigil, followed solely by digits matching `\p{POSIX_Digit}`, like `$0`, `$1`, or `$10000`.

- A sigil, followed by either a caret and a single POSIX uppercase letter, like `$^V` or `$^W`, or a sigil followed by a literal non-space, non-`NUL` control character matching the `\p{POSIX_Cntrl}` property. Due to a historical oddity, if not running under `use utf8`, the 128 characters in the `[0x80-0xff]` range are considered to be controls, and may also be used in length-one variables. However, the use of non-graphical characters is deprecated as of v5.22, and support for them will be removed in a future version of perl. ASCII space characters and `NUL` already aren't allowed, so this means that a single-character variable name with that name being any other C0 control `[0x01-0x1F]`, or `DEL` will generate a deprecated warning. Already, under `"use utf8"`, non-ASCII characters must match `Perl_XIDS`. As of v5.22, when not under `"use utf8"` C1 controls `[0x80-0x9F]`, NO BREAK SPACE, and SOFT HYPHEN (`SHY`)) generate a deprecated warning.
- Similar to the above, a sigil, followed by bareword text in brackets, where the first character is either a caret followed by an uppercase letter, like `${^GLOBAL_PHASE}` or a non-`NUL`, non-space literal control like `${\7LOBAL_PHASE}`. Like the above, when not under `"use utf8"`, the characters in `[0x80-0xFF]` are considered controls, but as of v5.22, the use of any that are non-graphical are deprecated, and as of v5.20 the use of any ASCII-range literal control is deprecated. Support for these will be removed in a future version of perl.
- A sigil followed by a single character matching the `\p{POSIX_Punct}` property, like `$!` or `%+`, except the character `"{"` doesn't work.

Note that as of Perl 5.20, literal control characters in variable names are deprecated; and as of Perl 5.22, any other non-graphic characters are also deprecated.

11.2.3 Context

The interpretation of operations and values in Perl sometimes depends on the requirements of the context around the operation or value. There are two major contexts: list and scalar. Certain operations return list values in contexts wanting a list, and scalar values otherwise. If this is true of an operation it will be mentioned in the documentation for that operation. In other words, Perl overloads certain operations based on whether the expected return value is singular or plural. Some words in English work this way, like "fish" and "sheep".

In a reciprocal fashion, an operation provides either a scalar or a list context to each of its arguments. For example, if you say

```
int( <STDIN> )
```

the integer operation provides scalar context for the `<>` operator, which responds by reading one line from STDIN and passing it back to the integer operation, which will then find the integer value of that line and return that. If, on the other hand, you say

```
sort( <STDIN> )
```

then the sort operation provides list context for `<>`, which will proceed to read every line available up to the end of file, and pass that list of lines back to the sort routine, which will then sort those lines and return them as a list to whatever the context of the sort was.

Assignment is a little bit special in that it uses its left argument to determine the context for the right argument. Assignment to a scalar evaluates the right-hand side in scalar context, while assignment to an array or hash evaluates the righthand side in list context. Assignment to a list (or slice, which is just a list anyway) also evaluates the right-hand side in list context.

When you use the `use warnings` pragma or Perl's `-w` command-line option, you may see warnings about useless uses of constants or functions in "void context". Void context just means the value has been discarded, such as a statement containing only `"fred";` or `getpwuid(0);`. It still counts as scalar context for functions that care whether or not they're being called in list context.

User-defined subroutines may choose to care whether they are being called in a void, scalar, or list context. Most subroutines do not need to bother, though. That's because both scalars and lists are automatically interpolated into lists. See [perlfunc wantarray], page 488 for how you would dynamically discern your function's calling context.

11.2.4 Scalar values

All data in Perl is a scalar, an array of scalars, or a hash of scalars. A scalar may contain one single value in any of three different flavors: a number, a string, or a reference. In general, conversion from one form to another is transparent. Although a scalar may not directly hold multiple values, it may contain a reference to an array or hash which in turn contains multiple values.

Scalars aren't necessarily one thing or another. There's no place to declare a scalar variable to be of type "string", type "number", type "reference", or anything else. Because of the automatic conversion of scalars, operations that return scalars don't need to care (and in fact, cannot care) whether their caller is looking for a string, a number, or a reference. Perl is a contextually polymorphic language whose scalars can be strings, numbers, or references (which includes objects). Although strings and numbers are considered pretty much the same thing for nearly all purposes, references are strongly-typed, uncastable pointers with builtin reference-counting and destructor invocation.

A scalar value is interpreted as FALSE in the Boolean sense if it is undefined, the null string or the number 0 (or its string equivalent, "0"), and TRUE if it is anything else. The Boolean context is just a special kind of scalar context where no conversion to a string or a number is ever performed.

There are actually two varieties of null strings (sometimes referred to as "empty" strings), a defined one and an undefined one. The defined version is just a string of length zero, such as "". The undefined version is the value that indicates that there is no real value for something, such as when there was an error, or at end of file, or when you refer to an uninitialized variable or element of an array or hash. Although in early versions of Perl, an undefined scalar could become defined when first used in a place expecting a defined value, this no longer happens except for rare cases of autovivification as explained in Section 62.1 [perlref NAME], page 1077. You can use the defined() operator to determine whether a scalar value is defined (this has no meaning on arrays or hashes), and the undef() operator to produce an undefined value.

To find out whether a given string is a valid non-zero number, it's sometimes enough to test it against both numeric 0 and also lexical "0" (although this will cause noises if warnings are on). That's because strings that aren't numbers count as 0, just as they do in **awk**:

```
    if ($str == 0 && $str ne "0")  {
```

```
    warn "That doesn't look like a number";
}
```

That method may be best because otherwise you won't treat IEEE notations like `NaN` or `Infinity` properly. At other times, you might prefer to determine whether string data can be used numerically by calling the POSIX::strtod() function or by inspecting your string with a regular expression (as documented in Section 58.1 [perlre NAME], page 989).

```
warn "has nondigits"        if     /\D/;
warn "not a natural number" unless /^\d+$/;             # rejects -3
warn "not an integer"       unless /^-?\d+$/;           # rejects +3
warn "not an integer"       unless /^[+-]?\d+$/;
warn "not a decimal number" unless /^-?\d+\.?\d*$/;     # rejects .2
warn "not a decimal number" unless /^-?(?:\d+(?:\.\d*)?|\.\d+)$/;
warn "not a C float"
    unless /^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/;
```

The length of an array is a scalar value. You may find the length of array @days by evaluating `$#days`, as in **csh**. However, this isn't the length of the array; it's the subscript of the last element, which is a different value since there is ordinarily a 0th element. Assigning to `$#days` actually changes the length of the array. Shortening an array this way destroys intervening values. Lengthening an array that was previously shortened does not recover values that were in those elements.

You can also gain some minuscule measure of efficiency by pre-extending an array that is going to get big. You can also extend an array by assigning to an element that is off the end of the array. You can truncate an array down to nothing by assigning the null list () to it. The following are equivalent:

```
@whatever = ();
$#whatever = -1;
```

If you evaluate an array in scalar context, it returns the length of the array. (Note that this is not true of lists, which return the last value, like the C comma operator, nor of built-in functions, which return whatever they feel like returning.) The following is always true:

```
scalar(@whatever) == $#whatever + 1;
```

Some programmers choose to use an explicit conversion so as to leave nothing to doubt:

```
$element_count = scalar(@whatever);
```

If you evaluate a hash in scalar context, it returns false if the hash is empty. If there are any key/value pairs, it returns true; more precisely, the value returned is a string consisting of the number of used buckets and the number of allocated buckets, separated by a slash. This is pretty much useful only to find out whether Perl's internal hashing algorithm is performing poorly on your data set. For example, you stick 10,000 things in a hash, but evaluating %HASH in scalar context reveals `"1/16"`, which means only one out of sixteen buckets has been touched, and presumably contains all 10,000 of your items. This isn't supposed to happen. If a tied hash is evaluated in scalar context, the `SCALAR` method is called (with a fallback to `FIRSTKEY`).

You can preallocate space for a hash by assigning to the keys() function. This rounds up the allocated buckets to the next power of two:

```
keys(%users) = 1000;                # allocate 1024 buckets
```

11.2.5 Scalar value constructors

Numeric literals are specified in any of the following floating point or integer formats:

```
12345
12345.67
.23E-10             # a very small number
3.14_15_92          # a very important number
4_294_967_296       # underscore for legibility
0xff                # hex
0xdead_beef         # more hex
0377                # octal (only numbers, begins with 0)
0b011011            # binary
0x1.999ap-4         # hexadecimal floating point (the ’p’ is required)
```

You are allowed to use underscores (underbars) in numeric literals between digits for legibility (but not multiple underscores in a row: 23__500 is not legal; 23_500 is). You could, for example, group binary digits by threes (as for a Unix-style mode argument such as 0b110_100_100) or by fours (to represent nibbles, as in 0b1010_0110) or in other groups.

String literals are usually delimited by either single or double quotes. They work much like quotes in the standard Unix shells: double-quoted string literals are subject to backslash and variable substitution; single-quoted strings are not (except for \’ and \\). The usual C-style backslash rules apply for making characters such as newline, tab, etc., as well as some more exotic forms. See Section 48.2.29 [perlop Quote and Quote-like Operators], page 818 for a list.

Hexadecimal, octal, or binary, representations in string literals (e.g. ’0xff’) are not automatically converted to their integer representation. The hex() and oct() functions make these conversions for you. See [perlfunc hex], page 395 and [perlfunc oct], page 406 for more details.

Hexadecimal floating point can start just like a hexadecimal literal, and it can be followed by an optional fractional hexadecimal part, but it must be followed by p, an optional sign, and a power of two. The format is useful for accurately presenting floating point values, avoiding conversions to or from decimal floating point, and therefore avoiding possible loss in precision. Notice that while most current platforms use the 64-bit IEEE 754 floating point, not all do. Another potential source of (low-order) differences are the floating point rounding modes, which can differ between CPUs, operating systems, and compilers, and which Perl doesn’t control.

You can also embed newlines directly in your strings, i.e., they can end on a different line than they begin. This is nice, but if you forget your trailing quote, the error will not be reported until Perl finds another line containing the quote character, which may be much further on in the script. Variable substitution inside strings is limited to scalar variables, arrays, and array or hash slices. (In other words, names beginning with $ or @, followed by an optional bracketed expression as a subscript.) The following code segment prints out "The price is $100."

```
    $Price = ’$100’;    # not interpolated
    print "The price is $Price.\n";     # interpolated
```

There is no double interpolation in Perl, so the $100 is left as is.

By default floating point numbers substituted inside strings use the dot (".") as the decimal separator. If `use locale` is in effect, and POSIX::setlocale() has been called, the character used for the decimal separator is affected by the LC_NUMERIC locale. See Section 38.1 [perllocale NAME], page 701 and `POSIX`.

As in some shells, you can enclose the variable name in braces to disambiguate it from following alphanumerics (and underscores). You must also do this when interpolating a variable into a string to separate the variable name from a following double-colon or an apostrophe, since these would be otherwise treated as a package separator:

```
    $who = "Larry";
    print PASSWD "${who}::0:0:Superuser:/:/bin/perl\n";
    print "We use ${who}speak when ${who}'s here.\n";
```

Without the braces, Perl would have looked for a $whospeak, a `$who::0`, and a `$who's` variable. The last two would be the $0 and the $s variables in the (presumably) non-existent package `who`.

In fact, a simple identifier within such curlies is forced to be a string, and likewise within a hash subscript. Neither need quoting. Our earlier example, `$days{'Feb'}` can be written as `$days{Feb}` and the quotes will be assumed automatically. But anything more complicated in the subscript will be interpreted as an expression. This means for example that `$version{2.0}++` is equivalent to `$version{2}++`, not to `$version{'2.0'}++`.

11.2.5.1 Special floating point: infinity (Inf) and not-a-number (NaN)

Floating point values include the special values `Inf` and `NaN`, for infinity and not-a-number. The infinity can be also negative.

The infinity is the result of certain math operations that overflow the floating point range, like 9**9**9. The not-a-number is the result when the result is undefined or unrepresentable. Though note that you cannot get `NaN` from some common "undefined" or "out-of-range" operations like dividing by zero, or square root of a negative number, since Perl generates fatal errors for those.

The infinity and not-a-number have their own special arithmetic rules. The general rule is that they are "contagious": `Inf` plus one is `Inf`, and `NaN` plus one is `NaN`. Where things get interesting is when you combine infinities and not-a-numbers: `Inf` minus `Inf` and `Inf` divided by `INf` are `NaN` (while `Inf` plus `Inf` is `Inf` and `Inf` times `Inf` is `Inf`). `NaN` is also curious in that it does not equal any number, *including* itself: `NaN` != `NaN`.

Perl doesn't understand `Inf` and `NaN` as numeric literals, but you can have them as strings, and Perl will convert them as needed: "Inf" + 1. (You can, however, import them from the POSIX extension; `use POSIX qw(Inf NaN);` and then use them as literals.)

Note that on input (string to number) Perl accepts `Inf` and `NaN` in many forms. Case is ignored, and the Win32-specific forms like `1.#INF` are understood, but on output the values are normalized to `Inf` and `NaN`.

11.2.5.2 Version Strings

A literal of the form `v1.20.300.4000` is parsed as a string composed of characters with the specified ordinals. This form, known as v-strings, provides an alternative, more readable way to construct strings, rather than use the somewhat less readable interpolation form

"\x{1}\x{14}\x{12c}\x{fa0}". This is useful for representing Unicode strings, and for comparing version "numbers" using the string comparison operators, cmp, gt, lt etc. If there are two or more dots in the literal, the leading v may be omitted.

```
print v9786;              # prints SMILEY, "\x{263a}"
print v102.111.111;       # prints "foo"
print 102.111.111;        # same
```

Such literals are accepted by both require and use for doing a version check. Note that using the v-strings for IPv4 addresses is not portable unless you also use the inet_aton()/inet_ntoa() routines of the Socket package.

Note that since Perl 5.8.1 the single-number v-strings (like v65) are not v-strings before the => operator (which is usually used to separate a hash key from a hash value); instead they are interpreted as literal strings ('v65'). They were v-strings from Perl 5.6.0 to Perl 5.8.0, but that caused more confusion and breakage than good. Multi-number v-strings like v65.66 and 65.66.67 continue to be v-strings always.

11.2.5.3 Special Literals

The special literals __FILE__, __LINE__, and __PACKAGE__ represent the current filename, line number, and package name at that point in your program. __SUB__ gives a reference to the current subroutine. They may be used only as separate tokens; they will not be interpolated into strings. If there is no current package (due to an empty package; directive), __PACKAGE__ is the undefined value. (But the empty package; is no longer supported, as of version 5.10.) Outside of a subroutine, __SUB__ is the undefined value. __SUB__ is only available in 5.16 or higher, and only with a use v5.16 or use feature "current_sub" declaration.

The two control characters ^D and ^Z, and the tokens __END__ and __DATA__ may be used to indicate the logical end of the script before the actual end of file. Any following text is ignored.

Text after __DATA__ may be read via the filehandle PACKNAME::DATA, where PACKNAME is the package that was current when the __DATA__ token was encountered. The filehandle is left open pointing to the line after __DATA__. The program should close DATA when it is done reading from it. (Leaving it open leaks filehandles if the module is reloaded for any reason, so it's a safer practice to close it.) For compatibility with older scripts written before __DATA__ was introduced, __END__ behaves like __DATA__ in the top level script (but not in files loaded with require or do) and leaves the remaining contents of the file accessible via main::DATA.

See SelfLoader for more description of __DATA__, and an example of its use. Note that you cannot read from the DATA filehandle in a BEGIN block: the BEGIN block is executed as soon as it is seen (during compilation), at which point the corresponding __DATA__ (or __END__) token has not yet been seen.

11.2.5.4 Barewords

A word that has no other interpretation in the grammar will be treated as if it were a quoted string. These are known as "barewords". As with filehandles and labels, a bareword that consists entirely of lowercase letters risks conflict with future reserved words, and if you use the use warnings pragma or the -w switch, Perl will warn you about any such words. Perl limits barewords (like identifiers) to about 250 characters. Future versions of Perl are likely to eliminate these arbitrary limitations.

Some people may wish to outlaw barewords entirely. If you say

```
use strict 'subs';
```

then any bareword that would NOT be interpreted as a subroutine call produces a compile-time error instead. The restriction lasts to the end of the enclosing block. An inner block may countermand this by saying `no strict 'subs'`.

11.2.5.5 Array Interpolation

Arrays and slices are interpolated into double-quoted strings by joining the elements with the delimiter specified in the `$"` variable (`$LIST_SEPARATOR` if "use English;" is specified), space by default. The following are equivalent:

```
$temp = join($", @ARGV);
system "echo $temp";

system "echo @ARGV";
```

Within search patterns (which also undergo double-quotish substitution) there is an unfortunate ambiguity: Is `/$foo[bar]/` to be interpreted as `/${foo}[bar]/` (where `[bar]` is a character class for the regular expression) or as `/${foo[bar]}/` (where `[bar]` is the subscript to array @foo)? If @foo doesn't otherwise exist, then it's obviously a character class. If @foo exists, Perl takes a good guess about `[bar]`, and is almost always right. If it does guess wrong, or if you're just plain paranoid, you can force the correct interpretation with curly braces as above.

If you're looking for the information on how to use here-documents, which used to be here, that's been moved to Section 48.2.29 [perlop Quote and Quote-like Operators], page 818.

11.2.6 List value constructors

List values are denoted by separating individual values by commas (and enclosing the list in parentheses where precedence requires it):

```
(LIST)
```

In a context not requiring a list value, the value of what appears to be a list literal is simply the value of the final element, as with the C comma operator. For example,

```
@foo = ('cc', '-E', $bar);
```

assigns the entire list value to array @foo, but

```
$foo = ('cc', '-E', $bar);
```

assigns the value of variable $bar to the scalar variable $foo. Note that the value of an actual array in scalar context is the length of the array; the following assigns the value 3 to $foo:

```
@foo = ('cc', '-E', $bar);
$foo = @foo;                # $foo gets 3
```

You may have an optional comma before the closing parenthesis of a list literal, so that you can say:

```
@foo = (
    1,
    2,
    3,
);
```

To use a here-document to assign an array, one line per element, you might use an approach like this:

```
@sauces = <<End_Lines =~ m/(\S.*\S)/g;
    normal tomato
    spicy tomato
    green chile
    pesto
    white wine
End_Lines
```

LISTs do automatic interpolation of sublists. That is, when a LIST is evaluated, each element of the list is evaluated in list context, and the resulting list value is interpolated into LIST just as if each individual element were a member of LIST. Thus arrays and hashes lose their identity in a LIST–the list

```
(@foo,@bar,&SomeSub,%glarch)
```

contains all the elements of @foo followed by all the elements of @bar, followed by all the elements returned by the subroutine named SomeSub called in list context, followed by the key/value pairs of %glarch. To make a list reference that does *NOT* interpolate, see Section 62.1 [perlref NAME], page 1077.

The null list is represented by (). Interpolating it in a list has no effect. Thus ((),(),()) is equivalent to (). Similarly, interpolating an array with no elements is the same as if no array had been interpolated at that point.

This interpolation combines with the facts that the opening and closing parentheses are optional (except when necessary for precedence) and lists may end with an optional comma to mean that multiple commas within lists are legal syntax. The list `1,,3` is a concatenation of two lists, `1,` and `3`, the first of which ends with that optional comma. `1,,3` is `(1,),(3)` is `1,3` (And similarly for `1,,,3` is `(1,),(,),3` is `1,3` and so on.) Not that we'd advise you to use this obfuscation.

A list value may also be subscripted like a normal array. You must put the list in parentheses to avoid ambiguity. For example:

```
# Stat returns list value.
$time = (stat($file))[8];

# SYNTAX ERROR HERE.
$time = stat($file)[8];  # OOPS, FORGOT PARENTHESES

# Find a hex digit.
$hexdigit = ('a','b','c','d','e','f')[$digit-10];

# A "reverse comma operator".
return (pop(@foo),pop(@foo))[0];
```

Lists may be assigned to only when each element of the list is itself legal to assign to:

```
($a, $b, $c) = (1, 2, 3);

($map{'red'}, $map{'blue'}, $map{'green'}) = (0x00f, 0x0f0, 0xf00);
```

An exception to this is that you may assign to `undef` in a list. This is useful for throwing away some of the return values of a function:

```
($dev, $ino, undef, undef, $uid, $gid) = stat($file);
```

As of Perl 5.22, you can also use `(undef)x2` instead of `undef, undef`. (You can also do `($x) x 2`, which is less useful, because it assigns to the same variable twice, clobbering the first value assigned.)

List assignment in scalar context returns the number of elements produced by the expression on the right side of the assignment:

```
$x = (($foo,$bar) = (3,2,1));       # set $x to 3, not 2
$x = (($foo,$bar) = f());           # set $x to f()'s return count
```

This is handy when you want to do a list assignment in a Boolean context, because most list functions return a null list when finished, which when assigned produces a 0, which is interpreted as FALSE.

It's also the source of a useful idiom for executing a function or performing an operation in list context and then counting the number of return values, by assigning to an empty list and then using that assignment in scalar context. For example, this code:

```
$count = () = $string =~ /\d+/g;
```

will place into $count the number of digit groups found in $string. This happens because the pattern match is in list context (since it is being assigned to the empty list), and will therefore return a list of all matching parts of the string. The list assignment in scalar context will translate that into the number of elements (here, the number of times the pattern matched) and assign that to $count. Note that simply using

```
$count = $string =~ /\d+/g;
```

would not have worked, since a pattern match in scalar context will only return true or false, rather than a count of matches.

The final element of a list assignment may be an array or a hash:

```
($a, $b, @rest) = split;
my($a, $b, %rest) = @_;
```

You can actually put an array or hash anywhere in the list, but the first one in the list will soak up all the values, and anything after it will become undefined. This may be useful in a my() or local().

A hash can be initialized using a literal list holding pairs of items to be interpreted as a key and a value:

```
# same as map assignment above
%map = ('red',0x00f,'blue',0x0f0,'green',0xf00);
```

While literal lists and named arrays are often interchangeable, that's not the case for hashes. Just because you can subscript a list value like a normal array does not mean that you can subscript a list value as a hash. Likewise, hashes included as parts of other lists (including parameters lists and return lists from functions) always flatten out into key/value pairs. That's why it's good to use references sometimes.

It is often more readable to use the => operator between key/value pairs. The => operator is mostly just a more visually distinctive synonym for a comma, but it also arranges for its left-hand operand to be interpreted as a string if it's a bareword that would be a legal simple identifier. => doesn't quote compound identifiers, that contain double colons. This makes it nice for initializing hashes:

```
    %map = (
                 red   => 0x00f,
                 blue  => 0x0f0,
                 green => 0xf00,
   );
```

or for initializing hash references to be used as records:

```
    $rec = {
                witch => 'Mable the Merciless',
                cat   => 'Fluffy the Ferocious',
                date  => '10/31/1776',
    };
```

or for using call-by-named-parameter to complicated functions:

```
   $field = $query->radio_group(
               name      => 'group_name',
               values    => ['eenie','meenie','minie'],
               default   => 'meenie',
               linebreak => 'true',
               labels    => \%labels
   );
```

Note that just because a hash is initialized in that order doesn't mean that it comes out in that order. See ⟨undefined⟩ [perlfunc sort], page ⟨undefined⟩ for examples of how to arrange for an output ordering.

If a key appears more than once in the initializer list of a hash, the last occurrence wins:

```
    %circle = (
                  center => [5, 10],
                  center => [27, 9],
                  radius => 100,
                  color => [0xDF, 0xFF, 0x00],
                  radius => 54,
    );

    # same as
    %circle = (
                  center => [27, 9],
                  color => [0xDF, 0xFF, 0x00],
                  radius => 54,
    );
```

This can be used to provide overridable configuration defaults:

```
# values in %args take priority over %config_defaults
%config = (%config_defaults, %args);
```

11.2.7 Subscripts

An array can be accessed one scalar at a time by specifying a dollar sign ($), then the name of the array (without the leading @), then the subscript inside square brackets. For example:

```
@myarray = (5, 50, 500, 5000);
print "The Third Element is", $myarray[2], "\n";
```

The array indices start with 0. A negative subscript retrieves its value from the end. In our example, $myarray[-1] would have been 5000, and $myarray[-2] would have been 500.

Hash subscripts are similar, only instead of square brackets curly brackets are used. For example:

```
%scientists =
(
    "Newton" => "Isaac",
    "Einstein" => "Albert",
    "Darwin" => "Charles",
    "Feynman" => "Richard",
);

print "Darwin's First Name is ", $scientists{"Darwin"}, "\n";
```

You can also subscript a list to get a single element from it:

```
$dir = (getpwnam("daemon"))[7];
```

11.2.8 Multi-dimensional array emulation

Multidimensional arrays may be emulated by subscripting a hash with a list. The elements of the list are joined with the subscript separator (see [perlvar $;], page 1380).

```
$foo{$a,$b,$c}
```

is equivalent to

```
$foo{join($;, $a, $b, $c)}
```

The default subscript separator is "\034", the same as SUBSEP in **awk**.

11.2.9 Slices

A slice accesses several elements of a list, an array, or a hash simultaneously using a list of subscripts. It's more convenient than writing out the individual elements as a list of separate scalar values.

```
($him, $her)   = @folks[0,-1];              # array slice
@them          = @folks[0 .. 3];            # array slice
($who, $home)  = @ENV{"USER", "HOME"};      # hash slice
($uid, $dir)   = (getpwnam("daemon"))[2,7]; # list slice
```

Since you can assign to a list of variables, you can also assign to an array or hash slice.

```
@days[3..5]    = qw/Wed Thu Fri/;
@colors{'red','blue','green'}
               = (0xff0000, 0x0000ff, 0x00ff00);
@folks[0, -1]  = @folks[-1, 0];
```

The previous assignments are exactly equivalent to

```
($days[3], $days[4], $days[5]) = qw/Wed Thu Fri/;
($colors{'red'}, $colors{'blue'}, $colors{'green'})
               = (0xff0000, 0x0000ff, 0x00ff00);
($folks[0], $folks[-1]) = ($folks[-1], $folks[0]);
```

Since changing a slice changes the original array or hash that it's slicing, a `foreach` construct will alter some–or even all–of the values of the array or hash.

```
foreach (@array[ 4 .. 10 ]) { s/peter/paul/ }

foreach (@hash{qw[key1 key2]}) {
    s/^\s+//;           # trim leading whitespace
    s/\s+$//;           # trim trailing whitespace
    s/(\w+)/\u\L$1/g;   # "titlecase" words
}
```

As a special exception, when you slice a list (but not an array or a hash), if the list evaluates to empty, then taking a slice of that empty list will always yield the empty list in turn. Thus:

```
@a = ()[0,1];          # @a has no elements
@b = (@a)[0,1];        # @b has no elements
@c = (sub{}->())[0,1]; # @c has no elements
@d = ('a','b')[0,1];   # @d has two elements
@e = (@d)[0,1,8,9];    # @e has four elements
@f = (@d)[8,9];        # @f has two elements
```

This makes it easy to write loops that terminate when a null list is returned:

```
while ( ($home, $user) = (getpwent)[7,0] ) {
    printf "%-8s %s\n", $user, $home;
}
```

As noted earlier in this document, the scalar sense of list assignment is the number of elements on the right-hand side of the assignment. The null list contains no elements, so when the password file is exhausted, the result is 0, not 2.

Slices in scalar context return the last item of the slice.

```
@a = qw/first second third/;
%h = (first => 'A', second => 'B');
$t = @a[0, 1];                  # $t is now 'second'
$u = @h{'first', 'second'};     # $u is now 'B'
```

If you're confused about why you use an '@' there on a hash slice instead of a '%', think of it like this. The type of bracket (square or curly) governs whether it's an array or a hash being looked at. On the other hand, the leading symbol ('$' or '@') on the array or hash indicates whether you are getting back a singular value (a scalar) or a plural one (a list).

11.2.9.1 Key/Value Hash Slices

Starting in Perl 5.20, a hash slice operation with the % symbol is a variant of slice operation returning a list of key/value pairs rather than just values:

```
%h = (blonk => 2, foo => 3, squink => 5, bar => 8);
%subset = %h{'foo', 'bar'}; # key/value hash slice
# %subset is now (foo => 3, bar => 8)
```

However, the result of such a slice cannot be localized, deleted or used in assignment. These are otherwise very much consistent with hash slices using the @ symbol.

11.2.9.2 Index/Value Array Slices

Similar to key/value hash slices (and also introduced in Perl 5.20), the % array slice syntax returns a list of index/value pairs:

```
@a = "a".."z";
@list = %a[3,4,6];
# @list is now (3, "d", 4, "e", 6, "g")
```

11.2.10 Typeglobs and Filehandles

Perl uses an internal type called a *typeglob* to hold an entire symbol table entry. The type prefix of a typeglob is a *, because it represents all types. This used to be the preferred way to pass arrays and hashes by reference into a function, but now that we have real references, this is seldom needed.

The main use of typeglobs in modern Perl is create symbol table aliases. This assignment:

```
*this = *that;
```

makes $this an alias for $that, @this an alias for @that, %this an alias for %that, &this an alias for &that, etc. Much safer is to use a reference. This:

```
local *Here::blue = \$There::green;
```

temporarily makes $Here::blue an alias for $There::green, but doesn't make @Here::blue an alias for @There::green, or %Here::blue an alias for %There::green, etc. See Section 40.2.3 [perlmod Symbol Tables], page 733 for more examples of this. Strange though this may seem, this is the basis for the whole module import/export system.

Another use for typeglobs is to pass filehandles into a function or to create new filehandles. If you need to use a typeglob to save away a filehandle, do it this way:

```
$fh = *STDOUT;
```

or perhaps as a real reference, like this:

```
$fh = \*STDOUT;
```

See Section 73.1 [perlsub NAME], page 1216 for examples of using these as indirect filehandles in functions.

Typeglobs are also a way to create a local filehandle using the local() operator. These last until their block is exited, but may be passed back. For example:

```
sub newopen {
    my $path = shift;
    local  *FH;  # not my!
    open   (FH, $path)          or  return undef;
```

```
        return *FH;
    }
    $fh = newopen('/etc/passwd');
```

Now that we have the `*foo{THING}` notation, typeglobs aren't used as much for filehandle manipulations, although they're still needed to pass brand new file and directory handles into or out of functions. That's because `*HANDLE{IO}` only works if HANDLE has already been used as a handle. In other words, `*FH` must be used to create new symbol table entries; `*foo{THING}` cannot. When in doubt, use `*FH`.

All functions that are capable of creating filehandles (open(), opendir(), pipe(), socketpair(), sysopen(), socket(), and accept()) automatically create an anonymous filehandle if the handle passed to them is an uninitialized scalar variable. This allows the constructs such as `open(my $fh, ...)` and `open(local $fh,...)` to be used to create filehandles that will conveniently be closed automatically when the scope ends, provided there are no other references to them. This largely eliminates the need for typeglobs when opening filehandles that must be passed around, as in the following example:

```
    sub myopen {
        open my $fh, "@_"
             or die "Can't open '@_': $!";
        return $fh;
    }

    {
        my $f = myopen("</etc/motd");
        print <$f>;
        # $f implicitly closed here
    }
```

Note that if an initialized scalar variable is used instead the result is different: `my $fh='zzz'; open($fh, ...)` is equivalent to `open( *{'zzz'}, ...)`. `use strict 'refs'` forbids such practice.

Another way to create anonymous filehandles is with the Symbol module or with the IO::Handle module and its ilk. These modules have the advantage of not hiding different types of the same name during the local(). See the bottom of ⟨undefined⟩ [perlfunc open], page ⟨undefined⟩ for an example.

11.3 SEE ALSO

See Section 86.1 [perlvar NAME], page 1375 for a description of Perl's built-in variables and a discussion of legal variable names. See Section 62.1 [perlref NAME], page 1077, Section 73.1 [perlsub NAME], page 1216, and Section 40.2.3 [perlmod Symbol Tables], page 733 for more discussion on typeglobs and the `*foo{THING}` syntax.

12 perldbmfilter

12.1 NAME

perldbmfilter - Perl DBM Filters

12.2 SYNOPSIS

```
    $db = tie %hash, 'DBM', ...

    $old_filter = $db->filter_store_key  ( sub { ... } );
    $old_filter = $db->filter_store_value( sub { ... } );
    $old_filter = $db->filter_fetch_key  ( sub { ... } );
    $old_filter = $db->filter_fetch_value( sub { ... } );
```

12.3 DESCRIPTION

The four `filter_*` methods shown above are available in all the DBM modules that ship with Perl, namely DB_File, GDBM_File, NDBM_File, ODBM_File and SDBM_File.

Each of the methods works identically, and is used to install (or uninstall) a single DBM Filter. The only difference between them is the place that the filter is installed.

To summarise:

filter_store_key
If a filter has been installed with this method, it will be invoked every time you write a key to a DBM database.

filter_store_value
If a filter has been installed with this method, it will be invoked every time you write a value to a DBM database.

filter_fetch_key
If a filter has been installed with this method, it will be invoked every time you read a key from a DBM database.

filter_fetch_value
If a filter has been installed with this method, it will be invoked every time you read a value from a DBM database.

You can use any combination of the methods from none to all four.

All filter methods return the existing filter, if present, or `undef` if not.

To delete a filter pass `undef` to it.

12.3.1 The Filter

When each filter is called by Perl, a local copy of `$_` will contain the key or value to be filtered. Filtering is achieved by modifying the contents of `$_`. The return code from the filter is ignored.

12.3.2 An Example: the NULL termination problem.

DBM Filters are useful for a class of problems where you *always* want to make the same transformation to all keys, all values or both.

For example, consider the following scenario. You have a DBM database that you need to share with a third-party C application. The C application assumes that *all* keys and values are NULL terminated. Unfortunately when Perl writes to DBM databases it doesn't use NULL termination, so your Perl application will have to manage NULL termination itself. When you write to the database you will have to use something like this:

```
$hash{"$key\0"} = "$value\0";
```

Similarly the NULL needs to be taken into account when you are considering the length of existing keys/values.

It would be much better if you could ignore the NULL terminations issue in the main application code and have a mechanism that automatically added the terminating NULL to all keys and values whenever you write to the database and have them removed when you read from the database. As I'm sure you have already guessed, this is a problem that DBM Filters can fix very easily.

```
use strict;
use warnings;
use SDBM_File;
use Fcntl;

my %hash;
my $filename = "filt";
unlink $filename;

my $db = tie(%hash, 'SDBM_File', $filename, O_RDWR|O_CREAT, 0640)
  or die "Cannot open $filename: $!\n";

# Install DBM Filters
$db->filter_fetch_key  ( sub { s/\0$//    } );
$db->filter_store_key  ( sub { $_ .= "\0" } );
$db->filter_fetch_value(
    sub { no warnings 'uninitialized'; s/\0$// } );
$db->filter_store_value( sub { $_ .= "\0" } );

$hash{"abc"} = "def";
my $a = $hash{"ABC"};
# ...
undef $db;
untie %hash;
```

The code above uses SDBM_File, but it will work with any of the DBM modules.

Hopefully the contents of each of the filters should be self-explanatory. Both "fetch" filters remove the terminating NULL, and both "store" filters add a terminating NULL.

12.3.3 Another Example: Key is a C int.

Here is another real-life example. By default, whenever Perl writes to a DBM database it always writes the key and value as strings. So when you use this:

```
$hash{12345} = "something";
```

the key 12345 will get stored in the DBM database as the 5 byte string "12345". If you actually want the key to be stored in the DBM database as a C int, you will have to use pack when writing, and unpack when reading.

Here is a DBM Filter that does it:

```
use strict;
use warnings;
use DB_File;
my %hash;
my $filename = "filt";
unlink $filename;


my $db = tie %hash, 'DB_File', $filename, O_CREAT|O_RDWR, 0666,
    $DB_HASH or die "Cannot open $filename: $!\n";

$db->filter_fetch_key  ( sub { $_ = unpack("i", $_) } );
$db->filter_store_key  ( sub { $_ = pack ("i", $_) } );
$hash{123} = "def";
# ...
undef $db;
untie %hash;
```

The code above uses DB_File, but again it will work with any of the DBM modules.

This time only two filters have been used; we only need to manipulate the contents of the key, so it wasn't necessary to install any value filters.

12.4 SEE ALSO

DB_File, GDBM_File, NDBM_File, ODBM_File and SDBM_File.

12.5 AUTHOR

Paul Marquess

13 perldebguts

13.1 NAME

perldebguts - Guts of Perl debugging

13.2 DESCRIPTION

This is not Section 15.1 [perldebug NAME], page 120, which tells you how to use the debugger. This manpage describes low-level details concerning the debugger's internals, which range from difficult to impossible to understand for anyone who isn't incredibly intimate with Perl's guts. Caveat lector.

13.3 Debugger Internals

Perl has special debugging hooks at compile-time and run-time used to create debugging environments. These hooks are not to be confused with the *perl -Dxxx* command described in Section 69.1 [perlrun NAME], page 1176, which is usable only if a special Perl is built per the instructions in the `INSTALL` podpage in the Perl source tree.

For example, whenever you call Perl's built-in `caller` function from the package `DB`, the arguments that the corresponding stack frame was called with are copied to the `@DB::args` array. These mechanisms are enabled by calling Perl with the **-d** switch. Specifically, the following additional features are enabled (cf. [perlvar $^P], page 1405):

- Perl inserts the contents of `$ENV{PERL5DB}` (or `BEGIN {require 'perl5db.pl'}` if not present) before the first line of your program.
- Each array `@{"_<$filename"}` holds the lines of $filename for a file compiled by Perl. The same is also true for `eval`ed strings that contain subroutines, or which are currently being executed. The $filename for `eval`ed strings looks like `(eval 34)`.

  Values in this array are magical in numeric context: they compare equal to zero only if the line is not breakable.
- Each hash `%{"_<$filename"}` contains breakpoints and actions keyed by line number. Individual entries (as opposed to the whole hash) are settable. Perl only cares about Boolean true here, although the values used by `perl5db.pl` have the form `"$break_condition\0$action"`.

  The same holds for evaluated strings that contain subroutines, or which are currently being executed. The $filename for `eval`ed strings looks like `(eval 34)`.
- Each scalar `${"_<$filename"}` contains `"_<$filename"`. This is also the case for evaluated strings that contain subroutines, or which are currently being executed. The $filename for `eval`ed strings looks like `(eval 34)`.
- After each `require`d file is compiled, but before it is executed, `DB::postponed(*{"_<$filename"})` is called if the subroutine `DB::postponed` exists. Here, the $filename is the expanded name of the `require`d file, as found in the values of %INC.
- After each subroutine `subname` is compiled, the existence of `$DB::postponed{subname}` is checked. If this key exists, `DB::postponed(subname)` is called if the `DB::postponed` subroutine also exists.

- A hash `%DB::sub` is maintained, whose keys are subroutine names and whose values have the form `filename:startline-endline`. `filename` has the form `(eval 34)` for subroutines defined inside `eval`s.
- When the execution of your program reaches a point that can hold a breakpoint, the `DB::DB()` subroutine is called if any of the variables `$DB::trace`, `$DB::single`, or `$DB::signal` is true. These variables are not `local`izable. This feature is disabled when executing inside `DB::DB()`, including functions called from it unless `$^D & (1<<30)` is true.
- When execution of the program reaches a subroutine call, a call to `&DB::sub`(*args*) is made instead, with `$DB::sub` holding the name of the called subroutine. (This doesn't happen if the subroutine was compiled in the `DB` package.)

  If the call is to an lvalue subroutine, and `&DB::lsub` is defined `&DB::lsub`(*args*) is called instead, otherwise falling back to `&DB::sub`(*args*).
- When execution of the program uses `goto` to enter a non-XS subroutine and the 0x80 bit is set in `$^P`, a call to `&DB::goto` is made, with `$DB::sub` holding the name of the subroutine being entered.

Note that if `&DB::sub` needs external data for it to work, no subroutine call is possible without it. As an example, the standard debugger's `&DB::sub` depends on the `$DB::deep` variable (it defines how many levels of recursion deep into the debugger you can go before a mandatory break). If `$DB::deep` is not defined, subroutine calls are not possible, even though `&DB::sub` exists.

13.3.1 Writing Your Own Debugger

13.3.1.1 Environment Variables

The `PERL5DB` environment variable can be used to define a debugger. For example, the minimal "working" debugger (it actually doesn't do anything) consists of one line:

```
sub DB::DB {}
```

It can easily be defined like this:

```
$ PERL5DB="sub DB::DB {}" perl -d your-script
```

Another brief debugger, slightly more useful, can be created with only the line:

```
sub DB::DB {print ++$i; scalar <STDIN>}
```

This debugger prints a number which increments for each statement encountered and waits for you to hit a newline before continuing to the next statement.

The following debugger is actually useful:

```
{
  package DB;
  sub DB  {}
  sub sub {print ++$i, " $sub\n"; &$sub}
}
```

It prints the sequence number of each subroutine call and the name of the called subroutine. Note that `&DB::sub` is being compiled into the package `DB` through the use of the `package` directive.

When it starts, the debugger reads your rc file (`./.perldb` or `~/.perldb` under Unix), which can set important options. (A subroutine (`&afterinit`) can be defined here as well; it is executed after the debugger completes its own initialization.)

After the rc file is read, the debugger reads the PERLDB_OPTS environment variable and uses it to set debugger options. The contents of this variable are treated as if they were the argument of an `o ...` debugger command (q.v. in Section 15.3.3 [perldebug Configurable Options], page 128).

13.3.1.2 Debugger Internal Variables

In addition to the file and subroutine-related variables mentioned above, the debugger also maintains various magical internal variables.

- `@DB::dbline` is an alias for `@{"::_<current_file"}`, which holds the lines of the currently-selected file (compiled by Perl), either explicitly chosen with the debugger's `f` command, or implicitly by flow of execution.

  Values in this array are magical in numeric context: they compare equal to zero only if the line is not breakable.

- `%DB::dbline` is an alias for `%{"::_<current_file"}`, which contains breakpoints and actions keyed by line number in the currently-selected file, either explicitly chosen with the debugger's `f` command, or implicitly by flow of execution.

  As previously noted, individual entries (as opposed to the whole hash) are settable. Perl only cares about Boolean true here, although the values used by `perl5db.pl` have the form `"$break_condition\0$action"`.

13.3.1.3 Debugger Customization Functions

Some functions are provided to simplify customization.

- See Section 15.3.3 [perldebug Configurable Options], page 128 for a description of options parsed by `DB::parse_options(string)`.
- `DB::dump_trace(skip[,count])` skips the specified number of frames and returns a list containing information about the calling frames (all of them, if `count` is missing). Each entry is reference to a hash with keys `context` (either `.`, `$`, or `@`), `sub` (subroutine name, or info about `eval`), `args` (`undef` or a reference to an array), `file`, and `line`.
- `DB::print_trace(FH, skip[, count[, short]])` prints formatted info about caller frames. The last two functions may be convenient as arguments to `<`, `<<` commands.

Note that any variables and functions that are not documented in this manpages (or in Section 15.1 [perldebug NAME], page 120) are considered for internal use only, and as such are subject to change without notice.

13.4 Frame Listing Output Examples

The `frame` option can be used to control the output of frame information. For example, contrast this expression trace:

```
$ perl -de 42
Stack dump during die enabled outside of evals.
```

```
Loading DB routines from perl5db.pl patch level 0.94
Emacs support available.

Enter h or 'h h' for help.

main::(-e:1):   0
  DB<1> sub foo { 14 }

  DB<2> sub bar { 3 }

  DB<3> t print foo() * bar()
main::((eval 172):3):   print foo() + bar();
main::foo((eval 168):2):
main::bar((eval 170):2):
42
```

with this one, once the option `frame=2` has been set:

```
  DB<4> o f=2
               frame = '2'
  DB<5> t print foo() * bar()
3:      foo() * bar()
entering main::foo
 2:     sub foo { 14 };
exited main::foo
entering main::bar
 2:     sub bar { 3 };
exited main::bar
42
```

By way of demonstration, we present below a laborious listing resulting from setting your `PERLDB_OPTS` environment variable to the value `f=n N`, and running *perl -d -V* from the command line. Examples using various values of `n` are shown to give you a feel for the difference between settings. Long though it may be, this is not a complete listing, but only excerpts.

1.

```
entering main::BEGIN
 entering Config::BEGIN
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
 Package lib/Config.pm.
 entering Config::TIEHASH
 entering Exporter::import
  entering Exporter::export
entering Config::myconfig
 entering Config::FETCH
 entering Config::FETCH
 entering Config::FETCH
 entering Config::FETCH
```

2.

```
entering main::BEGIN
 entering Config::BEGIN
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
 exited Config::BEGIN
 Package lib/Config.pm.
 entering Config::TIEHASH
 exited Config::TIEHASH
 entering Exporter::import
  entering Exporter::export
  exited Exporter::export
 exited Exporter::import
exited main::BEGIN
entering Config::myconfig
 entering Config::FETCH
 exited Config::FETCH
 entering Config::FETCH
 exited Config::FETCH
 entering Config::FETCH
```

3.

```
in  $=main::BEGIN() from /dev/null:0
 in  $=Config::BEGIN() from lib/Config.pm:2
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
 Package lib/Config.pm.
 in  $=Config::TIEHASH('Config') from lib/Config.pm:644
 in  $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
  in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from li
in  @=Config::myconfig() from /dev/null:0
 in  $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'PERL_VERSION') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'PERL_SUBVERSION') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'osname') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'osvers') from lib/Config.pm:574
```

4.

```
in  $=main::BEGIN() from /dev/null:0
 in  $=Config::BEGIN() from lib/Config.pm:2
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
 out $=Config::BEGIN() from lib/Config.pm:0
 Package lib/Config.pm.
 in  $=Config::TIEHASH('Config') from lib/Config.pm:644
 out $=Config::TIEHASH('Config') from lib/Config.pm:644
 in  $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
```

```
  in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/
  out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/
 out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
out $=main::BEGIN() from /dev/null:0
in  @=Config::myconfig() from /dev/null:0
 in  $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
 out $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
 out $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'PERL_VERSION') from lib/Config.pm:574
 out $=Config::FETCH(ref(Config), 'PERL_VERSION') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'PERL_SUBVERSION') from lib/Config.pm:574
```

5.

```
in  $=main::BEGIN() from /dev/null:0
 in  $=Config::BEGIN() from lib/Config.pm:2
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
 out $=Config::BEGIN() from lib/Config.pm:0
 Package lib/Config.pm.
 in  $=Config::TIEHASH('Config') from lib/Config.pm:644
 out $=Config::TIEHASH('Config') from lib/Config.pm:644
 in  $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
  in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/E
  out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/E
 out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
out $=main::BEGIN() from /dev/null:0
in  @=Config::myconfig() from /dev/null:0
 in  $=Config::FETCH('Config=HASH(0x1aa444)', 'package') from lib/Config.pm:574
 out $=Config::FETCH('Config=HASH(0x1aa444)', 'package') from lib/Config.pm:574
 in  $=Config::FETCH('Config=HASH(0x1aa444)', 'baserev') from lib/Config.pm:574
 out $=Config::FETCH('Config=HASH(0x1aa444)', 'baserev') from lib/Config.pm:574
```

6.

```
in  $=CODE(0x15eca4)() from /dev/null:0
 in  $=CODE(0x182528)() from lib/Config.pm:2
  Package lib/Exporter.pm.
 out $=CODE(0x182528)() from lib/Config.pm:0
 scalar context return from CODE(0x182528): undef
 Package lib/Config.pm.
 in  $=Config::TIEHASH('Config') from lib/Config.pm:628
 out $=Config::TIEHASH('Config') from lib/Config.pm:628
 scalar context return from Config::TIEHASH:   empty hash
 in  $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
  in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/Exporte
  out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/Exporte
  scalar context return from Exporter::export: ''
 out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
```

```
    scalar context return from Exporter::import: ''
```

In all cases shown above, the line indentation shows the call tree. If bit 2 of `frame` is set, a line is printed on exit from a subroutine as well. If bit 4 is set, the arguments are printed along with the caller info. If bit 8 is set, the arguments are printed even if they are tied or references. If bit 16 is set, the return value is printed, too.

When a package is compiled, a line like this

```
 Package lib/Carp.pm.
```

is printed with proper indentation.

13.5 Debugging Regular Expressions

There are two ways to enable debugging output for regular expressions.

If your perl is compiled with `-DDEBUGGING`, you may use the **-Dr** flag on the command line.

Otherwise, one can `use re 'debug'`, which has effects at compile time and run time. Since Perl 5.9.5, this pragma is lexically scoped.

13.5.1 Compile-time Output

The debugging output at compile time looks like this:

```
Compiling REx '[bc]d(ef*g)+h[ij]k$'
size 45 Got 364 bytes for offset annotations.
first at 1
rarest char g at 0
rarest char d at 0
   1: ANYOF[bc](12)
  12: EXACT <d>(14)
  14: CURLYX[0] {1,32767}(28)
  16:   OPEN1(18)
  18:     EXACT <e>(20)
  20:     STAR(23)
  21:       EXACT <f>(0)
  23:     EXACT <g>(25)
  25:   CLOSE1(27)
  27:   WHILEM[1/1](0)
  28: NOTHING(29)
  29: EXACT <h>(31)
  31: ANYOF[ij](42)
  42: EXACT <k>(44)
  44: EOL(45)
  45: END(0)
anchored 'de' at 1 floating 'gh' at 3..2147483647 (checking floating)
      stclass 'ANYOF[bc]' minlen 7
Offsets: [45]
      1[4] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 5[1]
      0[0] 12[1] 0[0] 6[1] 0[0] 7[1] 0[0] 9[1] 8[1] 0[0] 10[1] 0[0]
```

```
        11[1] 0[0] 12[0] 12[0] 13[1] 0[0] 14[4] 0[0] 0[0] 0[0] 0[0]
        0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 18[1] 0[0] 19[1] 20[0]
Omitting $‘ $& $’ support.
```

The first line shows the pre-compiled form of the regex. The second shows the size of the compiled form (in arbitrary units, usually 4-byte words) and the total number of bytes allocated for the offset/length table, usually 4+`size`*8. The next line shows the label *id* of the first node that does a match.

The

```
anchored ’de’ at 1 floating ’gh’ at 3..2147483647 (checking floating)
        stclass ’ANYOF[bc]’ minlen 7
```

line (split into two lines above) contains optimizer information. In the example shown, the optimizer found that the match should contain a substring `de` at offset 1, plus substring `gh` at some offset between 3 and infinity. Moreover, when checking for these substrings (to abandon impossible matches quickly), Perl will check for the substring `gh` before checking for the substring `de`. The optimizer may also use the knowledge that the match starts (at the `first` *id*) with a character class, and no string shorter than 7 characters can possibly match.

The fields of interest which may appear in this line are

`anchored` *STRING* `at` *POS*
`floating` *STRING* `at` *POS1..POS2*

See above.

`matching floating/anchored`

Which substring to check first.

`minlen`

The minimal length of the match.

`stclass` *TYPE*

Type of first matching node.

`noscan`

Don’t scan for the found substrings.

`isall`

Means that the optimizer information is all that the regular expression contains, and thus one does not need to enter the regex engine at all.

`GPOS`

Set if the pattern contains `\G`.

`plus`

Set if the pattern starts with a repeated char (as in `x+y`).

`implicit`

Set if the pattern starts with `.*`.

`with eval`

Set if the pattern contain eval-groups, such as `(?{ code })` and `(??{ code })`.

`anchored(TYPE)`
: If the pattern may match only at a handful of places, with `TYPE` being `SBOL`, `MBOL`, or `GPOS`. See the table below.

If a substring is known to match at end-of-line only, it may be followed by `$`, as in `floating 'k'$`.

The optimizer-specific information is used to avoid entering (a slow) regex engine on strings that will not definitely match. If the `isall` flag is set, a call to the regex engine may be avoided even when the optimizer found an appropriate place for the match.

Above the optimizer section is the list of *nodes* of the compiled form of the regex. Each line has format

id: *TYPE OPTIONAL-INFO* (*next-id*)

13.5.2 Types of Nodes

Here are the current possible types, with short descriptions:

```
 # TYPE arg-description [num-args] [longjump-len] DESCRIPTION

 # Exit points

 END             no         End of program.
 SUCCEED         no         Return from a subroutine, basically.

 # Line Start Anchors:
 SBOL            no         Match "" at beginning of line: /^/, /\A/
 MBOL            no         Same, assuming multiline: /^/m

 # Line End Anchors:
 SEOL            no         Match "" at end of line: /$/
 MEOL            no         Same, assuming multiline: /$/m
 EOS             no         Match "" at end of string: /\z/

 # Match Start Anchors:
 GPOS            no         Matches where last m//g left off.

 # Word Boundary Opcodes:
 BOUND           no         Like BOUNDA for non-utf8, otherwise match ""
                            between any Unicode \w\W or \W\w
 BOUNDL          no         Like BOUND/BOUNDU, but \w and \W are defined
                            by current locale
 BOUNDU          no         Match "" at any boundary of a given type
                            using Unicode rules
 BOUNDA          no         Match "" at any boundary between \w\W or
                            \W\w, where \w is [_a-zA-Z0-9]
 NBOUND          no         Like NBOUNDA for non-utf8, otherwise match
                            "" between any Unicode \w\w or \W\W
```

```
NBOUNDL         no         Like NBOUND/NBOUNDU, but \w and \W are
                           defined by current locale
NBOUNDU         no         Match "" at any non-boundary of a given type
                           using using Unicode rules
NBOUNDA         no         Match "" betweeen any \w\w or \W\W, where \w
                           is [_a-zA-Z0-9]

# [Special] alternatives:
REG_ANY         no         Match any one character (except newline).
SANY            no         Match any one character.
CANY            no         Match any one byte.
ANYOF           sv 1       Match character in (or not in) this class,
                           single char match only
ANYOFL          sv 1       Like ANYOF, but /l is in effect

# POSIX Character Classes:
POSIXD          none       Some [[:class:]] under /d; the FLAGS field
                           gives which one
POSIXL          none       Some [[:class:]] under /l; the FLAGS field
                           gives which one
POSIXU          none       Some [[:class:]] under /u; the FLAGS field
                           gives which one
POSIXA          none       Some [[:class:]] under /a; the FLAGS field
                           gives which one
NPOSIXD         none       complement of POSIXD, [[:^class:]]
NPOSIXL         none       complement of POSIXL, [[:^class:]]
NPOSIXU         none       complement of POSIXU, [[:^class:]]
NPOSIXA         none       complement of POSIXA, [[:^class:]]

CLUMP           no         Match any extended grapheme cluster sequence

# Alternation

# BRANCH        The set of branches constituting a single choice are
#               hooked together with their "next" pointers, since
#               precedence prevents anything being concatenated to
#               any individual branch.  The "next" pointer of the last
#               BRANCH in a choice points to the thing following the
#               whole choice.  This is also where the final "next"
#               pointer of each individual branch points; each branch
#               starts with the operand node of a BRANCH node.
#
BRANCH          node       Match this alternative, or the next...

# Literals

EXACT           str        Match this string (preceded by length).
```

```
EXACTL          str        Like EXACT, but /l is in effect.
EXACTF          str        Match this non-UTF-8 string (not guaranteed
                           to be folded) using /id rules (w/len).
EXACTFL         str        Match this string (not guaranteed to be
                           folded) using /il rules (w/len).
EXACTFU         str        Match this string (folded iff in UTF-8,
                           length in folding doesn't change if not in
                           UTF-8) using /iu rules (w/len).
EXACTFA         str        Match this string (not guaranteed to be
                           folded) using /iaa rules (w/len).

EXACTFU_SS      str        Match this string (folded iff in UTF-8,
                           length in folding may change even if not in
                           UTF-8) using /iu rules (w/len).
EXACTFLU8       str        Rare cirucmstances: like EXACTFU, but is
                           under /l, UTF-8, folded, and everything in
                           it is above 255.
EXACTFA_NO_TRIE str        Match this string (which is not trie-able;
                           not guaranteed to be folded) using /iaa
                           rules (w/len).

# Do nothing types

NOTHING         no         Match empty string.
# A variant of above which delimits a group, thus stops optimizations
TAIL            no         Match empty string. Can jump here from
                           outside.

# Loops

# STAR,PLUS    '?', and complex '*' and '+', are implemented as
#               circular BRANCH structures.  Simple cases
#               (one character per match) are implemented with STAR
#               and PLUS for speed and to minimize recursive plunges.
#
STAR            node       Match this (simple) thing 0 or more times.
PLUS            node       Match this (simple) thing 1 or more times.

CURLY           sv 2       Match this simple thing {n,m} times.
CURLYN          no 2       Capture next-after-this simple thing
CURLYM          no 2       Capture this medium-complex thing {n,m}
                           times.
CURLYX          sv 2       Match this complex thing {n,m} times.

# This terminator creates a loop structure for CURLYX
WHILEM          no         Do curly processing and see if rest matches.
```

```
# Buffer related

# OPEN,CLOSE,GROUPP     ...are numbered at compile time.
OPEN            num 1      Mark this point in input as start of #n.
CLOSE           num 1      Analogous to OPEN.

REF             num 1      Match some already matched string
REFF            num 1      Match already matched string, folded using
                           native charset rules for non-utf8
REFFL           num 1      Match already matched string, folded in loc.
REFFU           num 1      Match already matched string, folded using
                           unicode rules for non-utf8
REFFA           num 1      Match already matched string, folded using
                           unicode rules for non-utf8, no mixing ASCII,
                           non-ASCII

# Named references.  Code in regcomp.c assumes that these all are after
# the numbered references
NREF            no-sv 1    Match some already matched string
NREFF           no-sv 1    Match already matched string, folded using
                           native charset rules for non-utf8
NREFFL          no-sv 1    Match already matched string, folded in loc.
NREFFU          num 1      Match already matched string, folded using
                           unicode rules for non-utf8
NREFFA          num 1      Match already matched string, folded using
                           unicode rules for non-utf8, no mixing ASCII,
                           non-ASCII

# Support for long RE
LONGJMP         off 1 1    Jump far away.
BRANCHJ         off 1 1    BRANCH with long offset.

# Special Case Regops
IFMATCH         off 1 2    Succeeds if the following matches.
UNLESSM         off 1 2    Fails if the following matches.
SUSPEND         off 1 1    "Independent" sub-RE.
IFTHEN          off 1 1    Switch, should be preceded by switcher.
GROUPP          num 1      Whether the group matched.

# The heavy worker

EVAL            evl/flags  Execute some Perl code.
                2L

# Modifiers

MINMOD          no         Next operator is not greedy.
```

```
LOGICAL         no         Next opcode should set the flag only.

# This is not used yet
RENUM           off 1 1    Group with independently numbered parens.

# Trie Related

# Behave the same as A|LIST|OF|WORDS would. The '..C' variants
# have inline charclass data (ascii only), the 'C' store it in the
# structure.

TRIE            trie 1     Match many EXACT(F[ALU]?)? at once.
                           flags==type
TRIEC           trie       Same as TRIE, but with embedded charclass
                charclass  data

AHOCORASICK     trie 1     Aho Corasick stclass. flags==type
AHOCORASICKC    trie       Same as AHOCORASICK, but with embedded
                charclass  charclass data

# Regex Subroutines
GOSUB           num/ofs 2L recurse to paren arg1 at (signed) ofs arg2
GOSTART         no         recurse to start of pattern

# Special conditionals
NGROUPP         no-sv 1    Whether the group matched.
INSUBP          num 1      Whether we are in a specific recurse.
DEFINEP         none 1     Never execute directly.

# Backtracking Verbs
ENDLIKE         none       Used only for the type field of verbs
OPFAIL          none       Same as (?!)
ACCEPT          parno 1    Accepts the current matched string.

# Verbs With Arguments
VERB            no-sv 1    Used only for the type field of verbs
PRUNE           no-sv 1    Pattern fails at this startpoint if no-
                           backtracking through this
MARKPOINT       no-sv 1    Push the current location for rollback by
                           cut.
SKIP            no-sv 1    On failure skip forward (to the mark) before
                           retrying
COMMIT          no-sv 1    Pattern fails outright if backtracking
                           through this
CUTGROUP        no-sv 1    On failure go to the next alternation in the
                           group
```

```
# Control what to keep in $&.
KEEPS           no         $& begins here.

# New charclass like patterns
LNBREAK         none       generic newline pattern

# SPECIAL  REGOPS

# This is not really a node, but an optimized away piece of a "long"
# node.  To simplify debugging output, we mark it as if it were a node
OPTIMIZED       off        Placeholder for dump.

# Special opcode with the property that no opcode in a compiled program
# will ever be of this type. Thus it can be used as a flag value that
# no other opcode has been seen. END is used similarly, in that an END
# node cant be optimized. So END implies "unoptimizable" and PSEUDO
# mean "not seen anything to optimize yet".
PSEUDO          off        Pseudo opcode for internal use.
```

Following the optimizer information is a dump of the offset/length table, here split across several lines:

```
Offsets: [45]
      1[4] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 5[1]
      0[0] 12[1] 0[0] 6[1] 0[0] 7[1] 0[0] 9[1] 8[1] 0[0] 10[1] 0[0]
      11[1] 0[0] 12[0] 12[0] 13[1] 0[0] 14[4] 0[0] 0[0] 0[0] 0[0]
      0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 18[1] 0[0] 19[1] 20[0]
```

The first line here indicates that the offset/length table contains 45 entries. Each entry is a pair of integers, denoted by `offset[length]`. Entries are numbered starting with 1, so entry #1 here is `1[4]` and entry #12 is `5[1]`. `1[4]` indicates that the node labeled `1:` (the `1: ANYOF[bc]`) begins at character position 1 in the pre-compiled form of the regex, and has a length of 4 characters. `5[1]` in position 12 indicates that the node labeled `12:` (the `12: EXACT <d>`) begins at character position 5 in the pre-compiled form of the regex, and has a length of 1 character. `12[1]` in position 14 indicates that the node labeled `14:` (the `14: CURLYX[0] {1,32767}`) begins at character position 12 in the pre-compiled form of the regex, and has a length of 1 character—that is, it corresponds to the `+` symbol in the precompiled regex.

`0[0]` items indicate that there is no corresponding node.

13.5.3 Run-time Output

First of all, when doing a match, one may get no run-time output even if debugging is enabled. This means that the regex engine was never entered and that all of the job was therefore done by the optimizer.

If the regex engine was entered, the output may look like this:

```
Matching '[bc]d(ef*g)+h[ij]k$' against 'abcdefg__gh__'
  Setting an EVAL scope, savestack=3
   2 <ab> <cdefg__gh_>    |  1: ANYOF
```

```
 3 <abc> <defg__gh_>    | 11: EXACT <d>
 4 <abcd> <efg__gh_>    | 13: CURLYX {1,32767}
 4 <abcd> <efg__gh_>    | 26:   WHILEM
                            0 out of 1..32767  cc=effff31c
 4 <abcd> <efg__gh_>    | 15:     OPEN1
 4 <abcd> <efg__gh_>    | 17:     EXACT <e>
 5 <abcde> <fg__gh_>    | 19:     STAR
                         EXACT <f> can match 1 times out of 32767...
Setting an EVAL scope, savestack=3
 6 <bcdef> <g__gh__>    | 22:       EXACT <g>
 7 <bcdefg> <__gh__>    | 24:       CLOSE1
 7 <bcdefg> <__gh__>    | 26:       WHILEM
                                1 out of 1..32767  cc=effff31c
Setting an EVAL scope, savestack=12
 7 <bcdefg> <__gh__>    | 15:         OPEN1
 7 <bcdefg> <__gh__>    | 17:         EXACT <e>
   restoring \1 to 4(4)..7
                                failed, try continuation...
 7 <bcdefg> <__gh__>    | 27:         NOTHING
 7 <bcdefg> <__gh__>    | 28:         EXACT <h>
                                failed...
                            failed...
```

The most significant information in the output is about the particular *node* of the compiled regex that is currently being tested against the target string. The format of these lines is

STRING-OFFSET <*PRE-STRING*> <*POST-STRING*> |*ID*: *TYPE*

The *TYPE* info is indented with respect to the backtracking level. Other incidental information appears interspersed within.

13.6 Debugging Perl Memory Usage

Perl is a profligate wastrel when it comes to memory use. There is a saying that to estimate memory usage of Perl, assume a reasonable algorithm for memory allocation, multiply that estimate by 10, and while you still may miss the mark, at least you won't be quite so astonished. This is not absolutely true, but may provide a good grasp of what happens.

Assume that an integer cannot take less than 20 bytes of memory, a float cannot take less than 24 bytes, a string cannot take less than 32 bytes (all these examples assume 32-bit architectures, the result are quite a bit worse on 64-bit architectures). If a variable is accessed in two of three different ways (which require an integer, a float, or a string), the memory footprint may increase yet another 20 bytes. A sloppy malloc(3) implementation can inflate these numbers dramatically.

On the opposite end of the scale, a declaration like

```
sub foo;
```

may take up to 500 bytes of memory, depending on which release of Perl you're running.

Anecdotal estimates of source-to-compiled code bloat suggest an eightfold increase. This means that the compiled form of reasonable (normally commented, properly indented etc.) code will take about eight times more space in memory than the code took on disk.

The **-DL** command-line switch is obsolete since circa Perl 5.6.0 (it was available only if Perl was built with `-DDEBUGGING`). The switch was used to track Perl's memory allocations and possible memory leaks. These days the use of malloc debugging tools like `Purify` or `valgrind` is suggested instead. See also Section 30.8.2 [perlhacktips PERL_MEM_LOG], page 597.

One way to find out how much memory is being used by Perl data structures is to install the Devel::Size module from CPAN: it gives you the minimum number of bytes required to store a particular data structure. Please be mindful of the difference between the size() and total_size().

If Perl has been compiled using Perl's malloc you can analyze Perl memory usage by setting $ENV{PERL_DEBUG_MSTATS}.

13.6.1 Using `$ENV{PERL_DEBUG_MSTATS}`

If your perl is using Perl's malloc() and was compiled with the necessary switches (this is the default), then it will print memory usage statistics after compiling your code when `$ENV{PERL_DEBUG_MSTATS} > 1`, and before termination of the program when `$ENV{PERL_DEBUG_MSTATS} >= 1`. The report format is similar to the following example:

```
$ PERL_DEBUG_MSTATS=2 perl -e "require Carp"
Memory allocation statistics after compilation: (buckets 4(4)..8188(8192)
   14216 free:   130   117    28     7     9   0   2     2   1 0 0
              437    61    36     0     5
   60924 used:   125   137   161    55     7   8   6    16   2 0 1
               74   109   304    84    20
Total sbrk(): 77824/21:119. Odd ends: pad+heads+chain+tail: 0+636+0+2048.
Memory allocation statistics after execution:   (buckets 4(4)..8188(8192)
   30888 free:   245    78    85    13     6   2   1     3   2 0 1
              315   162    39    42    11
  175816 used:   265   176  1112   111    26  22  11    27   2 1 1
              196   178  1066   798    39
Total sbrk(): 215040/47:145. Odd ends: pad+heads+chain+tail: 0+2192+0+6144.
```

It is possible to ask for such a statistic at arbitrary points in your execution using the mstat() function out of the standard Devel::Peek module.

Here is some explanation of that format:

`buckets SMALLEST(APPROX)..GREATEST(APPROX)`

Perl's malloc() uses bucketed allocations. Every request is rounded up to the closest bucket size available, and a bucket is taken from the pool of buckets of that size.

The line above describes the limits of buckets currently in use. Each bucket has two sizes: memory footprint and the maximal size of user data that can fit into this bucket. Suppose in the above example that the smallest bucket were size 4. The biggest bucket would have usable size 8188, and the memory footprint would be 8192.

In a Perl built for debugging, some buckets may have negative usable size. This means that these buckets cannot (and will not) be used. For larger buckets, the memory footprint may be one page greater than a power of 2. If so, the corresponding power of two is printed in the `APPROX` field above.

Free/Used

The 1 or 2 rows of numbers following that correspond to the number of buckets of each size between `SMALLEST` and `GREATEST`. In the first row, the sizes (memory footprints) of buckets are powers of two–or possibly one page greater. In the second row, if present, the memory footprints of the buckets are between the memory footprints of two buckets "above".

For example, suppose under the previous example, the memory footprints were

```
free:    8     16    32    64    128  256 512 1024 2048 4096 8192
       4     12    24    48    80
```

With a non-`DEBUGGING` perl, the buckets starting from `128` have a 4-byte overhead, and thus an 8192-long bucket may take up to 8188-byte allocations.

`Total sbrk(): SBRKed/SBRKs:CONTINUOUS`

The first two fields give the total amount of memory perl sbrk(2)ed (ess-broken? :-) and number of sbrk(2)s used. The third number is what perl thinks about continuity of returned chunks. So long as this number is positive, malloc() will assume that it is probable that sbrk(2) will provide continuous memory.

Memory allocated by external libraries is not counted.

`pad: 0`

The amount of sbrk(2)ed memory needed to keep buckets aligned.

`heads: 2192`

Although memory overhead of bigger buckets is kept inside the bucket, for smaller buckets, it is kept in separate areas. This field gives the total size of these areas.

`chain: 0`

malloc() may want to subdivide a bigger bucket into smaller buckets. If only a part of the deceased bucket is left unsubdivided, the rest is kept as an element of a linked list. This field gives the total size of these chunks.

`tail: 6144`

To minimize the number of sbrk(2)s, malloc() asks for more memory. This field gives the size of the yet unused part, which is sbrk(2)ed, but never touched.

13.7 SEE ALSO

Section 15.1 [perldebug NAME], page 120, Section 28.1 [perlguts NAME], page 512, Section 69.1 [perlrun NAME], page 1176 `re`, and `Devel-DProf`.

14 perldebtut

14.1 NAME

perldebtut - Perl debugging tutorial

14.2 DESCRIPTION

A (very) lightweight introduction in the use of the perl debugger, and a pointer to existing, deeper sources of information on the subject of debugging perl programs.

There's an extraordinary number of people out there who don't appear to know anything about using the perl debugger, though they use the language every day. This is for them.

14.3 use strict

First of all, there's a few things you can do to make your life a lot more straightforward when it comes to debugging perl programs, without using the debugger at all. To demonstrate, here's a simple script, named "hello", with a problem:

```
#!/usr/bin/perl

$var1 = 'Hello World'; # always wanted to do that :-)
$var2 = "$varl\n";

print $var2;
exit;
```

While this compiles and runs happily, it probably won't do what's expected, namely it doesn't print "Hello World\n" at all; It will on the other hand do exactly what it was told to do, computers being a bit that way inclined. That is, it will print out a newline character, and you'll get what looks like a blank line. It looks like there's 2 variables when (because of the typo) there's really 3:

```
$var1 = 'Hello World';
$varl = undef;
$var2 = "\n";
```

To catch this kind of problem, we can force each variable to be declared before use by pulling in the strict module, by putting 'use strict;' after the first line of the script.

Now when you run it, perl complains about the 3 undeclared variables and we get four error messages because one variable is referenced twice:

```
Global symbol "$var1" requires explicit package name at ./t1 line 4.
Global symbol "$var2" requires explicit package name at ./t1 line 5.
Global symbol "$varl" requires explicit package name at ./t1 line 5.
Global symbol "$var2" requires explicit package name at ./t1 line 7.
Execution of ./hello aborted due to compilation errors.
```

Luvverly! and to fix this we declare all variables explicitly and now our script looks like this:

```
#!/usr/bin/perl
use strict;

my $var1 = ’Hello World’;
my $varl = undef;
my $var2 = "$varl\n";

print $var2;
exit;
```

We then do (always a good idea) a syntax check before we try to run it again:

```
> perl -c hello
hello syntax OK
```

And now when we run it, we get "\n" still, but at least we know why. Just getting this script to compile has exposed the ’$varl’ (with the letter ’l’) variable, and simply changing $varl to $var1 solves the problem.

14.4 Looking at data and -w and v

Ok, but how about when you want to really see your data, what’s in that dynamic variable, just before using it?

```
#!/usr/bin/perl
use strict;

my $key = ’welcome’;
my %data = (
        ’this’ => qw(that),
        ’tom’ => qw(and jerry),
        ’welcome’ => q(Hello World),
        ’zip’ => q(welcome),
);
my @data = keys %data;

print "$data{$key}\n";
exit;
```

Looks OK, after it’s been through the syntax check (perl -c scriptname), we run it and all we get is a blank line again! Hmmmm.

One common debugging approach here, would be to liberally sprinkle a few print statements, to add a check just before we print out our data, and another just after:

```
print "All OK\n" if grep($key, keys %data);
print "$data{$key}\n";
print "done: ’$data{$key}’\n";
```

And try again:

```
> perl data
All OK
```

```
done: ''
```

After much staring at the same piece of code and not seeing the wood for the trees for some time, we get a cup of coffee and try another approach. That is, we bring in the cavalry by giving perl the '**-d**' switch on the command line:

```
> perl -d data
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(./data:4):     my $key = 'welcome';
```

Now, what we've done here is to launch the built-in perl debugger on our script. It's stopped at the first line of executable code and is waiting for input.

Before we go any further, you'll want to know how to quit the debugger: use just the letter '**q**', not the words 'quit' or 'exit':

```
DB<1> q
>
```

That's it, you're back on home turf again.

14.5 help

Fire the debugger up again on your script and we'll look at the help menu. There's a couple of ways of calling help: a simple '**h**' will get the summary help list, '**|h**' (pipe-h) will pipe the help through your pager (which is (probably 'more' or 'less'), and finally, '**h h**' (h-space-h) will give you the entire help screen. Here is the summary page:

D**1**h

```
List/search source lines:               Control script execution:
  l [ln|sub]  List source code            T           Stack trace
  - or .      List previous/current line  s [expr]    Single step [in expr]
  v [line]    View around line            n [expr]    Next, steps over subs
  f filename  View source in file         <CR/Enter>  Repeat last n or s
  /pattern/ ?patt?   Search forw/backw    r           Return from subroutine
  M           Show module versions        c [ln|sub]  Continue until position
Debugger controls:                        L           List break/watch/actions
  o [...]     Set debugger options        t [expr]    Toggle trace [trace expr]
  <[<]|{[{]|>[>] [cmd] Do pre/post-prompt b [ln|event|sub] [cnd] Set breakpoint
  ! [N|pat]   Redo a previous command     B ln|*      Delete a/all breakpoints
  H [-num]    Display last num commands   a [ln] cmd  Do cmd before line
  = [a val]   Define/list an alias        A ln|*      Delete a/all actions
  h [db_cmd]  Get help on command         w expr      Add a watch expression
  h h         Complete help page          W expr|*    Delete a/all watch exprs
  |[|]db_cmd  Send output to pager        ![!] syscmd Run cmd in a subprocess
  q or ^D     Quit                        R           Attempt a restart
```

```
 Data Examination:     expr     Execute perl code, also see: s,n,t expr
  x|m expr       Evals expr in list context, dumps the result or lists methods.
  p expr         Print expression (uses script's current package).
  S [[!]pat]     List subroutine names [not] matching pattern
  V [Pk [Vars]]  List Variables in Package.  Vars can be ~pattern or !pattern.
  X [Vars]       Same as "V current_package [Vars]".
  y [n [Vars]]   List lexicals in higher scope <n>.  Vars same as V.
 For more help, type h cmd_letter, or run man perldebug for all docs.
```

More confusing options than you can shake a big stick at! It's not as bad as it looks and it's very useful to know more about all of it, and fun too!

There's a couple of useful ones to know about straight away. You wouldn't think we're using any libraries at all at the moment, but '**M**' will show which modules are currently loaded, and their version number, while '**m**' will show the methods, and '**S**' shows all subroutines (by pattern) as shown below. '**V**' and '**X**' show variables in the program by package scope and can be constrained by pattern.

```
DB<2>S str
dumpvar::stringify
strict::bits
strict::import
strict::unimport
```

Using 'X' and cousins requires you not to use the type identifiers ($@%), just the 'name':

```
DM<3>X ~err
FileHandle(stderr) => fileno(2)
```

Remember we're in our tiny program with a problem, we should have a look at where we are, and what our data looks like. First of all let's view some code at our present position (the first line of code in this case), via '**v**':

```
DB<4> v
1       #!/usr/bin/perl
2:      use strict;
3
4==>    my $key = 'welcome';
5:      my %data = (
6               'this' => qw(that),
7               'tom' => qw(and jerry),
8               'welcome' => q(Hello World),
9               'zip' => q(welcome),
10      );
```

At line number 4 is a helpful pointer, that tells you where you are now. To see more code, type 'v' again:

```
DB<4> v
8               'welcome' => q(Hello World),
9               'zip' => q(welcome),
10      );
11:     my @data = keys %data;
12:     print "All OK\n" if grep($key, keys %data);
```

```
13:     print "$data{$key}\n";
14:     print "done: ’$data{$key}’\n";
15:     exit;
```

And if you wanted to list line 5 again, type ’l 5’, (note the space):

```
DB<4> l 5
5:      my %data = (
```

In this case, there’s not much to see, but of course normally there’s pages of stuff to wade through, and ’l’ can be very useful. To reset your view to the line we’re about to execute, type a lone period ’.’:

```
DB<5> .
main::(./data_a:4):     my $key = ’welcome’;
```

The line shown is the one that is about to be executed **next**, it hasn’t happened yet. So while we can print a variable with the letter ’**p**’, at this point all we’d get is an empty (undefined) value back. What we need to do is to step through the next executable statement with an ’**s**’:

```
DB<6> s
main::(./data_a:5):     my %data = (
main::(./data_a:6):             ’this’ => qw(that),
main::(./data_a:7):             ’tom’ => qw(and jerry),
main::(./data_a:8):             ’welcome’ => q(Hello World),
main::(./data_a:9):             ’zip’ => q(welcome),
main::(./data_a:10):    );
```

Now we can have a look at that first ($key) variable:

```
DB<7> p $key
welcome
```

line 13 is where the action is, so let’s continue down to there via the letter ’**c**’, which by the way, inserts a ’one-time-only’ breakpoint at the given line or sub routine:

```
DB<8> c 13
All OK
main::(./data_a:13):    print "$data{$key}\n";
```

We’ve gone past our check (where ’All OK’ was printed) and have stopped just before the meat of our task. We could try to print out a couple of variables to see what is happening:

```
DB<9> p $data{$key}
```

Not much in there, lets have a look at our hash:

```
DB<10> p %data
Hello Worldziptomandwelcomejerrywelcomethisthat

DB<11> p keys %data
Hello Worldtomwelcomejerrythis
```

Well, this isn’t very easy to read, and using the helpful manual (**h h**), the ’**x**’ command looks promising:

```
DB<12> x %data
0  ’Hello World’
```

```
1  'zip'
2  'tom'
3  'and'
4  'welcome'
5  undef
6  'jerry'
7  'welcome'
8  'this'
9  'that'
```

That's not much help, a couple of welcomes in there, but no indication of which are keys, and which are values, it's just a listed array dump and, in this case, not particularly helpful. The trick here, is to use a **reference** to the data structure:

```
DB<13> x \%data
0  HASH(0x8194bc4)
   'Hello World' => 'zip'
   'jerry' => 'welcome'
   'this' => 'that'
   'tom' => 'and'
   'welcome' => undef
```

The reference is truly dumped and we can finally see what we're dealing with. Our quoting was perfectly valid but wrong for our purposes, with 'and jerry' being treated as 2 separate words rather than a phrase, thus throwing the evenly paired hash structure out of alignment.

The '**-w**' switch would have told us about this, had we used it at the start, and saved us a lot of trouble:

```
> perl -w data
Odd number of elements in hash assignment at ./data line 5.
```

We fix our quoting: 'tom' => q(and jerry), and run it again, this time we get our expected output:

```
> perl -w data
Hello World
```

While we're here, take a closer look at the '**x**' command, it's really useful and will merrily dump out nested references, complete objects, partial objects - just about whatever you throw at it:

Let's make a quick object and x-plode it, first we'll start the debugger: it wants some form of input from STDIN, so we give it something non-committal, a zero:

```
> perl -de 0
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(-e:1):   0
```

Now build an on-the-fly object over a couple of lines (note the backslash):

```
DB<1> $obj = bless({'unique_id'=>'123', 'attr'=> \
cont:   {'col' => 'black', 'things' => [qw(this that etc)]}}, 'MY_class')
```

And let's have a look at it:

```
DB<2> x $obj
0  MY_class=HASH(0x828ad98)
        'attr' => HASH(0x828ad68)
'col' => 'black'
'things' => ARRAY(0x828abb8)
        0  'this'
        1  'that'
        2  'etc'
        'unique_id' => 123
DB<3>
```

Useful, huh? You can eval nearly anything in there, and experiment with bits of code or regexes until the cows come home:

```
DB<3> @data = qw(this that the other atheism leather theory scythe)

DB<4> p 'saw -> '.($cnt += map { print "\t:\t$_\n" } grep(/the/, sort @data))
atheism
leather
other
scythe
the
theory
saw -> 6
```

If you want to see the command History, type an '**H**':

```
DB<5> H
4: p 'saw -> '.($cnt += map { print "\t:\t$_\n" } grep(/the/, sort @data))
3: @data = qw(this that the other atheism leather theory scythe)
2: x $obj
1: $obj = bless({'unique_id'=>'123', 'attr'=>
{'col' => 'black', 'things' => [qw(this that etc)]}}, 'MY_class')
DB<5>
```

And if you want to repeat any previous command, use the exclamation: '**!**':

```
DB<5> !4
p 'saw -> '.($cnt += map { print "$_\n" } grep(/the/, sort @data))
atheism
leather
other
scythe
the
theory
saw -> 12
```

For more on references see Section 62.1 [perlref NAME], page 1077 and Section 63.1 [perlreftut NAME], page 1092

14.6 Stepping through code

Here’s a simple program which converts between Celsius and Fahrenheit, it too has a problem:

```
#!/usr/bin/perl -w
use strict;

my $arg = $ARGV[0] || ’-c20’;

if ($arg =˜ /ˆ\-(c|f)((\-|\+)*\d+(\.\d+)*)$/) {
        my ($deg, $num) = ($1, $2);
        my ($in, $out) = ($num, $num);
        if ($deg eq ’c’) {
                $deg = ’f’;
                $out = &c2f($num);
        } else {
                $deg = ’c’;
                $out = &f2c($num);
        }
        $out = sprintf(’%0.2f’, $out);
        $out =˜ s/ˆ((\-|\+)*\d+)\.0+$/$1/;
        print "$out $deg\n";
} else {
        print "Usage: $0 -[c|f] num\n";
}
exit;

sub f2c {
        my $f = shift;
        my $c = 5 * $f - 32 / 9;
        return $c;
}

sub c2f {
        my $c = shift;
        my $f = 9 * $c / 5 + 32;
        return $f;
}
```

For some reason, the Fahrenheit to Celsius conversion fails to return the expected output. This is what it does:

```
> temp -c0.72
33.30 f

> temp -f33.3
```

```
162.94 c
```

Not very consistent! We'll set a breakpoint in the code manually and run it under the debugger to see what's going on. A breakpoint is a flag, to which the debugger will run without interruption, when it reaches the breakpoint, it will stop execution and offer a prompt for further interaction. In normal use, these debugger commands are completely ignored, and they are safe - if a little messy, to leave in production code.

```
my ($in, $out) = ($num, $num);
$DB::single=2; # insert at line 9!
if ($deg eq 'c')
        ...

> perl -d temp -f33.3
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(temp:4): my $arg = $ARGV[0] || '-c100';
```

We'll simply continue down to our pre-set breakpoint with a '**c**':

```
DB<1> c
main::(temp:10):                if ($deg eq 'c') {
```

Followed by a view command to see where we are:

```
DB<1> v
7:              my ($deg, $num) = ($1, $2);
8:              my ($in, $out) = ($num, $num);
9:              $DB::single=2;
10==>           if ($deg eq 'c') {
11:                     $deg = 'f';
12:                     $out = &c2f($num);
13              } else {
14:                     $deg = 'c';
15:                     $out = &f2c($num);
16              }
```

And a print to show what values we're currently using:

```
DB<1> p $deg, $num
f33.3
```

We can put another break point on any line beginning with a colon, we'll use line 17 as that's just as we come out of the subroutine, and we'd like to pause there later on:

```
DB<2> b 17
```

There's no feedback from this, but you can see what breakpoints are set by using the list 'L' command:

```
DB<3> L
```

```
temp:
        17:            print "$out $deg\n";
        break if (1)
```

Note that to delete a breakpoint you use 'B'.

Now we'll continue down into our subroutine, this time rather than by line number, we'll use the subroutine name, followed by the now familiar 'v':

```
DB<3> c f2c
main::f2c(temp:30):             my $f = shift;

DB<4> v
24:     exit;
25
26      sub f2c {
27==>           my $f = shift;
28:             my $c = 5 * $f - 32 / 9;
29:             return $c;
30      }
31
32      sub c2f {
33:             my $c = shift;
```

Note that if there was a subroutine call between us and line 29, and we wanted to **single-step** through it, we could use the '**s**' command, and to step over it we would use '**n**' which would execute the sub, but not descend into it for inspection. In this case though, we simply continue down to line 29:

```
DB<4> c 29
main::f2c(temp:29):             return $c;
```

And have a look at the return value:

```
DB<5> p $c
162.944444444444
```

This is not the right answer at all, but the sum looks correct. I wonder if it's anything to do with operator precedence? We'll try a couple of other possibilities with our sum:

```
DB<6> p (5 * $f - 32 / 9)
162.944444444444

DB<7> p 5 * $f - (32 / 9)
162.944444444444

DB<8> p (5 * $f) - 32 / 9
162.944444444444

DB<9> p 5 * ($f - 32) / 9
0.722222222222221
```

:-) that's more like it! Ok, now we can set our return variable and we'll return out of the sub with an 'r':

```
DB<10> $c = 5 * ($f - 32) / 9

DB<11> r
scalar context return from main::f2c: 0.722222222222221
```

Looks good, let's just continue off the end of the script:

```
DB<12> c
0.72 c
Debugged program terminated.  Use q to quit or R to restart,
use O inhibit_exit to avoid stopping after program termination,
h q, h R or h O to get additional info.
```

A quick fix to the offending line (insert the missing parentheses) in the actual program and we're finished.

14.7 Placeholder for a, w, t, T

Actions, watch variables, stack traces etc.: on the TODO list.

```
a

w

t

T
```

14.8 REGULAR EXPRESSIONS

Ever wanted to know what a regex looked like? You'll need perl compiled with the DEBUGGING flag for this one:

```
        > perl -Dr -e '/^pe(a)*rl$/i'
        Compiling REx `^pe(a)*rl$'
        size 17 first at 2
        rarest char
         at 0
           1: BOL(2)
           2: EXACTF <pe>(4)
           4: CURLYN[1] {0,32767}(14)
           6:   NOTHING(8)
           8:   EXACTF <a>(0)
          12:   WHILEM(0)
          13: NOTHING(14)
          14: EXACTF <rl>(16)
          16: EOL(17)
          17: END(0)
        floating `'$ at 4..2147483647 (checking floating) stclass `EXACTF <pe>'
anchored(BOL) minlen 4
        Omitting $` $& $' support.
```

```
EXECUTING...

Freeing REx: `^pe(a)*rl$'
```

Did you really want to know? :-) For more gory details on getting regular expressions to work, have a look at Section 58.1 [perlre NAME], page 989, Section 68.1 [perlretut NAME], page 1131, and to decode the mysterious labels (BOL and CURLYN, etc. above), see Section 13.1 [perldebguts NAME], page 90.

14.9 OUTPUT TIPS

To get all the output from your error log, and not miss any messages via helpful operating system buffering, insert a line like this, at the start of your script:

```
$|=1;
```

To watch the tail of a dynamically growing logfile, (from the command line):

```
tail -f $error_log
```

Wrapping all die calls in a handler routine can be useful to see how, and from where, they're being called, Section 86.1 [perlvar NAME], page 1375 has more information:

```
BEGIN { $SIG{__DIE__} = sub { require Carp; Carp::confess(@_) } }
```

Various useful techniques for the redirection of STDOUT and STDERR filehandles are explained in Section 49.1 [perlopentut NAME], page 852 and `perlfaq8`.

14.10 CGI

Just a quick hint here for all those CGI programmers who can't figure out how on earth to get past that 'waiting for input' prompt, when running their CGI script from the command-line, try something like this:

```
> perl -d my_cgi.pl -nodebug
```

Of course `CGI` and `perlfaq9` will tell you more.

14.11 GUIs

The command line interface is tightly integrated with an **emacs** extension and there's a **vi** interface too.

You don't have to do this all on the command line, though, there are a few GUI options out there. The nice thing about these is you can wave a mouse over a variable and a dump of its data will appear in an appropriate window, or in a popup balloon, no more tiresome typing of 'x $varname' :-)

In particular have a hunt around for the following:

ptkdb perlTK based wrapper for the built-in debugger

ddd data display debugger

PerlDevKit and **PerlBuilder** are NT specific

NB. (more info on these and others would be appreciated).

14.12 SUMMARY

We've seen how to encourage good coding practices with **use strict** and **-w**. We can run the perl debugger **perl -d scriptname** to inspect your data from within the perl debugger with the **p** and **x** commands. You can walk through your code, set breakpoints with **b** and step through that code with **s** or **n**, continue with **c** and return from a sub with **r**. Fairly intuitive stuff when you get down to it.

There is of course lots more to find out about, this has just scratched the surface. The best way to learn more is to use perldoc to find out more about the language, to read the on-line help (Section 15.1 [perldebug NAME], page 120 is probably the next place to go), and of course, experiment.

14.13 SEE ALSO

Section 15.1 [perldebug NAME], page 120, Section 13.1 [perldebguts NAME], page 90, Section 16.1 [perldiag NAME], page 137, Section 69.1 [perlrun NAME], page 1176

14.14 AUTHOR

Richard Foley <richard.foley@rfi.net> Copyright (c) 2000

14.15 CONTRIBUTORS

Various people have made helpful suggestions and contributions, in particular:

Ronald J Kimball <rjk@linguist.dartmouth.edu>

Hugo van der Sanden <hv@crypt0.demon.co.uk>

Peter Scott <Peter@PSDT.com>

15 perldebug

15.1 NAME

perldebug - Perl debugging

15.2 DESCRIPTION

First of all, have you tried using the **-w** switch?

If you're new to the Perl debugger, you may prefer to read Section 14.1 [perldebtut NAME], page 107, which is a tutorial introduction to the debugger.

15.3 The Perl Debugger

If you invoke Perl with the **-d** switch, your script runs under the Perl source debugger. This works like an interactive Perl environment, prompting for debugger commands that let you examine source code, set breakpoints, get stack backtraces, change the values of variables, etc. This is so convenient that you often fire up the debugger all by itself just to test out Perl constructs interactively to see what they do. For example:

```
    $ perl -d -e 42
```

In Perl, the debugger is not a separate program the way it usually is in the typical compiled environment. Instead, the **-d** flag tells the compiler to insert source information into the parse trees it's about to hand off to the interpreter. That means your code must first compile correctly for the debugger to work on it. Then when the interpreter starts up, it preloads a special Perl library file containing the debugger.

The program will halt *right before* the first run-time executable statement (but see below regarding compile-time statements) and ask you to enter a debugger command. Contrary to popular expectations, whenever the debugger halts and shows you a line of code, it always displays the line it's *about* to execute, rather than the one it has just executed.

Any command not recognized by the debugger is directly executed (`eval`'d) as Perl code in the current package. (The debugger uses the DB package for keeping its own state information.)

Note that the said `eval` is bound by an implicit scope. As a result any newly introduced lexical variable or any modified capture buffer content is lost after the eval. The debugger is a nice environment to learn Perl, but if you interactively experiment using material which should be in the same scope, stuff it in one line.

For any text entered at the debugger prompt, leading and trailing whitespace is first stripped before further processing. If a debugger command coincides with some function in your own program, merely precede the function with something that doesn't look like a debugger command, such as a leading `;` or perhaps a `+`, or by wrapping it with parentheses or braces.

15.3.1 Calling the Debugger

There are several ways to call the debugger:

perl -d program_name

On the given program identified by `program_name`.

perl -d -e 0

Interactively supply an arbitrary `expression` using `-e`.

perl -d:ptkdb program_name

Debug a given program via the `Devel::ptkdb` GUI.

perl -dt threaded_program_name

Debug a given program using threads (experimental).

15.3.2 Debugger Commands

The interactive debugger understands the following commands:

h

Prints out a summary help message

h [command]

Prints out a help message for the given debugger command.

h h

The special argument of `h h` produces the entire help page, which is quite long.

If the output of the `h h` command (or any command, for that matter) scrolls past your screen, precede the command with a leading pipe symbol so that it's run through your pager, as in

```
DB> |h h
```

You may change the pager which is used via `o pager=...` command.

p expr

Same as `print {$DB::OUT} expr` in the current package. In particular, because this is just Perl's own `print` function, this means that nested data structures and objects are not dumped, unlike with the `x` command.

The `DB::OUT` filehandle is opened to `/dev/tty`, regardless of where STDOUT may be redirected to.

x [maxdepth] expr

Evaluates its expression in list context and dumps out the result in a pretty-printed fashion. Nested data structures are printed out recursively, unlike the real `print` function in Perl. When dumping hashes, you'll probably prefer 'x \%h' rather than 'x %h'. See `Dumpvalue` if you'd like to do this yourself.

The output format is governed by multiple options described under Section 15.3.3 [Configurable Options], page 128.

If the `maxdepth` is included, it must be a numeral *N*; the value is dumped only *N* levels deep, as if the `dumpDepth` option had been temporarily set to *N*.

V [pkg [vars]]

Display all (or some) variables in package (defaulting to `main`) using a data pretty-printer (hashes show their keys and values so you see what's what, control characters are made printable, etc.). Make sure you don't put the type specifier (like `$`) there, just the symbol names, like this:

```
V DB filename line
```

Use `~pattern` and `!pattern` for positive and negative regexes.

This is similar to calling the `x` command on each applicable var.

X [vars]

Same as `V currentpackage [vars]`.

y [level [vars]]

Display all (or some) lexical variables (mnemonic: `mY` variables) in the current scope or *level* scopes higher. You can limit the variables that you see with *vars* which works exactly as it does for the `V` and `X` commands. Requires the `PadWalker` module version 0.08 or higher; will warn if this isn't installed. Output is pretty-printed in the same style as for `V` and the format is controlled by the same options.

T

Produce a stack backtrace. See below for details on its output.

s [expr]

Single step. Executes until the beginning of another statement, descending into subroutine calls. If an expression is supplied that includes function calls, it too will be single-stepped.

n [expr]

Next. Executes over subroutine calls, until the beginning of the next statement. If an expression is supplied that includes function calls, those functions will be executed with stops before each statement.

r

Continue until the return from the current subroutine. Dump the return value if the `PrintRet` option is set (default).

<CR>

Repeat last `n` or `s` command.

c [line|sub]

Continue, optionally inserting a one-time-only breakpoint at the specified line or subroutine.

l

List next window of lines.

l min+incr

List `incr+1` lines starting at `min`.

l min-max

List lines `min` through `max`. `l -` is synonymous to `-`.

l line

List a single line.

l subname

List first window of lines from subroutine. *subname* may be a variable that contains a code reference.

-

List previous window of lines.

v [line]

View a few lines of code around the current line.

.

Return the internal debugger pointer to the line last executed, and print out that line.

f filename

Switch to viewing a different file or `eval` statement. If *filename* is not a full pathname found in the values of %INC, it is considered a regex.

`eval`ed strings (when accessible) are considered to be filenames: `f (eval 7)` and `f eval 7\b` access the body of the 7th `eval`ed string (in the order of execution). The bodies of the currently executed `eval` and of `eval`ed strings that define subroutines are saved and thus accessible.

/pattern/

Search forwards for pattern (a Perl regex); final / is optional. The search is case-insensitive by default.

?pattern?

Search backwards for pattern; final ? is optional. The search is case-insensitive by default.

L [abw]

List (default all) actions, breakpoints and watch expressions

S [[!]regex]

List subroutine names [not] matching the regex.

t [n]

Toggle trace mode (see also the `AutoTrace` option). Optional argument is the maximum number of levels to trace below the current one; anything deeper than that will be silent.

t [n] expr

Trace through execution of `expr`. Optional first argument is the maximum number of levels to trace below the current one; anything deeper than that will be silent. See Section 13.4 [perldebguts Frame Listing Output Examples], page 92 for examples.

b

Sets breakpoint on current line

b [line] [condition]

Set a breakpoint before the given line. If a condition is specified, it's evaluated each time the statement is reached: a breakpoint is taken only if the condition is true. Breakpoints may only be set on lines that begin an executable statement. Conditions don't use `if`:

```
    b 237 $x > 30
    b 237 ++$count237 < 11
    b 33 /pattern/i
```

If the line number is `.`, sets a breakpoint on the current line:

```
    b . $n > 100
```

b [file]:[line] [condition]

Set a breakpoint before the given line in a (possibly different) file. If a condition is specified, it's evaluated each time the statement is reached: a breakpoint is taken only if the condition is true. Breakpoints may only be set on lines that begin an executable statement. Conditions don't use `if`:

```
    b lib/MyModule.pm:237 $x > 30
    b /usr/lib/perl5/site_perl/CGI.pm:100 ++$count100 < 11
```

b subname [condition]

Set a breakpoint before the first line of the named subroutine. *subname* may be a variable containing a code reference (in this case *condition* is not supported).

b postpone subname [condition]

Set a breakpoint at first line of subroutine after it is compiled.

b load filename

Set a breakpoint before the first executed line of the *filename*, which should be a full pathname found amongst the %INC values.

b compile subname

Sets a breakpoint before the first statement executed after the specified subroutine is compiled.

B line

Delete a breakpoint from the specified *line*.

B *

Delete all installed breakpoints.

disable [file]:[line]

Disable the breakpoint so it won't stop the execution of the program. Breakpoints are enabled by default and can be re-enabled using the `enable` command.

disable [line]

Disable the breakpoint so it won't stop the execution of the program. Breakpoints are enabled by default and can be re-enabled using the `enable` command.

This is done for a breakpoint in the current file.

enable [file]:[line]

Enable the breakpoint so it will stop the execution of the program.

enable [line]

Enable the breakpoint so it will stop the execution of the program.

This is done for a breakpoint in the current file.

a [line] command

Set an action to be done before the line is executed. If *line* is omitted, set an action on the line about to be executed. The sequence of steps taken by the debugger is

```
  1. check for a breakpoint at this line
  2. print the line if necessary (tracing)
  3. do any actions associated with that line
  4. prompt user if at a breakpoint or in single-step
  5. evaluate line
```

For example, this will print out $foo every time line 53 is passed:

```
    a 53 print "DB FOUND $foo\n"
```

A line

Delete an action from the specified line.

A *

Delete all installed actions.

w expr

Add a global watch-expression. Whenever a watched global changes the debugger will stop and display the old and new values.

W expr

Delete watch-expression

W *

Delete all watch-expressions.

o

Display all options.

o booloption ...

Set each listed Boolean option to the value `1`.

o anyoption? ...

Print out the value of one or more options.

o option=value ...

Set the value of one or more options. If the value has internal whitespace, it should be quoted. For example, you could set `o pager="less -MQeicsNfr"` to call **less** with those specific options. You may use either single or double quotes, but if you do, you must escape any embedded instances of same sort of quote you began with, as well as any escaping any escapes that immediately precede that quote but which are not meant to escape the quote itself. In other words, you follow single-quoting rules irrespective of the quote; eg: `o option='this isn\'t bad'` or `o option="She said, \"Isn't it?\""`.

For historical reasons, the `=value` is optional, but defaults to 1 only where it is safe to do so–that is, mostly for Boolean options. It is always better to assign a specific value using `=`. The `option` can be abbreviated, but for clarity probably should not be. Several options can be set together. See Section 15.3.3 [Configurable Options], page 128 for a list of these.

`<` ?

List out all pre-prompt Perl command actions.

`<` [command]

Set an action (Perl command) to happen before every debugger prompt. A multi-line command may be entered by backslashing the newlines.

`<` *

Delete all pre-prompt Perl command actions.

`<<` command

Add an action (Perl command) to happen before every debugger prompt. A multi-line command may be entered by backwhacking the newlines.

`>` ? `>>`

List out post-prompt Perl command actions.

`>` command `>>`

Set an action (Perl command) to happen after the prompt when you've just given a command to return to executing the script. A multi-line command may be entered by backslashing the newlines (we bet you couldn't have guessed this by now).

`>` * `>>`

Delete all post-prompt Perl command actions.

`>>` command `>>>`

Adds an action (Perl command) to happen after the prompt when you've just given a command to return to executing the script. A multi-line command may be entered by backslashing the newlines.

{ ?

List out pre-prompt debugger commands.

{ [command]

Set an action (debugger command) to happen before every debugger prompt. A multi-line command may be entered in the customary fashion.

Because this command is in some senses new, a warning is issued if you appear to have accidentally entered a block instead. If that's what you mean to do, write it as with `;{ ... }` or even `do { ... }`.

{ *

Delete all pre-prompt debugger commands.

{{ command

Add an action (debugger command) to happen before every debugger prompt. A multi-line command may be entered, if you can guess how: see above.

! number

Redo a previous command (defaults to the previous command).

! -number

Redo number'th previous command.

! pattern

Redo last command that started with pattern. See `o recallCommand`, too.

!! cmd

Run cmd in a subprocess (reads from DB::IN, writes to DB::OUT) See `o shellBang`, also. Note that the user's current shell (well, their `$ENV{SHELL}` variable) will be used, which can interfere with proper interpretation of exit status or signal and coredump information.

source file

Read and execute debugger commands from *file*. *file* may itself contain `source` commands.

H -number

Display last n commands. Only commands longer than one character are listed. If *number* is omitted, list them all.

q or ^D

Quit. ("quit" doesn't work for this, unless you've made an alias) This is the only supported way to exit the debugger, though typing `exit` twice might work.

Set the `inhibit_exit` option to 0 if you want to be able to step off the end the script. You may also need to set $finished to 0 if you want to step through global destruction.

R

Restart the debugger by `exec()`ing a new session. We try to maintain your history across this, but internal settings and command-line options may be lost.

The following setting are currently preserved: history, breakpoints, actions, debugger options, and the Perl command-line options **-w**, **-I**, and **-e**.

|dbcmd

Run the debugger command, piping DB::OUT into your current pager.

||dbcmd

Same as `|dbcmd` but DB::OUT is temporarily `select`ed as well.

= [alias value]

Define a command alias, like

```
= quit q
```

or list current aliases.

command

Execute command as a Perl statement. A trailing semicolon will be supplied. If the Perl statement would otherwise be confused for a Perl debugger, use a leading semicolon, too.

m expr

List which methods may be called on the result of the evaluated expression. The expression may evaluated to a reference to a blessed object, or to a package name.

M

Display all loaded modules and their versions.

man [manpage]

Despite its name, this calls your system's default documentation viewer on the given page, or on the viewer itself if *manpage* is omitted. If that viewer is **man**, the current `Config` information is used to invoke **man** using the proper MANPATH or **-M** *manpath* option. Failed lookups of the form `XXX` that match known manpages of the form *perlXXX* will be retried. This lets you type `man debug` or `man op` from the debugger.

On systems traditionally bereft of a usable **man** command, the debugger invokes **perldoc**. Occasionally this determination is incorrect due to recalcitrant vendors or rather more felicitously, to enterprising users. If you fall into either category, just manually set the $DB::doccmd variable to whatever viewer to view the Perl documentation on your system. This may be set in an rc file, or through direct assignment. We're still waiting for a working example of something along the lines of:

```
$DB::doccmd = 'netscape -remote http://something.here/';
```

15.3.3 Configurable Options

The debugger has numerous options settable using the `o` command, either interactively or from the environment or an rc file. (./.perldb or ~/.perldb under Unix.)

`recallCommand`, `ShellBang`

The characters used to recall a command or spawn a shell. By default, both are set to `!`, which is unfortunate.

`pager`

Program to use for output of pager-piped commands (those beginning with a `|` character.) By default, `$ENV{PAGER}` will be used. Because the debugger uses your current terminal characteristics for bold and underlining, if the chosen pager does not pass escape sequences through unchanged, the output of some debugger commands will not be readable when sent through the pager.

`tkRunning`

Run Tk while prompting (with ReadLine).

`signalLevel`, `warnLevel`, `dieLevel`

Level of verbosity. By default, the debugger leaves your exceptions and warnings alone, because altering them can break correctly running programs. It will attempt to print a message when uncaught INT, BUS, or SEGV signals arrive. (But see the mention of signals in Section 15.7 [BUGS], page 136 below.)

To disable this default safe mode, set these values to something higher than 0. At a level of 1, you get backtraces upon receiving any kind of warning (this is often annoying) or exception (this is often valuable). Unfortunately, the debugger cannot discern fatal exceptions from non-fatal ones. If `dieLevel` is even 1, then your non-fatal exceptions are also traced and unceremoniously altered if they came from `eval'ed` strings or from any kind of `eval` within modules you're attempting to load. If `dieLevel` is 2, the debugger doesn't care where they came from: It usurps your exception handler and prints out a trace, then modifies all exceptions with its own embellishments. This may perhaps be useful for some tracing purposes, but tends to hopelessly destroy any program that takes its exception handling seriously.

`AutoTrace`

Trace mode (similar to `t` command, but can be put into `PERLDB_OPTS`).

`LineInfo`

File or pipe to print line number info to. If it is a pipe (say, `|visual_perl_db`), then a short message is used. This is the mechanism used to interact with a slave editor or visual debugger, such as the special `vi` or `emacs` hooks, or the `ddd` graphical debugger.

`inhibit_exit`

If 0, allows *stepping off* the end of the script.

`PrintRet`

Print return value after `r` command if set (default).

`ornaments`

Affects screen appearance of the command line (see `Term-ReadLine`). There is currently no way to disable these, which can render some output illegible on some displays, or with some pagers. This is considered a bug.

`frame`

Affects the printing of messages upon entry and exit from subroutines. If `frame & 2` is false, messages are printed on entry only. (Printing on exit might be useful if interspersed with other messages.)

If `frame & 4`, arguments to functions are printed, plus context and caller info. If `frame & 8`, overloaded `stringify` and `tied FETCH` is enabled on the printed arguments. If `frame & 16`, the return value from the subroutine is printed.

The length at which the argument list is truncated is governed by the next option:

`maxTraceLen`

Length to truncate the argument list when the `frame` option's bit 4 is set.

`windowSize`

Change the size of code list window (default is 10 lines).

The following options affect what happens with `V`, `X`, and `x` commands:

`arrayDepth`, `hashDepth`

Print only first N elements (” for all).

`dumpDepth`

Limit recursion depth to N levels when dumping structures. Negative values are interpreted as infinity. Default: infinity.

`compactDump`, `veryCompact`

Change the style of array and hash output. If `compactDump`, short array may be printed on one line.

`globPrint`

Whether to print contents of globs.

`DumpDBFiles`

Dump arrays holding debugged files.

`DumpPackages`

Dump symbol tables of packages.

`DumpReused`

Dump contents of "reused" addresses.

`quote`, `HighBit`, `undefPrint`

Change the style of string dump. The default value for `quote` is `auto`; one can enable double-quotish or single-quotish format by setting it to `"` or `'`, respectively. By default, characters with their high bit set are printed verbatim.

`UsageOnly`

Rudimentary per-package memory usage dump. Calculates total size of strings found in variables in the package. This does not include lexicals in a module's file scope, or lost in closures.

`HistFile`

The path of the file from which the history (assuming a usable Term::ReadLine backend) will be read on the debugger's startup, and to which it will be saved on shutdown (for persistence across sessions). Similar in concept to Bash's `.bash_history` file.

`HistSize`

The count of the saved lines in the history (assuming `HistFile` above).

After the rc file is read, the debugger reads the `$ENV{PERLDB_OPTS}` environment variable and parses this as the remainder of a "O ..." line as one might enter at the debugger prompt. You may place the initialization options `TTY`, `noTTY`, `ReadLine`, and `NonStop` there.

If your rc file contains:

```
  parse_options("NonStop=1 LineInfo=db.out AutoTrace");
```

then your script will run without human intervention, putting trace information into the file *db.out*. (If you interrupt it, you'd better reset `LineInfo` to `/dev/tty` if you expect to see anything.)

`TTY`

The TTY to use for debugging I/O.

`noTTY`

If set, the debugger goes into `NonStop` mode and will not connect to a TTY. If interrupted (or if control goes to the debugger via explicit setting of $DB::signal or $DB::single from the Perl script), it connects to a TTY specified in the `TTY` option at startup, or to a tty found at runtime using the `Term::Rendezvous` module of your choice.

This module should implement a method named `new` that returns an object with two methods: `IN` and `OUT`. These should return filehandles to use for debugging input and output correspondingly. The `new` method should inspect an argument containing the value of `$ENV{PERLDB_NOTTY}` at startup, or `"$ENV{HOME}/.perldbtty$$"` otherwise. This file is not inspected for proper ownership, so security hazards are theoretically possible.

`ReadLine`

If false, readline support in the debugger is disabled in order to debug applications that themselves use ReadLine.

`NonStop`

If set, the debugger goes into non-interactive mode until interrupted, or programmatically by setting $DB::signal or $DB::single.

Here's an example of using the `$ENV{PERLDB_OPTS}` variable:

```
  $ PERLDB_OPTS="NonStop frame=2" perl -d myprogram
```

That will run the script **myprogram** without human intervention, printing out the call tree with entry and exit points. Note that `NonStop=1 frame=2` is equivalent to `N f=2`, and that originally, options could be uniquely abbreviated by the first letter (modulo the `Dump*` options). It is nevertheless recommended that you always spell them out in full for legibility and future compatibility.

Other examples include

```
  $ PERLDB_OPTS="NonStop LineInfo=listing frame=2" perl -d myprogram
```

which runs script non-interactively, printing info on each entry into a subroutine and each executed line into the file named `listing`. (If you interrupt it, you would better reset `LineInfo` to something "interactive"!)

Other examples include (using standard shell syntax to show environment variable settings):

```
$ ( PERLDB_OPTS="NonStop frame=1 AutoTrace LineInfo=tperl.out"
    perl -d myprogram )
```

which may be useful for debugging a program that uses `Term::ReadLine` itself. Do not forget to detach your shell from the TTY in the window that corresponds to `/dev/ttyXX`, say, by issuing a command like

```
$ sleep 1000000
```

See Section 13.3 [perldebguts Debugger Internals], page 90 for details.

15.3.4 Debugger Input/Output

Prompt

The debugger prompt is something like

```
DB<8>
```

or even

```
DB<<17>>
```

where that number is the command number, and which you'd use to access with the built-in **csh**-like history mechanism. For example, `!17` would repeat command number 17. The depth of the angle brackets indicates the nesting depth of the debugger. You could get more than one set of brackets, for example, if you'd already at a breakpoint and then printed the result of a function call that itself has a breakpoint, or you step into an expression via `s/n/t expression` command.

Multiline commands

If you want to enter a multi-line command, such as a subroutine definition with several statements or a format, escape the newline that would normally end the debugger command with a backslash. Here's an example:

```
DB<1> for (1..4) {         \
cont:     print "ok\n";   \
cont: }
ok
ok
ok
ok
```

Note that this business of escaping a newline is specific to interactive commands typed into the debugger.

Stack backtrace

Here's an example of what a stack backtrace via `T` command might look like:

```
$ = main::infested called from file 'Ambulation.pm' line 10
@ = Ambulation::legs(1, 2, 3, 4) called from file 'camel_flea' line 7
$ = main::pests('bactrian', 4) called from file 'camel_flea' line 4
```

The left-hand character up there indicates the context in which the function was called, with `$` and `@` meaning scalar or list contexts respectively, and `.` meaning void context (which is actually a sort of scalar context). The display above says that you were in the function `main::infested` when you ran the stack dump, and that it was called in scalar context from line 10 of the file *Ambulation.pm*, but without any arguments at all, meaning it was called as `&infested`. The next stack frame shows that the function `Ambulation::legs` was called in list context from the *camel_flea* file with four arguments. The last stack frame shows that `main::pests` was called in scalar context, also from *camel_flea*, but from line 4.

If you execute the `T` command from inside an active `use` statement, the backtrace will contain both a `require` frame and an `eval` frame.

Line Listing Format

This shows the sorts of output the `l` command can produce:

```
 DB<<13>> l
101:      @i{@i} = ();
102:b     @isa{@i,$pack} = ()
103           if(exists $i{$prevpack} || exists $isa{$pack});
104     }
105
106     next
107==>       if(exists $isa{$pack});
108
109:a    if ($extra-- > 0) {
110:         %isa = ($pack,1);
```

Breakable lines are marked with `:`. Lines with breakpoints are marked by `b` and those with actions by `a`. The line that's about to be executed is marked by `==>`.

Please be aware that code in debugger listings may not look the same as your original source code. Line directives and external source filters can alter the code before Perl sees it, causing code to move from its original positions or take on entirely different forms.

Frame listing

When the `frame` option is set, the debugger would print entered (and optionally exited) subroutines in different styles. See Section 13.1 [perldebguts NAME], page 90 for incredibly long examples of these.

15.3.5 Debugging Compile-Time Statements

If you have compile-time executable statements (such as code within BEGIN, UNITCHECK and CHECK blocks or `use` statements), these will *not* be stopped by debugger, although `require`s and INIT blocks will, and compile-time statements can be traced with the `AutoTrace` option set in `PERLDB_OPTS`). From your own Perl code, however, you can transfer control back to the debugger using the following statement, which is harmless if the debugger is not running:

```
$DB::single = 1;
```

If you set `$DB::single` to 2, it's equivalent to having just typed the `n` command, whereas a value of 1 means the `s` command. The `$DB::trace` variable should be set to 1 to simulate having typed the `t` command.

Another way to debug compile-time code is to start the debugger, set a breakpoint on the *load* of some module:

```
  DB<7> b load f:/perllib/lib/Carp.pm
Will stop on load of 'f:/perllib/lib/Carp.pm'.
```

and then restart the debugger using the `R` command (if possible). One can use `b compile subname` for the same purpose.

15.3.6 Debugger Customization

The debugger probably contains enough configuration hooks that you won't ever have to modify it yourself. You may change the behaviour of the debugger from within the debugger using its `o` command, from the command line via the `PERLDB_OPTS` environment variable, and from customization files.

You can do some customization by setting up a `.perldb` file, which contains initialization code. For instance, you could make aliases like these (the last one is one people expect to be there):

```
$DB::alias{'len'}  = 's/^len(.*)/p length($1)/';
$DB::alias{'stop'} = 's/^stop (at|in)/b/';
$DB::alias{'ps'}   = 's/^ps\b/p scalar /';
$DB::alias{'quit'} = 's/^quit(\s*)/exit/';
```

You can change options from `.perldb` by using calls like this one;

```
parse_options("NonStop=1 LineInfo=db.out AutoTrace=1 frame=2");
```

The code is executed in the package `DB`. Note that `.perldb` is processed before processing `PERLDB_OPTS`. If `.perldb` defines the subroutine `afterinit`, that function is called after debugger initialization ends. `.perldb` may be contained in the current directory, or in the home directory. Because this file is sourced in by Perl and may contain arbitrary commands, for security reasons, it must be owned by the superuser or the current user, and writable by no one but its owner.

You can mock TTY input to debugger by adding arbitrary commands to @DB::typeahead. For example, your `.perldb` file might contain:

```
sub afterinit { push @DB::typeahead, "b 4", "b 6"; }
```

Which would attempt to set breakpoints on lines 4 and 6 immediately after debugger initialization. Note that @DB::typeahead is not a supported interface and is subject to change in future releases.

If you want to modify the debugger, copy `perl5db.pl` from the Perl library to another name and hack it to your heart's content. You'll then want to set your `PERL5DB` environment variable to say something like this:

```
BEGIN { require "myperl5db.pl" }
```

As a last resort, you could also use `PERL5DB` to customize the debugger by directly setting internal variables or calling debugger functions.

Note that any variables and functions that are not documented in this document (or in Section 13.1 [perldebguts NAME], page 90) are considered for internal use only, and as such are subject to change without notice.

15.3.7 Readline Support / History in the Debugger

As shipped, the only command-line history supplied is a simplistic one that checks for leading exclamation points. However, if you install the Term::ReadKey and Term::ReadLine modules from CPAN (such as Term::ReadLine::Gnu, Term::ReadLine::Perl, ...) you will have full editing capabilities much like those GNU *readline*(3) provides. Look for these in the `modules/by-module/Term` directory on CPAN. These do not support normal **vi** command-line editing, however.

A rudimentary command-line completion is also available, including lexical variables in the current scope if the `PadWalker` module is installed.

Without Readline support you may see the symbols "^[[A", "^[[C", "^[[B", "^[[D"", "^H", ... when using the arrow keys and/or the backspace key.

15.3.8 Editor Support for Debugging

If you have the GNU's version of **emacs** installed on your system, it can interact with the Perl debugger to provide an integrated software development environment reminiscent of its interactions with C debuggers.

Recent versions of Emacs come with a start file for making **emacs** act like a syntax-directed editor that understands (some of) Perl's syntax. See `perlfaq3`.

Users of **vi** should also look into **vim** and **gvim**, the mousey and windy version, for coloring of Perl keywords.

Note that only perl can truly parse Perl, so all such CASE tools fall somewhat short of the mark, especially if you don't program your Perl as a C programmer might.

15.3.9 The Perl Profiler

If you wish to supply an alternative debugger for Perl to run, invoke your script with a colon and a package argument given to the **-d** flag. Perl's alternative debuggers include a Perl profiler, `Devel-NYTProf`, which is available separately as a CPAN distribution. To profile your Perl program in the file `mycode.pl`, just type:

```
$ perl -d:NYTProf mycode.pl
```

When the script terminates the profiler will create a database of the profile information that you can turn into reports using the profiler's tools. See <perlperf> for details.

15.4 Debugging Regular Expressions

`use re 'debug'` enables you to see the gory details of how the Perl regular expression engine works. In order to understand this typically voluminous output, one must not only have some idea about how regular expression matching works in general, but also know how Perl's regular expressions are internally compiled into an automaton. These matters are explored in some detail in Section 13.5 [perldebguts Debugging Regular Expressions], page 96.

15.5 Debugging Memory Usage

Perl contains internal support for reporting its own memory usage, but this is a fairly advanced concept that requires some understanding of how memory allocation works. See Section 13.6 [perldebguts Debugging Perl Memory Usage], page 104 for the details.

15.6 SEE ALSO

You did try the **-w** switch, didn't you?

Section 14.1 [perldebtut NAME], page 107, Section 13.1 [perldebguts NAME], page 90, `re`, `DB`, `Devel-NYTProf`, `Dumpvalue`, and Section 69.1 [perlrun NAME], page 1176.

When debugging a script that uses #! and is thus normally found in $PATH, the -S option causes perl to search $PATH for it, so you don't have to type the path or `which $scriptname`.

```
$ perl -Sd foo.pl
```

15.7 BUGS

You cannot get stack frame information or in any fashion debug functions that were not compiled by Perl, such as those from C or C++ extensions.

If you alter your @_ arguments in a subroutine (such as with `shift` or `pop`), the stack backtrace will not show the original values.

The debugger does not currently work in conjunction with the **-W** command-line switch, because it itself is not free of warnings.

If you're in a slow syscall (like `wait`ing, `accept`ing, or `read`ing from your keyboard or a socket) and haven't set up your own `$SIG{INT}` handler, then you won't be able to CTRL-C your way back to the debugger, because the debugger's own `$SIG{INT}` handler doesn't understand that it needs to raise an exception to longjmp(3) out of slow syscalls.

16 perldiag

16.1 NAME

perldiag - various Perl diagnostics

16.2 DESCRIPTION

These messages are classified as follows (listed in increasing order of desperation):

```
(W) A warning (optional).
(D) A deprecation (enabled by default).
(S) A severe warning (enabled by default).
(F) A fatal error (trappable).
(P) An internal error you should never see (trappable).
(X) A very fatal error (nontrappable).
(A) An alien error message (not generated by Perl).
```

The majority of messages from the first three classifications above (W, D & S) can be controlled using the `warnings` pragma.

If a message can be controlled by the `warnings` pragma, its warning category is included with the classification letter in the description below. E.g. `(W closed)` means a warning in the `closed` category.

Optional warnings are enabled by using the `warnings` pragma or the **-w** and **-W** switches. Warnings may be captured by setting `$SIG{__WARN__}` to a reference to a routine that will be called on each warning instead of printing it. See Section 86.1 [perlvar NAME], page 1375.

Severe warnings are always enabled, unless they are explicitly disabled with the `warnings` pragma or the **-X** switch.

Trappable errors may be trapped using the eval operator. See [perlfunc eval], page 377. In almost all cases, warnings may be selectively disabled or promoted to fatal errors using the `warnings` pragma. See `warnings`.

The messages are in alphabetical order, without regard to upper or lower-case. Some of these messages are generic. Spots that vary are denoted with a %s or other printf-style escape. These escapes are ignored by the alphabetical order, as are all characters other than letters. To look up your message, just ignore anything that is not a letter.

accept() on closed socket %s

> (W closed) You tried to do an accept on a closed socket. Did you forget to check the return value of your socket() call? See ⟨undefined⟩ [perlfunc accept], page ⟨undefined⟩.

Aliasing via reference is experimental

> (S experimental::refaliasing) This warning is emitted if you use a reference constructor on the left-hand side of an assignment to alias one variable to another. Simply suppress the warning if you want to use the feature, but know that in doing so you are taking the risk of using an experimental feature which may change or be removed in a future Perl version:

```
no warnings "experimental::refaliasing";
use feature "refaliasing";
\$x = \$y;
```

Allocation too large: %x

(X) You can't allocate more than 64K on an MS-DOS machine.

'%c' allowed only after types %s in %s

(F) The modifiers '!', '<' and '>' are allowed in pack() or unpack() only after certain types. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

Ambiguous call resolved as CORE::%s(), qualify as such or use &

(W ambiguous) A subroutine you have declared has the same name as a Perl keyword, and you have used the name without qualification for calling one or the other. Perl decided to call the builtin because the subroutine is not imported.

To force interpretation as a subroutine call, either put an ampersand before the subroutine name, or qualify the name with its package. Alternatively, you can import the subroutine (or pretend that it's imported with the `use subs` pragma).

To silently interpret it as the Perl operator, use the `CORE::` prefix on the operator (e.g. `CORE::log($x)`) or declare the subroutine to be an object method (see Section 73.3.14 [perlsub Subroutine Attributes], page 1247 or `attributes`).

Ambiguous range in transliteration operator

(F) You wrote something like `tr/a-z-0//` which doesn't mean anything at all. To include a `-` character in a transliteration, put it either first or last. (In the past, `tr/a-z-0//` was synonymous with `tr/a-y//`, which was probably not what you would have expected.)

Ambiguous use of %s resolved as %s

(S ambiguous) You said something that may not be interpreted the way you thought. Normally it's pretty easy to disambiguate it by supplying a missing quote, operator, parenthesis pair or declaration.

Ambiguous use of -%s resolved as -&%s()

(S ambiguous) You wrote something like `-foo`, which might be the string `"-foo"`, or a call to the function `foo`, negated. If you meant the string, just write `"-foo"`. If you meant the function call, write `-foo()`.

Ambiguous use of %c resolved as operator %c

(S ambiguous) `%`, `&`, and `*` are both infix operators (modulus, bitwise and, and multiplication) *and* initial special characters (denoting hashes, subroutines and typeglobs), and you said something like `*foo * foo` that might be interpreted as either of them. We assumed you meant the infix operator, but please try to make it more clear – in the example given, you might write `*foo * foo()` if you really meant to multiply a glob by the result of calling a function.

Ambiguous use of %c{%s} resolved to %c%s

(W ambiguous) You wrote something like `@{foo}`, which might be asking for the variable `@foo`, or it might be calling a function named foo, and dereferencing it as an array reference. If you wanted the variable, you can just write `@foo`. If you wanted to call the function, write `@{foo()}` ... or you could just not have a variable and a function with the same name, and save yourself a lot of trouble.

Ambiguous use of %c{%s[...]} resolved to %c%s[...]
Ambiguous use of %c{%s{...}} resolved to %c%s{...}

(W ambiguous) You wrote something like `${foo[2]}` (where foo represents the name of a Perl keyword), which might be looking for element number 2 of the array named `@foo`, in which case please write `$foo[2]`, or you might have meant to pass an anonymous arrayref to the function named foo, and then do a scalar deref on the value it returns. If you meant that, write `${foo([2])}`.

In regular expressions, the `${foo[2]}` syntax is sometimes necessary to disambiguate between array subscripts and character classes. `/$length[2345]/`, for instance, will be interpreted as `$length` followed by the character class `[2345]`. If an array subscript is what you want, you can avoid the warning by changing `/${length[2345]}/` to the unsightly `/${\$length[2345]}/`, by renaming your array to something that does not coincide with a built-in keyword, or by simply turning off warnings with `no warnings 'ambiguous';`.

'`|`' and '<' may not both be specified on command line

(F) An error peculiar to VMS. Perl does its own command line redirection, and found that STDIN was a pipe, and that you also tried to redirect STDIN using '<'. Only one STDIN stream to a customer, please.

'`|`' and '>' may not both be specified on command line

(F) An error peculiar to VMS. Perl does its own command line redirection, and thinks you tried to redirect stdout both to a file and into a pipe to another command. You need to choose one or the other, though nothing's stopping you from piping into a program or Perl script which 'splits' output into two streams, such as

```
open(OUT,">$ARGV[0]") or die "Can't write to $ARGV[0]: $!";
while (<STDIN>) {
    print;
    print OUT;
}
close OUT;
```

Applying %s to %s will act on scalar(%s)

(W misc) The pattern match (`//`), substitution (`s///`), and transliteration (`tr///`) operators work on scalar values. If you apply one of them to an array or a hash, it will convert the array or hash to a scalar value (the length of an array, or the population info of a hash) and then work on that scalar value. This is probably not what you meant to do. See ⟨undefined⟩ [perlfunc grep], page ⟨undefined⟩ and ⟨undefined⟩ [perlfunc map], page ⟨undefined⟩ for alternatives.

Arg too short for msgsnd

(F) msgsnd() requires a string at least as long as sizeof(long).

Argument "%s" isn't numeric%s

(W numeric) The indicated string was fed as an argument to an operator that expected a numeric value instead. If you're fortunate the message will identify which operator was so unfortunate.

Note that for the `Inf` and `NaN` (infinity and not-a-number) the definition of "numeric" is somewhat unusual: the strings themselves (like "Inf") are considered numeric, and anything following them is considered non-numeric.

Argument list not closed for PerlIO layer "%s"

(W layer) When pushing a layer with arguments onto the Perl I/O system you forgot the) that closes the argument list. (Layers take care of transforming data between external and internal representations.) Perl stopped parsing the layer list at this point and did not attempt to push this layer. If your program didn't explicitly request the failing operation, it may be the result of the value of the environment variable PERLIO.

Argument "%s" treated as 0 in increment (++)

(W numeric) The indicated string was fed as an argument to the `++` operator which expects either a number or a string matching `/^[a-zA-Z]*[0-9]*\z/`. See Section 48.2.4 [perlop Auto-increment and Auto-decrement], page 800 for details.

assertion botched: %s

(X) The malloc package that comes with Perl had an internal failure.

Assertion %s failed: file "%s", line %d

(X) A general assertion failed. The file in question must be examined.

Assigned value is not a reference

(F) You tried to assign something that was not a reference to an lvalue reference (e.g., `\$x = $y`). If you meant to make $x an alias to $y, use `\$x = \$y`.

Assigned value is not %s reference

(F) You tried to assign a reference to a reference constructor, but the two references were not of the same type. You cannot alias a scalar to an array, or an array to a hash; the two types must match.

```
\$x = \@y;  # error
\@x = \%y;  # error
 $y = [];
\$x = $y;   # error; did you mean \$y?
```

Assigning non-zero to $[is no longer possible

(F) When the "array_base" feature is disabled (e.g., under `use v5.16;`) the special variable `$[`, which is deprecated, is now a fixed zero value.

Assignment to both a list and a scalar

(F) If you assign to a conditional operator, the 2nd and 3rd arguments must either both be scalars or both be lists. Otherwise Perl won't know which context to supply to the right side.

`<>` at require-statement should be quotes

(F) You wrote `require <file>` when you should have written `require 'file'`.

Attempt to access disallowed key '%s' in a restricted hash

(F) The failing code has attempted to get or set a key which is not in the current set of allowed keys of a restricted hash.

Attempt to bless into a freed package

(F) You wrote `bless $foo` with one argument after somehow causing the current package to be freed. Perl cannot figure out what to do, so it throws up in hands in despair.

Attempt to bless into a reference

(F) The CLASSNAME argument to the bless() operator is expected to be the name of the package to bless the resulting object into. You've supplied instead a reference to something: perhaps you wrote

```
    bless $self, $proto;
```

when you intended

```
    bless $self, ref($proto) || $proto;
```

If you actually want to bless into the stringified version of the reference supplied, you need to stringify it yourself, for example by:

```
    bless $self, "$proto";
```

Attempt to clear deleted array

(S debugging) An array was assigned to when it was being freed. Freed values are not supposed to be visible to Perl code. This can also happen if XS code calls `av_clear` from a custom magic callback on the array.

Attempt to delete disallowed key '%s' from a restricted hash

(F) The failing code attempted to delete from a restricted hash a key which is not in its key set.

Attempt to delete readonly key '%s' from a restricted hash

(F) The failing code attempted to delete a key whose value has been declared readonly from a restricted hash.

Attempt to free non-arena SV: 0x%x

(S internal) All SV objects are supposed to be allocated from arenas that will be garbage collected on exit. An SV was discovered to be outside any of those arenas.

Attempt to free nonexistent shared string '%s'%s

(S internal) Perl maintains a reference-counted internal table of strings to optimize the storage and access of hash keys and other strings. This indicates someone tried to decrement the reference count of a string that can no longer be found in the table.

Attempt to free temp prematurely: SV 0x%x

(S debugging) Mortalized values are supposed to be freed by the free_tmps() routine. This indicates that something else is freeing the SV before the free_tmps() routine gets a chance, which means that the free_tmps() routine will be freeing an unreferenced scalar when it does try to free it.

Attempt to free unreferenced glob pointers

(S internal) The reference counts got screwed up on symbol aliases.

Attempt to free unreferenced scalar: SV 0x%x

(S internal) Perl went to decrement the reference count of a scalar to see if it would go to 0, and discovered that it had already gone to 0 earlier, and should have been freed, and in fact, probably was freed. This could indicate that SvREFCNT_dec() was called too many times, or that SvREFCNT_inc() was called too few times, or that the SV was mortalized when it shouldn't have been, or that memory has been corrupted.

Attempt to pack pointer to temporary value

(W pack) You tried to pass a temporary value (like the result of a function, or a computed expression) to the "p" pack() template. This means the result contains a pointer to a location that could become invalid anytime, even before the end of the current statement. Use literals or global values as arguments to the "p" pack() template to avoid this warning.

Attempt to reload %s aborted.

(F) You tried to load a file with `use` or `require` that failed to compile once already. Perl will not try to compile this file again unless you delete its entry from %INC. See [perlfunc require], page 437 and [perlvar %INC], page 1382.

Attempt to set length of freed array

(W misc) You tried to set the length of an array which has been freed. You can do this by storing a reference to the scalar representing the last index of an array and later assigning through that reference. For example

```
$r = do {my @a; \$#a};
$$r = 503
```

Attempt to use reference as lvalue in substr

(W substr) You supplied a reference as the first argument to substr() used as an lvalue, which is pretty strange. Perhaps you forgot to dereference it first. See ⟨undefined⟩ [perlfunc substr], page ⟨undefined⟩.

Attribute "locked" is deprecated

(D deprecated) You have used the attributes pragma to modify the "locked" attribute on a code reference. The :locked attribute is obsolete, has had no effect since 5005 threads were removed, and will be removed in a future release of Perl 5.

Attribute prototype(%s) discards earlier prototype attribute in same sub

(W misc) A sub was declared as sub foo : prototype(A) : prototype(B) {}, for example. Since each sub can only have one prototype, the earlier declaration(s) are discarded while the last one is applied.

Attribute "unique" is deprecated

(D deprecated) You have used the attributes pragma to modify the "unique" attribute on an array, hash or scalar reference. The :unique attribute has had no effect since Perl 5.8.8, and will be removed in a future release of Perl 5.

av_reify called on tied array

(S debugging) This indicates that something went wrong and Perl got *very* confused about `@_` or `@DB::args` being tied.

Bad arg length for %s, is %u, should be %d

(F) You passed a buffer of the wrong size to one of msgctl(), semctl() or shmctl(). In C parlance, the correct sizes are, respectively, sizeof(struct msqid_ds *), sizeof(struct semid_ds *), and sizeof(struct shmid_ds *).

Bad evalled substitution pattern

(F) You've used the `/e` switch to evaluate the replacement for a substitution, but perl found a syntax error in the code to evaluate, most likely an unexpected right brace '}'.

Bad filehandle: %s

(F) A symbol was passed to something wanting a filehandle, but the symbol has no filehandle associated with it. Perhaps you didn't do an open(), or did it in another package.

Bad free() ignored

(S malloc) An internal routine called free() on something that had never been malloc()ed in the first place. Mandatory, but can be disabled by setting environment variable `PERL_BADFREE` to 0.

This message can be seen quite often with DB_File on systems with "hard" dynamic linking, like `AIX` and `OS/2`. It is a bug of `Berkeley DB` which is left unnoticed if `DB` uses *forgiving* system malloc().

Bad hash

(P) One of the internal hash routines was passed a null HV pointer.

Badly placed ()'s

(A) You've accidentally run your script through **csh** instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

Bad name after %s

(F) You started to name a symbol by using a package prefix, and then didn't finish the symbol. In particular, you can't interpolate outside of quotes, so

```
    $var = 'myvar';
    $sym = mypack::$var;
```

is not the same as

```
    $var = 'myvar';
    $sym = "mypack::$var";
```

Bad plugin affecting keyword '%s'

(F) An extension using the keyword plugin mechanism violated the plugin API.

Bad realloc() ignored

(S malloc) An internal routine called realloc() on something that had never been malloc()ed in the first place. Mandatory, but can be disabled by setting the environment variable `PERL_BADFREE` to 1.

Bad symbol for array

(P) An internal request asked to add an array entry to something that wasn't a symbol table entry.

Bad symbol for dirhandle

(P) An internal request asked to add a dirhandle entry to something that wasn't a symbol table entry.

Bad symbol for filehandle

(P) An internal request asked to add a filehandle entry to something that wasn't a symbol table entry.

Bad symbol for hash

(P) An internal request asked to add a hash entry to something that wasn't a symbol table entry.

Bad symbol for scalar

(P) An internal request asked to add a scalar entry to something that wasn't a symbol table entry.

Bareword found in conditional

(W bareword) The compiler found a bareword where it expected a conditional, which often indicates that an `||` or && was parsed as part of the last argument of the previous construct, for example:

```
open FOO || die;
```

It may also indicate a misspelled constant that has been interpreted as a bareword:

```
use constant TYPO => 1;
if (TYOP) { print "foo" }
```

The `strict` pragma is useful in avoiding such errors.

Bareword "%s" not allowed while "strict subs" in use

(F) With "strict subs" in use, a bareword is only allowed as a subroutine identifier, in curly brackets or to the left of the "=>" symbol. Perhaps you need to predeclare a subroutine?

Bareword "%s" refers to nonexistent package

(W bareword) You used a qualified bareword of the form `Foo::`, but the compiler saw no other uses of that namespace before that point. Perhaps you need to predeclare a package?

BEGIN failed–compilation aborted

(F) An untrapped exception was raised while executing a BEGIN subroutine. Compilation stops immediately and the interpreter is exited.

BEGIN not safe after errors–compilation aborted

(F) Perl found a `BEGIN {}` subroutine (or a `use` directive, which implies a `BEGIN {}`) after one or more compilation errors had already occurred. Since the intended environment for the `BEGIN {}` could not be guaranteed (due to the errors), and since subsequent code likely depends on its correct operation, Perl just gave up.

\%d better written as $%d

(W syntax) Outside of patterns, backreferences live on as variables. The use of backslashes is grandfathered on the right-hand side of a substitution, but stylistically it's better to use the variable form because other Perl programmers will expect it, and it works better if there are more than 9 backreferences.

Binary number > 0b11111111111111111111111111111111 non-portable

(W portable) The binary number you specified is larger than 2**32-1 (4294967295) and therefore non-portable between systems. See Section 56.1 [perlport NAME], page 951 for more on portability concerns.

bind() on closed socket %s

(W closed) You tried to do a bind on a closed socket. Did you forget to check the return value of your socket() call? See ⟨undefined⟩ [perlfunc bind], page ⟨undefined⟩.

binmode() on closed filehandle %s

(W unopened) You tried binmode() on a filehandle that was never opened. Check your control flow and number of arguments.

Bit vector size > 32 non-portable

(W portable) Using bit vector sizes larger than 32 is non-portable.

Bizarre copy of %s

(P) Perl detected an attempt to copy an internal value that is not copiable.

Bizarre SvTYPE [%d]

(P) When starting a new thread or returning values from a thread, Perl encountered an invalid data type.

Both or neither range ends should be Unicode in regex; marked by <– HERE in m/%s/

(W regexp) (only under `use re 'strict'` or within `(?[...])`)

In a bracketed character class in a regular expression pattern, you had a range which has exactly one end of it specified using `\N{}`, and the other end is specified using a non-portable mechanism. Perl treats the range as a Unicode range, that is, all the characters in it are considered to be the Unicode characters, and which may be different code points on some platforms Perl runs on. For example, `[\N{U+06}-\x08]` is treated as if you had instead said `[\N{U+06}-\N{U+08}]`, that is it matches the characters whose code points in Unicode are 6, 7, and 8. But that `\x08` might indicate that you meant something different, so the warning gets raised.

Buffer overflow in prime_env_iter: %s

(W internal) A warning peculiar to VMS. While Perl was preparing to iterate over %ENV, it encountered a logical name or symbol definition which was too long, so it was truncated to the string shown.

Callback called exit

(F) A subroutine invoked from an external package via call_sv() exited by calling exit.

%s() called too early to check prototype

(W prototype) You've called a function that has a prototype before the parser saw a definition or declaration for it, and Perl could not check that the call conforms to the prototype. You need to either add an early prototype declaration for the subroutine in question, or move the subroutine definition ahead of the call to get proper prototype checking. Alternatively, if you are certain that you're calling the function correctly, you may put an ampersand before the name to avoid the warning. See Section 73.1 [perlsub NAME], page 1216.

Calling POSIX::%s() is deprecated

(D deprecated) You called a function whose use is deprecated. See the function's name in `POSIX` for details.

Cannot chr %f

(F) You passed an invalid number (like an infinity or not-a-number) to `chr`.

Cannot compress %f in pack

(F) You tried compressing an infinity or not-a-number as an unsigned integer with BER, which makes no sense.

Cannot compress integer in pack

(F) An argument to pack("w",...) was too large to compress. The BER compressed integer format can only be used with positive integers, and you attempted to compress a very large number (> 1e308). See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

Cannot compress negative numbers in pack

(F) An argument to pack("w",...) was negative. The BER compressed integer format can only be used with positive integers. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

Cannot convert a reference to %s to typeglob

(F) You manipulated Perl's symbol table directly, stored a reference in it, then tried to access that symbol via conventional Perl syntax. The access triggers Perl to autovivify that typeglob, but it there is no legal conversion from that type of reference to a typeglob.

Cannot copy to %s

(P) Perl detected an attempt to copy a value to an internal type that cannot be directly assigned to.

Cannot find encoding "%s"

(S io) You tried to apply an encoding that did not exist to a filehandle, either with open() or binmode().

Cannot pack %f with '%c'

(F) You tried converting an infinity or not-a-number to an integer, which makes no sense.

Cannot printf %f with '%c'

(F) You tried printing an infinity or not-a-number as a character (%c), which makes no sense. Maybe you meant '%s', or just stringifying it?

Cannot set tied @DB::args

(F) `caller` tried to set `@DB::args`, but found it tied. Tying `@DB::args` is not supported. (Before this error was added, it used to crash.)

Cannot tie unreifiable array

(P) You somehow managed to call `tie` on an array that does not keep a reference count on its arguments and cannot be made to do so. Such arrays are not even supposed to be accessible to Perl code, but are only used internally.

Can only compress unsigned integers in pack

(F) An argument to pack("w",...) was not an integer. The BER compressed integer format can only be used with positive integers, and you attempted to compress something else. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

Can't bless non-reference value

(F) Only hard references may be blessed. This is how Perl "enforces" encapsulation of objects. See Section 46.1 [perlobj NAME], page 769.

Can't "break" in a loop topicalizer

(F) You called `break`, but you're in a `foreach` block rather than a `given` block. You probably meant to use `next` or `last`.

Can't "break" outside a given block

(F) You called `break`, but you're not inside a `given` block.

Can't call method "%s" on an undefined value

(F) You used the syntax of a method call, but the slot filled by the object reference or package name contains an undefined value. Something like this will reproduce the error:

```
$BADREF = undef;
process $BADREF 1,2,3;
$BADREF->process(1,2,3);
```

Can't call method "%s" on unblessed reference

(F) A method call must know in what package it's supposed to run. It ordinarily finds this out from the object reference you supply, but you didn't supply an object reference in this case. A reference isn't an object reference until it has been blessed. See Section 46.1 [perlobj NAME], page 769.

Can't call method "%s" without a package or object reference

(F) You used the syntax of a method call, but the slot filled by the object reference or package name contains an expression that returns a defined value which is neither an object reference nor a package name. Something like this will reproduce the error:

```
$BADREF = 42;
process $BADREF 1,2,3;
$BADREF->process(1,2,3);
```

Can't call mro_isa_changed_in() on anonymous symbol table

(P) Perl got confused as to whether a hash was a plain hash or a symbol table hash when trying to update @ISA caches.

Can't call mro_method_changed_in() on anonymous symbol table

(F) An XS module tried to call `mro_method_changed_in` on a hash that was not attached to the symbol table.

Can't chdir to %s

(F) You called `perl -x/foo/bar`, but `/foo/bar` is not a directory that you can chdir to, possibly because it doesn't exist.

Can't check filesystem of script "%s" for nosuid

(P) For some reason you can't check the filesystem of the script for nosuid.

Can't coerce %s to %s in %s

(F) Certain types of SVs, in particular real symbol table entries (typeglobs), can't be forced to stop being what they are. So you can't say things like:

```
    *foo += 1;
```

You CAN say

```
    $foo = *foo;
    $foo += 1;
```

but then $foo no longer contains a glob.

Can't "continue" outside a when block

(F) You called `continue`, but you're not inside a `when` or `default` block.

Can't create pipe mailbox

(P) An error peculiar to VMS. The process is suffering from exhausted quotas or other plumbing problems.

Can't declare %s in "%s"

(F) Only scalar, array, and hash variables may be declared as "my", "our" or "state" variables. They must have ordinary identifiers as names.

Can't "default" outside a topicalizer

(F) You have used a `default` block that is neither inside a `foreach` loop nor a `given` block. (Note that this error is issued on exit from the `default` block, so you won't get the error if you use an explicit `continue`.)

Can't do inplace edit: %s is not a regular file

(S inplace) You tried to use the **-i** switch on a special file, such as a file in /dev, a FIFO or an uneditable directory. The file was ignored.

Can't do inplace edit on %s: %s

(S inplace) The creation of the new file failed for the indicated reason.

Can't do inplace edit without backup

(F) You're on a system such as MS-DOS that gets confused if you try reading from a deleted (but still opened) file. You have to say `-i.bak`, or some such.

Can't do inplace edit: %s would not be unique

(S inplace) Your filesystem does not support filenames longer than 14 characters and Perl was unable to create a unique filename during inplace editing with the **-i** switch. The file was ignored.

Can't do %s("%s") on non-UTF-8 locale; resolved to "%s".

(W locale) You are 1) running under "`use locale`"; 2) the current locale is not a UTF-8 one; 3) you tried to do the designated case-change operation on the specified Unicode character; and 4) the result of this operation would mix

Unicode and locale rules, which likely conflict. Mixing of different rule types is forbidden, so the operation was not done; instead the result is the indicated value, which is the best available that uses entirely Unicode rules. That turns out to almost always be the original character, unchanged.

It is generally a bad idea to mix non-UTF-8 locales and Unicode, and this issue is one of the reasons why. This warning is raised when Unicode rules would normally cause the result of this operation to contain a character that is in the range specified by the locale, 0..255, and hence is subject to the locale's rules, not Unicode's.

If you are using locale purely for its characteristics related to things like its numeric and time formatting (and not `LC_CTYPE`), consider using a restricted form of the locale pragma (see Section 38.5.1 [perllocale The "use locale" pragma], page 703) like `"use locale ':not_characters'"`.

Note that failed case-changing operations done as a result of case-insensitive `/i` regular expression matching will show up in this warning as having the `fc` operation (as that is what the regular expression engine calls behind the scenes.)

Can't do waitpid with flags

(F) This machine doesn't have either waitpid() or wait4(), so only waitpid() without flags is emulated.

Can't emulate -%s on #! line

(F) The #! line specifies a switch that doesn't make sense at this point. For example, it'd be kind of silly to put a **-x** on the #! line.

Can't %s %s-endian %ss on this platform

(F) Your platform's byte-order is neither big-endian nor little-endian, or it has a very strange pointer size. Packing and unpacking big- or little-endian floating point values and pointers may not be possible. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

Can't exec "%s": %s

(W exec) A system(), exec(), or piped open call could not execute the named program for the indicated reason. Typical reasons include: the permissions were wrong on the file, the file wasn't found in `$ENV{PATH}`, the executable in question was compiled for another architecture, or the #! line in a script points to an interpreter that can't be run for similar reasons. (Or maybe your system doesn't support #! at all.)

Can't exec %s

(F) Perl was trying to execute the indicated program for you because that's what the #! line said. If that's not what you wanted, you may need to mention "perl" on the #! line somewhere.

Can't execute %s

(F) You used the **-S** switch, but the copies of the script to execute found in the PATH did not have correct permissions.

Can't find an opnumber for "%s"

(F) A string of a form `CORE::word` was given to prototype(), but there is no builtin with the name `word`.

Can't find %s character property "%s"

(F) You used `\p{}` or `\P{}` but the character property by that name could not be found. Maybe you misspelled the name of the property? See Section "Properties accessible through \p{} and \P{}" in `perluniprops` for a complete list of available official properties.

Can't find label %s

(F) You said to goto a label that isn't mentioned anywhere that it's possible for us to go to. See ⟨undefined⟩ [perlfunc goto], page ⟨undefined⟩.

Can't find %s on PATH

(F) You used the **-S** switch, but the script to execute could not be found in the PATH.

Can't find %s on PATH, '.' not in PATH

(F) You used the **-S** switch, but the script to execute could not be found in the PATH, or at least not with the correct permissions. The script exists in the current directory, but PATH prohibits running it.

Can't find string terminator %s anywhere before EOF

(F) Perl strings can stretch over multiple lines. This message means that the closing delimiter was omitted. Because bracketed quotes count nesting levels, the following is missing its final parenthesis:

```
    print q(The character '(' starts a side comment.);
```

If you're getting this error from a here-document, you may have included unseen whitespace before or after your closing tag or there may not be a linebreak after it. A good programmer's editor will have a way to help you find these characters (or lack of characters). See Section 48.1 [perlop NAME], page 798 for the full details on here-documents.

Can't find Unicode property definition "%s"

(F) You may have tried to use `\p` which means a Unicode property (for example `\p{Lu}` matches all uppercase letters). If you did mean to use a Unicode property, see Section "Properties accessible through \p{} and \P{}" in `perluniprops` for a complete list of available properties. If you didn't mean to use a Unicode property, escape the `\p`, either by `\\p` (just the `\p`) or by `\Q\p` (the rest of the string, or until `\E`).

Can't fork: %s

(F) A fatal error occurred while trying to fork while opening a pipeline.

Can't fork, trying again in 5 seconds

(W pipe) A fork in a piped open failed with EAGAIN and will be retried after five seconds.

Can't get filespec - stale stat buffer?

(S) A warning peculiar to VMS. This arises because of the difference between access checks under VMS and under the Unix model Perl assumes. Under VMS, access checks are done by filename, rather than by bits in the stat buffer, so that ACLs and other protections can be taken into account. Unfortunately, Perl assumes that the stat buffer contains all the necessary information, and passes it, instead of the filespec, to the access-checking routine. It will try to retrieve the filespec using the device name and FID present in the stat buffer, but this works only if you haven't made a subsequent call to the CRTL stat() routine, because the device name is overwritten with each call. If this warning appears, the name lookup failed, and the access-checking routine gave up and returned FALSE, just to be conservative. (Note: The access-checking routine knows about the Perl `stat` operator and file tests, so you shouldn't ever see this warning in response to a Perl command; it arises only if some internal code takes stat buffers lightly.)

Can't get pipe mailbox device name

(P) An error peculiar to VMS. After creating a mailbox to act as a pipe, Perl can't retrieve its name for later use.

Can't get SYSGEN parameter value for MAXBUF

(P) An error peculiar to VMS. Perl asked $GETSYI how big you want your mailbox buffers to be, and didn't get an answer.

Can't "goto" into the middle of a foreach loop

(F) A "goto" statement was executed to jump into the middle of a foreach loop. You can't get there from here. See ⟨undefined⟩ [perlfunc goto], page ⟨undefined⟩.

Can't "goto" out of a pseudo block

(F) A "goto" statement was executed to jump out of what might look like a block, except that it isn't a proper block. This usually occurs if you tried to jump out of a sort() block or subroutine, which is a no-no. See ⟨undefined⟩ [perlfunc goto], page ⟨undefined⟩.

Can't goto subroutine from an eval-%s

(F) The "goto subroutine" call can't be used to jump out of an eval "string" or block.

Can't goto subroutine from a sort sub (or similar callback)

(F) The "goto subroutine" call can't be used to jump out of the comparison sub for a sort(), or from a similar callback (such as the reduce() function in List::Util).

Can't goto subroutine outside a subroutine

(F) The deeply magical "goto subroutine" call can only replace one subroutine call for another. It can't manufacture one out of whole cloth. In general you should be calling it out of only an AUTOLOAD routine anyway. See ⟨undefined⟩ [perlfunc goto], page ⟨undefined⟩.

Can't ignore signal CHLD, forcing to default

(W signal) Perl has detected that it is being run with the SIGCHLD signal (sometimes known as SIGCLD) disabled. Since disabling this signal will interfere with proper determination of exit status of child processes, Perl has reset the signal to its default value. This situation typically indicates that the parent program under which Perl may be running (e.g. cron) is being very careless.

Can't kill a non-numeric process ID

(F) Process identifiers must be (signed) integers. It is a fatal error to attempt to kill() an undefined, empty-string or otherwise non-numeric process identifier.

Can't "last" outside a loop block

(F) A "last" statement was executed to break out of the current block, except that there's this itty bitty problem called there isn't a current block. Note that an "if" or "else" block doesn't count as a "loopish" block, as doesn't a block given to sort(), map() or grep(). You can usually double the curlies to get the same effect though, because the inner curlies will be considered a block that loops once. See [perlfunc last], page 399.

Can't linearize anonymous symbol table

(F) Perl tried to calculate the method resolution order (MRO) of a package, but failed because the package stash has no name.

Can't load '%s' for module %s

(F) The module you tried to load failed to load a dynamic extension. This may either mean that you upgraded your version of perl to one that is incompatible with your old dynamic extensions (which is known to happen between major versions of perl), or (more likely) that your dynamic extension was built against an older version of the library that is installed on your system. You may need to rebuild your old dynamic extensions.

Can't localize lexical variable %s

(F) You used local on a variable name that was previously declared as a lexical variable using "my" or "state". This is not allowed. If you want to localize a package variable of the same name, qualify it with the package name.

Can't localize through a reference

(F) You said something like `local $$ref`, which Perl can't currently handle, because when it goes to restore the old value of whatever $ref pointed to after the scope of the local() is finished, it can't be sure that $ref will still be a reference.

Can't locate %s

(F) You said to `do` (or `require`, or `use`) a file that couldn't be found. Perl looks for the file in all the locations mentioned in @INC, unless the file name included the full path to the file. Perhaps you need to set the PERL5LIB or PERL5OPT environment variable to say where the extra library is, or maybe the script needs to add the library name to @INC. Or maybe you just misspelled the name of the file. See [perlfunc require], page 437 and `lib`.

Can't locate auto/%s.al in @INC

(F) A function (or method) was called in a package which allows autoload, but there is no function to autoload. Most probable causes are a misprint in a function/method name or a failure to `AutoSplit` the file, say, by doing `make install`.

Can't locate loadable object for module %s in @INC

(F) The module you loaded is trying to load an external library, like for example, `foo.so` or `bar.dll`, but the `DynaLoader` module was unable to locate this library. See `DynaLoader`.

Can't locate object method "%s" via package "%s"

(F) You called a method correctly, and it correctly indicated a package functioning as a class, but that package doesn't define that particular method, nor does any of its base classes. See Section 46.1 [perlobj NAME], page 769.

Can't locate object method "%s" via package "%s" (perhaps you forgot to load "%s"?)

(F) You called a method on a class that did not exist, and the method could not be found in UNIVERSAL. This often means that a method requires a package that has not been loaded.

Can't locate package %s for @%s::ISA

(W syntax) The @ISA array contained the name of another package that doesn't seem to exist.

Can't locate PerlIO%s

(F) You tried to use in open() a PerlIO layer that does not exist, e.g. open(FH, ">:nosuchlayer", "somefile").

Can't make list assignment to %ENV on this system

(F) List assignment to %ENV is not supported on some systems, notably VMS.

Can't make loaded symbols global on this platform while loading %s

(S) A module passed the flag 0x01 to DynaLoader::dl_load_file() to request that symbols from the stated file are made available globally within the process, but that functionality is not available on this platform. Whilst the module likely will still work, this may prevent the perl interpreter from loading other XS-based extensions which need to link directly to functions defined in the C or XS code in the stated file.

Can't modify %s in %s

(F) You aren't allowed to assign to the item indicated, or otherwise try to change it, such as with an auto-increment.

Can't modify nonexistent substring

(P) The internal routine that does assignment to a substr() was handed a NULL.

Can't modify non-lvalue subroutine call

(F) Subroutines meant to be used in lvalue context should be declared as such. See Section 73.3.5 [perlsub Lvalue subroutines], page 1231.

Can't modify reference to %s in %s assignment

(F) Only a limited number of constructs can be used as the argument to a reference constructor on the left-hand side of an assignment, and what you used was not one of them. See Section 62.6 [perlref Assigning to References], page 1089.

Can't modify reference to localized parenthesized array in list assignment

(F) Assigning to `\local(@array)` or `\(local @array)` is not supported, as it is not clear exactly what it should do. If you meant to make @array refer to some other array, use `\@array = \@other_array`. If you want to make the elements of @array aliases of the scalars referenced on the right-hand side, use `\(@array) = @scalar_refs`.

Can't modify reference to parenthesized hash in list assignment

(F) Assigning to `\(%hash)` is not supported. If you meant to make %hash refer to some other hash, use `\%hash = \%other_hash`. If you want to make the elements of %hash into aliases of the scalars referenced on the right-hand side, use a hash slice: `\@hash{@keys} = @those_scalar_refs`.

Can't msgrcv to read-only var

(F) The target of a msgrcv must be modifiable to be used as a receive buffer.

Can't "next" outside a loop block

(F) A "next" statement was executed to reiterate the current block, but there isn't a current block. Note that an "if" or "else" block doesn't count as a "loopish" block, as doesn't a block given to sort(), map() or grep(). You can usually double the curlies to get the same effect though, because the inner curlies will be considered a block that loops once. See [perlfunc next], page 406.

Can't open %s: %s

(S inplace) The implicit opening of a file through use of the `<>` filehandle, either implicitly under the `-n` or `-p` command-line switches, or explicitly, failed for the indicated reason. Usually this is because you don't have read permission for a file which you named on the command line.

(F) You tried to call perl with the `-e` switch, but `/dev/null` (or your operating system's equivalent) could not be opened.

Can't open a reference

(W io) You tried to open a scalar reference for reading or writing, using the 3-arg open() syntax:

```
    open FH, '>', $ref;
```

but your version of perl is compiled without perlio, and this form of open is not supported.

Can't open bidirectional pipe

(W pipe) You tried to say `open(CMD, "|cmd|")`, which is not supported. You can try any of several modules in the Perl library to do this, such as IPC::Open2. Alternately, direct the pipe's output to a file using `">"`, and then read it in under a different file handle.

Can't open error file %s as stderr

(F) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the file specified after '2>' or '2>>' on the command line for writing.

Can't open input file %s as stdin

(F) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the file specified after '<' on the command line for reading.

Can't open output file %s as stdout

(F) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the file specified after '>' or '>>' on the command line for writing.

Can't open output pipe (name: %s)

(P) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the pipe into which to send data destined for stdout.

Can't open perl script "%s": %s

(F) The script you specified can't be opened for the indicated reason.

If you're debugging a script that uses #!, and normally relies on the shell's $PATH search, the -S option causes perl to do that search, so you don't have to type the path or `` `which $scriptname` ``.

Can't read CRTL environ

(S) A warning peculiar to VMS. Perl tried to read an element of %ENV from the CRTL's internal environment array and discovered the array was missing. You need to figure out where your CRTL misplaced its environ or define `PERL_ENV_TABLES` (see Section 87.1 [perlvms NAME], page 1409) so that environ is not searched.

Can't "redo" outside a loop block

(F) A "redo" statement was executed to restart the current block, but there isn't a current block. Note that an "if" or "else" block doesn't count as a "loopish" block, as doesn't a block given to sort(), map() or grep(). You can usually double the curlies to get the same effect though, because the inner curlies will be considered a block that loops once. See [perlfunc redo], page 435.

Can't remove %s: %s, skipping file

(S inplace) You requested an inplace edit without creating a backup file. Perl was unable to remove the original file to replace it with the modified file. The file was left unmodified.

Can't rename %s to %s: %s, skipping file

(S inplace) The rename done by the **-i** switch failed for some reason, probably because you don't have write permission to the directory.

Can't reopen input pipe (name: %s) in binary mode

(P) An error peculiar to VMS. Perl thought stdin was a pipe, and tried to reopen it to accept binary data. Alas, it failed.

Can't represent character for Ox%X on this platform

(F) There is a hard limit to how big a character code point can be due to the fundamental properties of UTF-8, especially on EBCDIC platforms. The given code point exceeds that. The only work-around is to not use such a large code point.

Can't reset %ENV on this system

(F) You called `reset('E')` or similar, which tried to reset all variables in the current package beginning with "E". In the main package, that includes %ENV. Resetting %ENV is not supported on some systems, notably VMS.

Can't resolve method "%s" overloading "%s" in package "%s"

(F)(P) Error resolving overloading specified by a method name (as opposed to a subroutine reference): no such method callable via the package. If the method name is `???`, this is an internal error.

Can't return %s from lvalue subroutine

(F) Perl detected an attempt to return illegal lvalues (such as temporary or readonly values) from a subroutine used as an lvalue. This is not allowed.

Can't return outside a subroutine

(F) The return statement was executed in mainline code, that is, where there was no subroutine call to return out of. See Section 73.1 [perlsub NAME], page 1216.

Can't return %s to lvalue scalar context

(F) You tried to return a complete array or hash from an lvalue subroutine, but you called the subroutine in a way that made Perl think you meant to return only one value. You probably meant to write parentheses around the call to the subroutine, which tell Perl that the call should be in list context.

Can't stat script "%s"

(P) For some reason you can't fstat() the script even though you have it open already. Bizarre.

Can't take log of %g

(F) For ordinary real numbers, you can't take the logarithm of a negative number or zero. There's a Math::Complex package that comes standard with Perl, though, if you really want to do that for the negative numbers.

Can't take sqrt of %g

(F) For ordinary real numbers, you can't take the square root of a negative number. There's a Math::Complex package that comes standard with Perl, though, if you really want to do that.

Can't undef active subroutine

(F) You can't undefine a routine that's currently running. You can, however, redefine it while it's running, and you can even undef the redefined subroutine while the old routine is running. Go figure.

Can't upgrade %s (%d) to %d

(P) The internal sv_upgrade routine adds "members" to an SV, making it into a more specialized kind of SV. The top several SV types are so specialized, however, that they cannot be interconverted. This message indicates that such a conversion was attempted.

Can't use '%c' after -mname

(F) You tried to call perl with the **-m** switch, but you put something other than "=" after the module name.

Can't use a hash as a reference

(F) You tried to use a hash as a reference, as in `%foo->{"bar"}` or `%$ref->{"hello"}`. Versions of perl <= 5.22.0 used to allow this syntax, but shouldn't have. This was deprecated in perl 5.6.1.

Can't use an array as a reference

(F) You tried to use an array as a reference, as in `@foo->[23]` or `@$ref->[99]`. Versions of perl <= 5.22.0 used to allow this syntax, but shouldn't have. This was deprecated in perl 5.6.1.

Can't use anonymous symbol table for method lookup

(F) The internal routine that does method lookup was handed a symbol table that doesn't have a name. Symbol tables can become anonymous for example by undefining stashes: `undef %Some::Package::`.

Can't use an undefined value as %s reference

(F) A value used as either a hard reference or a symbolic reference must be a defined value. This helps to delurk some insidious errors.

Can't use bareword ("%s") as %s ref while "strict refs" in use

(F) Only hard references are allowed by "strict refs". Symbolic references are disallowed. See Section 62.1 [perlref NAME], page 1077.

Can't use %! because Errno.pm is not available

(F) The first time the `%!` hash is used, perl automatically loads the Errno.pm module. The Errno module is expected to tie the %! hash to provide symbolic names for `$!` errno values.

Can't use both '<' and '>' after type '%c' in %s

(F) A type cannot be forced to have both big-endian and little-endian byte-order at the same time, so this combination of modifiers is not allowed. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

Can't use 'defined(@array)' (Maybe you should just omit the defined()?)

(F) defined() is not useful on arrays because it checks for an undefined *scalar* value. If you want to see if the array is empty, just use `if (@array) { # not empty }` for example.

Can't use 'defined(%hash)' (Maybe you should just omit the defined()?)

(F) `defined()` is not usually right on hashes.

Although `defined %hash` is false on a plain not-yet-used hash, it becomes true in several non-obvious circumstances, including iterators, weak references, stash names, even remaining true after `undef %hash`. These things make `defined %hash` fairly useless in practice, so it now generates a fatal error.

If a check for non-empty is what you wanted then just put it in boolean context (see Section 11.2.4 [perldata Scalar values], page 74):

```
if (%hash) {
   # not empty
}
```

If you had `defined %Foo::Bar::QUUX` to check whether such a package variable exists then that's never really been reliable, and isn't a good way to enquire about the features of a package, or whether it's loaded, etc.

Can't use %s for loop variable

(P) The parser got confused when trying to parse a `foreach` loop.

Can't use global %s in "%s"

(F) You tried to declare a magical variable as a lexical variable. This is not allowed, because the magic can be tied to only one location (namely the global variable) and it would be incredibly confusing to have variables in your program that looked like magical variables but weren't.

Can't use '%c' in a group with different byte-order in %s

> (F) You attempted to force a different byte-order on a type that is already inside a group with a byte-order modifier. For example you cannot force little-endianness on a type that is inside a big-endian group.

Can't use "my %s" in sort comparison

> (F) The global variables $a and $b are reserved for sort comparisons. You mentioned $a or $b in the same line as the `<=>` or cmp operator, and the variable had earlier been declared as a lexical variable. Either qualify the sort variable with the package name, or rename the lexical variable.

Can't use %s ref as %s ref

> (F) You've mixed up your reference types. You have to dereference a reference of the type needed. You can use the ref() function to test the type of the reference, if need be.

Can't use string ("%s") as %s ref while "strict refs" in use
Can't use string ("%s"...) as %s ref while "strict refs" in use

> (F) You've told Perl to dereference a string, something which `use strict` blocks to prevent it happening accidentally. See Section 62.3.4 [perlref Symbolic references], page 1084. This can be triggered by an `@` or `$` in a double-quoted string immediately before interpolating a variable, for example in `"user @$twitter_id"`, which says to treat the contents of `$twitter_id` as an array reference; use a `\` to have a literal `@` symbol followed by the contents of `$twitter_id`: `"user \@$twitter_id"`.

Can't use subscript on %s

> (F) The compiler tried to interpret a bracketed expression as a subscript. But to the left of the brackets was an expression that didn't look like a hash or array reference, or anything else subscriptable.

Can't use \%c to mean $%c in expression

> (W syntax) In an ordinary expression, backslash is a unary operator that creates a reference to its argument. The use of backslash to indicate a backreference to a matched substring is valid only as part of a regular expression pattern. Trying to do this in ordinary Perl code produces a value that prints out looking like SCALAR(0xdecaf). Use the $1 form instead.

Can't weaken a nonreference

> (F) You attempted to weaken something that was not a reference. Only references can be weakened.

Can't "when" outside a topicalizer

> (F) You have used a when() block that is neither inside a `foreach` loop nor a `given` block. (Note that this error is issued on exit from the `when` block, so you won't get the error if the match fails, or if you use an explicit `continue`.)

Can't x= to read-only value

> (F) You tried to repeat a constant value (often the undefined value) with an assignment operator, which implies modifying the value itself. Perhaps you need to copy the value to a temporary, and repeat that.

Character following "\c" must be printable ASCII

(F) In \c*X*, *X* must be a printable (non-control) ASCII character.

Note that ASCII characters that don't map to control characters are discouraged, and will generate the warning (when enabled) ["\c%c" is more clearly written simply as "%s"], page 160.

Character in 'C' format wrapped in pack

(W pack) You said

```
pack("C", $x)
```

where $x is either less than 0 or more than 255; the "C" format is only for encoding native operating system characters (ASCII, EBCDIC, and so on) and not for Unicode characters, so Perl behaved as if you meant

```
pack("C", $x & 255)
```

If you actually want to pack Unicode codepoints, use the "U" format instead.

Character in 'c' format wrapped in pack

(W pack) You said

```
pack("c", $x)
```

where $x is either less than -128 or more than 127; the "c" format is only for encoding native operating system characters (ASCII, EBCDIC, and so on) and not for Unicode characters, so Perl behaved as if you meant

```
pack("c", $x & 255);
```

If you actually want to pack Unicode codepoints, use the "U" format instead.

Character in '%c' format wrapped in unpack

(W unpack) You tried something like

```
unpack("H", "\x{2a1}")
```

where the format expects to process a byte (a character with a value below 256), but a higher value was provided instead. Perl uses the value modulus 256 instead, as if you had provided:

```
unpack("H", "\x{a1}")
```

Character in 'W' format wrapped in pack

(W pack) You said

```
pack("U0W", $x)
```

where $x is either less than 0 or more than 255. However, U0-mode expects all values to fall in the interval [0, 255], so Perl behaved as if you meant:

```
pack("U0W", $x & 255)
```

Character(s) in '%c' format wrapped in pack

(W pack) You tried something like

```
pack("u", "\x{1f3}b")
```

where the format expects to process a sequence of bytes (character with a value below 256), but some of the characters had a higher value. Perl uses the character values modulus 256 instead, as if you had provided:

```
pack("u", "\x{f3}b")
```

Character(s) in '%c' format wrapped in unpack

(W unpack) You tried something like

```
unpack("s", "\x{1f3}b")
```

where the format expects to process a sequence of bytes (character with a value below 256), but some of the characters had a higher value. Perl uses the character values modulus 256 instead, as if you had provided:

```
unpack("s", "\x{f3}b")
```

charnames alias definitions may not contain a sequence of multiple spaces

(F) You defined a character name which had multiple space characters in a row. Change them to single spaces. Usually these names are defined in the `:alias` import argument to `use charnames`, but they could be defined by a translator installed into `$^H{charnames}`. See Section "CUSTOM ALIASES" in `charnames`.

charnames alias definitions may not contain trailing white-space

(F) You defined a character name which ended in a space character. Remove the trailing space(s). Usually these names are defined in the `:alias` import argument to `use charnames`, but they could be defined by a translator installed into `$^H{charnames}`. See Section "CUSTOM ALIASES" in `charnames`.

\C is deprecated in regex; marked by <– HERE in m/%s/

(D deprecated, regexp) The `\C` character class is deprecated, and will become a compile-time error in a future release of perl (tentatively v5.24). This construct allows you to match a single byte of what makes up a multi-byte single UTF8 character, and breaks encapsulation. It is currently also very buggy. If you really need to process the individual bytes, you probably want to convert your string to one where each underlying byte is stored as a character, with utf8::encode().

"\c%c" is more clearly written simply as "%s"

(W syntax) The `\cX` construct is intended to be a way to specify non-printable characters. You used it for a printable one, which is better written as simply itself, perhaps preceded by a backslash for non-word characters. Doing it the way you did is not portable between ASCII and EBCDIC platforms.

Cloning substitution context is unimplemented

(F) Creating a new thread inside the `s///` operator is not supported.

closedir() attempted on invalid dirhandle %s

(W io) The dirhandle you tried to close is either closed or not really a dirhandle. Check your control flow.

close() on unopened filehandle %s

(W unopened) You tried to close a filehandle that was never opened.

Closure prototype called

(F) If a closure has attributes, the subroutine passed to an attribute handler is the prototype that is cloned when a new closure is created. This subroutine cannot be called.

Code missing after '/'

(F) You had a (sub-)template that ends with a '/'. There must be another template code following the slash. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

Code point 0x%X is not Unicode, may not be portable

(S non_unicode) You had a code point above the Unicode maximum of U+10FFFF.

Perl allows strings to contain a superset of Unicode code points, up to the limit of what is storable in an unsigned integer on your system, but these may not be accepted by other languages/systems. At one time, it was legal in some standards to have code points up to 0x7FFF_FFFF, but not higher. Code points above 0xFFFF_FFFF require larger than a 32 bit word.

%s: Command not found

(A) You've accidentally run your script through **csh** or another shell instead of Perl. Check the #! line, or manually feed your script into Perl yourself. The #! line at the top of your file could look like

```
#!/usr/bin/perl -w
```

Compilation failed in require

(F) Perl could not compile a file specified in a `require` statement. Perl uses this generic message when none of the errors that it encountered were severe enough to halt compilation immediately.

Complex regular subexpression recursion limit (%d) exceeded

(W regexp) The regular expression engine uses recursion in complex situations where back-tracking is required. Recursion depth is limited to 32766, or perhaps less in architectures where the stack cannot grow arbitrarily. ("Simple" and "medium" situations are handled without recursion and are not subject to a limit.) Try shortening the string under examination; looping in Perl code (e.g. with `while`) rather than in the regular expression engine; or rewriting the regular expression so that it is simpler or backtracks less. (See `perlfaq2` for information on *Mastering Regular Expressions.*)

connect() on closed socket %s

(W closed) You tried to do a connect on a closed socket. Did you forget to check the return value of your socket() call? See ⟨undefined⟩ [perlfunc connect], page ⟨undefined⟩.

Constant(%s): Call to &{$^H{%s}} did not return a defined value

(F) The subroutine registered to handle constant overloading (see `overload`) or a custom charnames handler (see Section "CUSTOM TRANSLATORS" in `charnames`) returned an undefined value.

Constant(%s): $^H{%s} is not defined

(F) The parser found inconsistencies while attempting to define an overloaded constant. Perhaps you forgot to load the corresponding `overload` pragma?

Constant is not %s reference

(F) A constant value (perhaps declared using the `use constant` pragma) is being dereferenced, but it amounts to the wrong type of reference. The message indicates the type of reference that was expected. This usually indicates a syntax error in dereferencing the constant value. See Section 73.3.11 [perlsub Constant Functions], page 1241 and `constant`.

Constants from lexical variables potentially modified elsewhere are deprecated

(D deprecated) You wrote something like

```
    my $var;
    $sub = sub () { $var };
```

but $var is referenced elsewhere and could be modified after the `sub` expression is evaluated. Either it is explicitly modified elsewhere (`$var = 3`) or it is passed to a subroutine or to an operator like `printf` or `map`, which may or may not modify the variable.

Traditionally, Perl has captured the value of the variable at that point and turned the subroutine into a constant eligible for inlining. In those cases where the variable can be modified elsewhere, this breaks the behavior of closures, in which the subroutine captures the variable itself, rather than its value, so future changes to the variable are reflected in the subroutine's return value.

This usage is deprecated, because the behavior is likely to change in a future version of Perl.

If you intended for the subroutine to be eligible for inlining, then make sure the variable is not referenced elsewhere, possibly by copying it:

```
    my $var2 = $var;
    $sub = sub () { $var2 };
```

If you do want this subroutine to be a closure that reflects future changes to the variable that it closes over, add an explicit `return`:

```
    my $var;
    $sub = sub () { return $var };
```

Constant subroutine %s redefined

(W redefine)(S) You redefined a subroutine which had previously been eligible for inlining. See Section 73.3.11 [perlsub Constant Functions], page 1241 for commentary and workarounds.

Constant subroutine %s undefined

(W misc) You undefined a subroutine which had previously been eligible for inlining. See Section 73.3.11 [perlsub Constant Functions], page 1241 for commentary and workarounds.

Constant(%s) unknown

(F) The parser found inconsistencies either while attempting to define an overloaded constant, or when trying to find the character name specified in the `\N{...}` escape. Perhaps you forgot to load the corresponding `overload` pragma?

:const is experimental

(S experimental::const_attr) The "const" attribute is experimental. If you want to use the feature, disable the warning with `no warnings 'experimental::const_attr'`, but know that in doing so you are taking the risk that your code may break in a future Perl version.

:const is not permitted on named subroutines

(F) The "const" attribute causes an anonymous subroutine to be run and its value captured at the time that it is cloned. Named subroutines are not cloned like this, so the attribute does not make sense on them.

Copy method did not return a reference

(F) The method which overloads "=" is buggy. See Section “Copy Constructor” in `overload`.

&CORE::%s cannot be called directly

(F) You tried to call a subroutine in the `CORE::` namespace with `&foo` syntax or through a reference. Some subroutines in this package cannot yet be called that way, but must be called as barewords. Something like this will work:

```
BEGIN { *shove = \&CORE::push; }
shove @array, 1,2,3; # pushes on to @array
```

CORE::%s is not a keyword

(F) The CORE:: namespace is reserved for Perl keywords.

Corrupted regexp opcode %d > %d

(P) This is either an error in Perl, or, if you’re using one, your Section 59.1 [custom regular expression engine], page 1032. If not the latter, report the problem through the `perlbug` utility.

corrupted regexp pointers

(P) The regular expression engine got confused by what the regular expression compiler gave it.

corrupted regexp program

(P) The regular expression engine got passed a regexp program without a valid magic number.

Corrupt malloc ptr 0x%x at 0x%x

(P) The malloc package that comes with Perl had an internal failure.

Count after length/code in unpack

(F) You had an unpack template indicating a counted-length string, but you have also specified an explicit size for the string. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

Deep recursion on anonymous subroutine
Deep recursion on subroutine "%s"

(W recursion) This subroutine has called itself (directly or indirectly) 100 times more than it has returned. This probably indicates an infinite recursion, unless you’re writing strange benchmark programs, in which case it indicates something else.

This threshold can be changed from 100, by recompiling the `perl` binary, setting the C pre-processor macro `PERL_SUB_DEPTH_WARN` to the desired value.

(?(DEFINE)....) does not allow branches in regex; marked by <– HERE in m/%s/

(F) You used something like `(?(DEFINE)...|..)` which is illegal. The most likely cause of this error is that you left out a parenthesis inside of the `....` part.

The <– HERE shows whereabouts in the regular expression the problem was discovered.

%s defines neither package nor VERSION–version check failed

(F) You said something like "use Module 42" but in the Module file there are neither package declarations nor a `$VERSION`.

delete argument is index/value array slice, use array slice

(F) You used index/value array slice syntax (`%array[...]`) as the argument to `delete`. You probably meant `@array[...]` with an @ symbol instead.

delete argument is key/value hash slice, use hash slice

(F) You used key/value hash slice syntax (`%hash{...}`) as the argument to `delete`. You probably meant `@hash{...}` with an @ symbol instead.

delete argument is not a HASH or ARRAY element or slice

(F) The argument to `delete` must be either a hash or array element, such as:

```
    $foo{$bar}
    $ref->{"susie"}[12]
```

or a hash or array slice, such as:

```
    @foo[$bar, $baz, $xyzzy]
    @{$ref->[12]}{"susie", "queue"}
```

Delimiter for here document is too long

(F) In a here document construct like `<<FOO`, the label `FOO` is too long for Perl to handle. You have to be seriously twisted to write code that triggers this error.

Deprecated use of my() in false conditional

(D deprecated) You used a declaration similar to `my $x if 0`. There has been a long-standing bug in Perl that causes a lexical variable not to be cleared at scope exit when its declaration includes a false conditional. Some people have exploited this bug to achieve a kind of static variable. Since we intend to fix this bug, we don't want people relying on this behavior. You can achieve a similar static effect by declaring the variable in a separate block outside the function, eg

```
    sub f { my $x if 0; return $x++ }
```

becomes

```
    { my $x; sub f { return $x++ } }
```

Beginning with perl 5.10.0, you can also use `state` variables to have lexicals that are initialized only once (see `feature`):

```
    sub f { state $x; return $x++ }
```

DESTROY created new reference to dead object '%s'

(F) A DESTROY() method created a new reference to the object which is just being DESTROYed. Perl is confused, and prefers to abort rather than to create a dangling reference.

Did not produce a valid header

See Server error.

%s did not return a true value

(F) A required (or used) file must return a true value to indicate that it compiled correctly and ran its initialization code correctly. It's traditional to end such a file with a "1;", though any true value would do. See [perlfunc require], page 437.

(Did you mean &%s instead?)

(W misc) You probably referred to an imported subroutine &FOO as $FOO or some such.

(Did you mean "local" instead of "our"?)

(W misc) Remember that "our" does not localize the declared global variable. You have declared it again in the same lexical scope, which seems superfluous.

(Did you mean $ or @ instead of %?)

(W) You probably said %hash{$key} when you meant $hash{$key} or @hash{@keys}. On the other hand, maybe you just meant %hash and got carried away.

Died

(F) You passed die() an empty string (the equivalent of `die ""`) or you called it with no args and `$@` was empty.

Document contains no data

See Server error.

%s does not define %s::VERSION–version check failed

(F) You said something like "use Module 42" but the Module did not define a `$VERSION`.

'/' does not take a repeat count

(F) You cannot put a repeat count of any kind right after the '/' code. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

Don't know how to get file name

(P) `PerlIO_getname`, a perl internal I/O function specific to VMS, was somehow called on another platform. This should not happen.

Don't know how to handle magic of type \%o

(P) The internal handling of magical variables has been cursed.

do_study: out of memory

(P) This should have been caught by safemalloc() instead.

(Do you need to predeclare %s?)

(S syntax) This is an educated guess made in conjunction with the message "%s found where operator expected". It often means a subroutine or module name is being referenced that hasn't been declared yet. This may be because of ordering problems in your file, or because of a missing "sub", "package", "require", or "use" statement. If you're referencing something that isn't defined yet, you don't actually have to define the subroutine or package before the current location. You can use an empty "sub foo;" or "package FOO;" to enter a "forward" declaration.

dump() better written as CORE::dump()

(W misc) You used the obsolescent `dump()` built-in function, without fully qualifying it as `CORE::dump()`. Maybe it's a typo. See [perlfunc dump], page 374.

dump is not supported

(F) Your machine doesn't support dump/undump.

Duplicate free() ignored

(S malloc) An internal routine called free() on something that had already been freed.

Duplicate modifier '%c' after '%c' in %s

(W unpack) You have applied the same modifier more than once after a type in a pack template. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

each on reference is experimental

(S experimental::autoderef) `each` with a scalar argument is experimental and may change or be removed in a future Perl version. If you want to take the risk of using this feature, simply disable this warning:

```
    no warnings "experimental::autoderef";
```

elseif should be elsif

(S syntax) There is no keyword "elseif" in Perl because Larry thinks it's ugly. Your code will be interpreted as an attempt to call a method named "elseif" for the class returned by the following block. This is unlikely to be what you want.

Empty \%c{} in regex; marked by <– HERE in m/%s/

(F) `\p` and `\P` are used to introduce a named Unicode property, as described in Section 81.1 [perlunicode NAME], page 1317 and Section 58.1 [perlre NAME], page 989. You used `\p` or `\P` in a regular expression without specifying the property name.

entering effective %s failed

(F) While under the `use filetest` pragma, switching the real and effective uids or gids failed.

%ENV is aliased to %s

(F) You're running under taint mode, and the `%ENV` variable has been aliased to another hash, so it doesn't reflect anymore the state of the program's environment. This is potentially insecure.

Error converting file specification %s

(F) An error peculiar to VMS. Because Perl may have to deal with file specifications in either VMS or Unix syntax, it converts them to a single form when it must operate on them directly. Either you've passed an invalid file specification to Perl, or you've found a case the conversion routines don't handle. Drat.

Eval-group in insecure regular expression

(F) Perl detected tainted data when trying to compile a regular expression that contains the `(?{ ... })` zero-width assertion, which is unsafe. See [perlre (?{ code })], page 1009, and Section 70.1 [perlsec NAME], page 1198.

Eval-group not allowed at runtime, use re 'eval' in regex m/%s/

(F) Perl tried to compile a regular expression containing the `(?{ ... })` zero-width assertion at run time, as it would when the pattern contains interpolated values. Since that is a security risk, it is not allowed. If you insist, you may still do this by using the `re 'eval'` pragma or by explicitly building the pattern from an interpolated string at run time and using that in an eval(). See [perlre (?{ code })], page 1009.

Eval-group not allowed, use re 'eval' in regex m/%s/

(F) A regular expression contained the `(?{ ... })` zero-width assertion, but that construct is only allowed when the `use re 'eval'` pragma is in effect. See [perlre (?{ code })], page 1009.

EVAL without pos change exceeded limit in regex; marked by <-- HERE in m/%s/

(F) You used a pattern that nested too many EVAL calls without consuming any text. Restructure the pattern so that text is consumed.

The <-- HERE shows whereabouts in the regular expression the problem was discovered.

Excessively long `<>` operator

(F) The contents of a `<>` operator may not exceed the maximum size of a Perl identifier. If you're just trying to glob a long list of filenames, try using the glob() operator, or put the filenames into a variable and glob that.

exec? I'm not *that* kind of operating system

(F) The `exec` function is not implemented on some systems, e.g., Symbian OS. See Section 56.1 [perlport NAME], page 951.

Execution of %s aborted due to compilation errors.

(F) The final summary message when a Perl compilation fails.

exists argument is not a HASH or ARRAY element or a subroutine

(F) The argument to `exists` must be a hash or array element or a subroutine with an ampersand, such as:

```
$foo{$bar}
$ref->{"susie"}[12]
&do_something
```

exists argument is not a subroutine name

(F) The argument to `exists` for `exists &sub` must be a subroutine name, and not a subroutine call. `exists &sub()` will generate this error.

Exiting eval via %s

(W exiting) You are exiting an eval by unconventional means, such as a goto, or a loop control statement.

Exiting format via %s

(W exiting) You are exiting a format by unconventional means, such as a goto, or a loop control statement.

Exiting pseudo-block via %s

(W exiting) You are exiting a rather special block construct (like a sort block or subroutine) by unconventional means, such as a goto, or a loop control statement. See ⟨undefined⟩ [perlfunc sort], page ⟨undefined⟩.

Exiting subroutine via %s

(W exiting) You are exiting a subroutine by unconventional means, such as a goto, or a loop control statement.

Exiting substitution via %s

(W exiting) You are exiting a substitution by unconventional means, such as a return, a goto, or a loop control statement.

Expecting close bracket in regex; marked by <– HERE in m/%s/

(F) You wrote something like

```
 (?13
```

to denote a capturing group of the form [(?*PARNO*)], page 1012, but omitted the ")".

Expecting '(?flags:(?[...' in regex; marked by <– HERE in m/%s/

(F) The (?[...]) extended character class regular expression construct only allows character classes (including character class escapes like \d), operators, and parentheses. The one exception is (?flags:...) containing at least one flag and exactly one (?[...]) construct. This allows a regular expression containing just (?[...]) to be interpolated. If you see this error message, then you probably have some other (?...) construct inside your character class. See Section 61.2.3.9 [perlrecharclass Extended Bracketed Character Classes], page 1073.

Experimental aliasing via reference not enabled

(F) To do aliasing via references, you must first enable the feature:

```
    no warnings "experimental::refaliasing";
    use feature "refaliasing";
    \$x = \$y;
```

Experimental subroutine signatures not enabled

(F) To use subroutine signatures, you must first enable them:

```
    no warnings "experimental::signatures";
    use feature "signatures";
    sub foo ($left, $right) { ... }
```

Experimental "%s" subs not enabled

(F) To use lexical subs, you must first enable them:

```
    no warnings 'experimental::lexical_subs';
    use feature 'lexical_subs';
    my sub foo { ... }
```

Explicit blessing to " (assuming package main)

(W misc) You are blessing a reference to a zero length string. This has the effect of blessing the reference into the package main. This is usually not what you want. Consider providing a default target package, e.g. bless($ref, $p `||` 'MyPackage');

%s: Expression syntax

(A) You've accidentally run your script through **csh** instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

%s failed–call queue aborted

(F) An untrapped exception was raised while executing a UNITCHECK, CHECK, INIT, or END subroutine. Processing of the remainder of the queue of such routines has been prematurely ended.

False [] range "%s" in regex; marked by <– HERE in m/%s/

(W regexp)(F) A character class range must start and end at a literal character, not another character class like `\d` or `[:alpha:]`. The `"-"` in your false range is interpreted as a literal `"-"`. In a `(?[...])` construct, this is an error, rather than a warning. Consider quoting the `"-"`, `"\-"`. The <– HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Fatal VMS error (status=%d) at %s, line %d

(P) An error peculiar to VMS. Something untoward happened in a VMS system service or RTL routine; Perl's exit status should provide more details. The filename in "at %s" and the line number in "line %d" tell you which section of the Perl source code is distressed.

fcntl is not implemented

(F) Your machine apparently doesn't implement fcntl(). What is this, a PDP-11 or something?

FETCHSIZE returned a negative value

(F) A tied array claimed to have a negative number of elements, which is not possible.

Field too wide in 'u' format in pack

(W pack) Each line in an uuencoded string starts with a length indicator which can't encode values above 63. So there is no point in asking for a line length bigger than that. Perl will behave as if you specified `u63` as the format.

Filehandle %s opened only for input

(W io) You tried to write on a read-only filehandle. If you intended it to be a read-write filehandle, you needed to open it with `"+<"` or `"+>"` or `"+>>"` instead of with `"<"` or nothing. If you intended only to write the file, use `">"` or `">>"`. See ⟨undefined⟩ [perlfunc open], page ⟨undefined⟩.

Filehandle %s opened only for output

(W io) You tried to read from a filehandle opened only for writing, If you intended it to be a read/write filehandle, you needed to open it with `"+<"` or `"+>"` or `"+>>"` instead of with `">"`. If you intended only to read from the file, use `"<"`. See ⟨undefined⟩ [perlfunc open], page ⟨undefined⟩. Another possibility is that you attempted to open filedescriptor 0 (also known as STDIN) for output (maybe you closed STDIN earlier?).

Filehandle %s reopened as %s only for input

(W io) You opened for reading a filehandle that got the same filehandle id as STDOUT or STDERR. This occurred because you closed STDOUT or STDERR previously.

Filehandle STDIN reopened as %s only for output

(W io) You opened for writing a filehandle that got the same filehandle id as STDIN. This occurred because you closed STDIN previously.

Final $ should be \$ or $name

(F) You must now decide whether the final $ in a string was meant to be a literal dollar sign, or was meant to introduce a variable name that happens to be missing. So you have to put either the backslash or the name.

flock() on closed filehandle %s

(W closed) The filehandle you're attempting to flock() got itself closed some time before now. Check your control flow. flock() operates on filehandles. Are you attempting to call flock() on a dirhandle by the same name?

Format not terminated

(F) A format must be terminated by a line with a solitary dot. Perl got to the end of your file without finding such a line.

Format %s redefined

(W redefine) You redefined a format. To suppress this warning, say

```
{
    no warnings 'redefine';
    eval "format NAME =...";
}
```

Found = in conditional, should be ==

(W syntax) You said

```
if ($foo = 123)
```

when you meant

```
if ($foo == 123)
```

(or something like that).

%s found where operator expected

(S syntax) The Perl lexer knows whether to expect a term or an operator. If it sees what it knows to be a term when it was expecting to see an operator, it gives you this warning. Usually it indicates that an operator or delimiter was omitted, such as a semicolon.

gdbm store returned %d, errno %d, key "%s"

(S) A warning from the GDBM_File extension that a store failed.

gethostent not implemented

(F) Your C library apparently doesn't implement gethostent(), probably because if it did, it'd feel morally obligated to return every hostname on the Internet.

get%sname() on closed socket %s

(W closed) You tried to get a socket or peer socket name on a closed socket. Did you forget to check the return value of your socket() call?

getpwnam returned invalid UIC %#o for user "%s"

(S) A warning peculiar to VMS. The call to `sys$getuai` underlying the `getpwnam` operator returned an invalid UIC.

getsockopt() on closed socket %s

(W closed) You tried to get a socket option on a closed socket. Did you forget to check the return value of your socket() call? See ⟨undefined⟩ [perlfunc getsockopt], page ⟨undefined⟩.

given is experimental

(S experimental::smartmatch) `given` depends on smartmatch, which is experimental, so its behavior may change or even be removed in any future release of perl. See the explanation under Section 74.2.16 [perlsyn Experimental Details on given and when], page 1263.

Global symbol "%s" requires explicit package name (did you forget to declare "my %s"?)

(F) You've said "use strict" or "use strict vars", which indicates that all variables must either be lexically scoped (using "my" or "state"), declared beforehand using "our", or explicitly qualified to say which package the global variable is in (using "::").

glob failed (%s)

(S glob) Something went wrong with the external program(s) used for `glob` and `<*.c>`. Usually, this means that you supplied a `glob` pattern that caused the external program to fail and exit with a nonzero status. If the message indicates that the abnormal exit resulted in a coredump, this may also mean that your csh (C shell) is broken. If so, you should change all of the csh-related variables in config.sh: If you have tcsh, make the variables refer to it as if it were csh (e.g. `full_csh='/usr/bin/tcsh'`); otherwise, make them all empty (except that `d_csh` should be `'undef'`) so that Perl will think csh is missing. In either case, after editing config.sh, run `./Configure -S` and rebuild Perl.

Glob not terminated

(F) The lexer saw a left angle bracket in a place where it was expecting a term, so it's looking for the corresponding right angle bracket, and not finding it. Chances are you left some needed parentheses out earlier in the line, and you really meant a "less than".

gmtime(%f) failed

(W overflow) You called `gmtime` with a number that it could not handle: too large, too small, or NaN. The returned value is `undef`.

gmtime(%f) too large

(W overflow) You called `gmtime` with a number that was larger than it can reliably handle and `gmtime` probably returned the wrong date. This warning is also triggered with NaN (the special not-a-number value).

gmtime(%f) too small

(W overflow) You called `gmtime` with a number that was smaller than it can reliably handle and `gmtime` probably returned the wrong date.

Got an error from DosAllocMem

(P) An error peculiar to OS/2. Most probably you're using an obsolete version of Perl, and this should not happen anyway.

goto must have label

(F) Unlike with "next" or "last", you're not allowed to goto an unspecified destination. See ⟨undefined⟩ [perlfunc goto], page ⟨undefined⟩.

Goto undefined subroutine%s

(F) You tried to call a subroutine with `goto &sub` syntax, but the indicated subroutine hasn't been defined, or if it was, it has since been undefined.

Group name must start with a non-digit word character in regex; marked by <– HERE in m/%s/

(F) Group names must follow the rules for perl identifiers, meaning they must start with a non-digit word character. A common cause of this error is using (?&0) instead of (?0). See Section 58.1 [perlre NAME], page 989.

()-group starts with a count

(F) A ()-group started with a count. A count is supposed to follow something: a template character or a ()-group. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

%s had compilation errors.

(F) The final summary message when a `perl -c` fails.

Had to create %s unexpectedly

(S internal) A routine asked for a symbol from a symbol table that ought to have existed already, but for some reason it didn't, and had to be created on an emergency basis to prevent a core dump.

%s has too many errors

(F) The parser has given up trying to parse the program after 10 errors. Further error messages would likely be uninformative.

Having more than one /%c regexp modifier is deprecated

(D deprecated, regexp) You used the indicated regular expression pattern modifier at least twice in a string of modifiers. It is deprecated to do this with this particular modifier, to allow future extensions to the Perl language.

Hexadecimal float: exponent overflow

(W overflow) The hexadecimal floating point has a larger exponent than the floating point supports.

Hexadecimal float: exponent underflow

(W overflow) The hexadecimal floating point has a smaller exponent than the floating point supports.

Hexadecimal float: internal error

(F) Something went horribly bad in hexadecimal float handling.

Hexadecimal float: mantissa overflow

(W overflow) The hexadecimal floating point literal had more bits in the mantissa (the part between the 0x and the exponent, also known as the fraction or the significand) than the floating point supports.

Hexadecimal float: precision loss

(W overflow) The hexadecimal floating point had internally more digits than could be output. This can be caused by unsupported long double formats, or by 64-bit integers not being available (needed to retrieve the digits under some configurations).

Hexadecimal float: unsupported long double format

(F) You have configured Perl to use long doubles but the internals of the long double format are unknown; therefore the hexadecimal float output is impossible.

Hexadecimal number > 0xffffffff non-portable

(W portable) The hexadecimal number you specified is larger than 2**32-1 (4294967295) and therefore non-portable between systems. See Section 56.1 [perlport NAME], page 951 for more on portability concerns.

Identifier too long

(F) Perl limits identifiers (names for variables, functions, etc.) to about 250 characters for simple names, and somewhat more for compound names (like `$A::B`). You've exceeded Perl's limits. Future versions of Perl are likely to eliminate these arbitrary limitations.

Ignoring zero length \N{} in character class in regex; marked by <– HERE in m/%s/

(W regexp) Named Unicode character escapes (`\N{...}`) may return a zero-length sequence. When such an escape is used in a character class its behavior is not well defined. Check that the correct escape has been used, and the correct charname handler is in scope.

Illegal binary digit %s

(F) You used a digit other than 0 or 1 in a binary number.

Illegal binary digit %s ignored

(W digit) You may have tried to use a digit other than 0 or 1 in a binary number. Interpretation of the binary number stopped before the offending digit.

Illegal character after '_' in prototype for %s : %s

(W illegalproto) An illegal character was found in a prototype declaration. The '_' in a prototype must be followed by a ';', indicating the rest of the parameters are optional, or one of '@' or '%', since those two will accept 0 or more final parameters.

Illegal character \%o (carriage return)

(F) Perl normally treats carriage returns in the program text as it would any other whitespace, which means you should never see this error when Perl was built using standard options. For some reason, your version of Perl appears to have been built without this support. Talk to your Perl administrator.

Illegal character in prototype for %s : %s

(W illegalproto) An illegal character was found in a prototype declaration. Legal characters in prototypes are $, @, %, *, ;, [,], &, \, and `+`. Perhaps you were trying to write a subroutine signature but didn't enable that feature first (`use feature 'signatures'`), so your signature was instead interpreted as a bad prototype.

Illegal declaration of anonymous subroutine

(F) When using the `sub` keyword to construct an anonymous subroutine, you must always specify a block of code. See Section 73.1 [perlsub NAME], page 1216.

Illegal declaration of subroutine %s

(F) A subroutine was not declared correctly. See Section 73.1 [perlsub NAME], page 1216.

Illegal division by zero

(F) You tried to divide a number by 0. Either something was wrong in your logic, or you need to put a conditional in to guard against meaningless input.

Illegal hexadecimal digit %s ignored

(W digit) You may have tried to use a character other than 0 - 9 or A - F, a - f in a hexadecimal number. Interpretation of the hexadecimal number stopped before the illegal character.

Illegal modulus zero

(F) You tried to divide a number by 0 to get the remainder. Most numbers don't take to this kindly.

Illegal number of bits in vec

(F) The number of bits in vec() (the third argument) must be a power of two from 1 to 32 (or 64, if your platform supports that).

Illegal octal digit %s

(F) You used an 8 or 9 in an octal number.

Illegal octal digit %s ignored

(W digit) You may have tried to use an 8 or 9 in an octal number. Interpretation of the octal number stopped before the 8 or 9.

Illegal pattern in regex; marked by <– HERE in m/%s/

(F) You wrote something like

```
(?+foo)
```

The `"+"` is valid only when followed by digits, indicating a capturing group. See [`(?`*`PARNO`*`)`], page 1012.

Illegal suidscript

(F) The script run under suidperl was somehow illegal.

Illegal switch in PERL5OPT: -%c

(X) The PERL5OPT environment variable may only be used to set the following switches: **-[CDIMUdmtw]**.

Ill-formed CRTL environ value "%s"

(W internal) A warning peculiar to VMS. Perl tried to read the CRTL's internal environ array, and encountered an element without the `=` delimiter used to separate keys from values. The element is ignored.

Ill-formed message in prime_env_iter: |%s|

(W internal) A warning peculiar to VMS. Perl tried to read a logical name or CLI symbol definition when preparing to iterate over %ENV, and didn't see the expected delimiter between key and value, so the line was ignored.

(in cleanup) %s

(W misc) This prefix usually indicates that a DESTROY() method raised the indicated exception. Since destructors are usually called by the system at arbitrary points during execution, and often a vast number of times, the warning is issued only once for any number of failures that would otherwise result in the same message being repeated.

Failure of user callbacks dispatched using the `G_KEEPERR` flag could also result in this warning. See Section 7.4.7 [perlcall G_KEEPERR], page 32.

Incomplete expression within '(?[ ])' in regex; marked by <– HERE in m/%s/

(F) There was a syntax error within the `(?[ ])`. This can happen if the expression inside the construct was completely empty, or if there are too many or few operands for the number of operators. Perl is not smart enough to give you a more precise indication as to what is wrong.

Inconsistent hierarchy during C3 merge of class '%s': merging failed on parent '%s'

(F) The method resolution order (MRO) of the given class is not C3-consistent, and you have enabled the C3 MRO for this class. See the C3 documentation in `mro` for more information.

In EBCDIC the v-string components cannot exceed 2147483647

(F) An error peculiar to EBCDIC. Internally, v-strings are stored as Unicode code points, and encoded in EBCDIC as UTF-EBCDIC. The UTF-EBCDIC encoding is limited to code points no larger than 2147483647 (0x7FFFFFFF).

Infinite recursion in regex

(F) You used a pattern that references itself without consuming any input text. You should check the pattern to ensure that recursive patterns either consume text or fail.

Initialization of state variables in list context currently forbidden

(F) Currently the implementation of "state" only permits the initialization of scalar variables in scalar context. Re-write `state ($a) = 42` as `state $a = 42` to change from list to scalar context. Constructions such as `state (@a) = foo()` will be supported in a future perl release.

%%s[%s] in scalar context better written as $%s[%s]

(W syntax) In scalar context, you've used an array index/value slice (indicated by %) to select a single element of an array. Generally it's better to ask for a scalar value (indicated by $). The difference is that `$foo[&bar]` always behaves like a scalar, both in the value it returns and when evaluating its argument, while `%foo[&bar]` provides a list context to its subscript, which can do weird things if you're expecting only one subscript. When called in list context, it also returns the index (what `&bar` returns) in addition to the value.

%%s{%s} in scalar context better written as $%s{%s}

(W syntax) In scalar context, you've used a hash key/value slice (indicated by %) to select a single element of a hash. Generally it's better to ask for a scalar value (indicated by $). The difference is that `$foo{&bar}` always behaves like a scalar, both in the value it returns and when evaluating its argument, while `@foo{&bar}` and provides a list context to its subscript, which can do weird things if you're expecting only one subscript. When called in list context, it also returns the key in addition to the value.

Insecure dependency in %s

(F) You tried to do something that the tainting mechanism didn't like. The tainting mechanism is turned on when you're running setuid or setgid, or when you specify **-T** to turn it on explicitly. The tainting mechanism labels all data that's derived directly or indirectly from the user, who is considered to be unworthy of your trust. If any such data is used in a "dangerous" operation, you get this error. See Section 70.1 [perlsec NAME], page 1198 for more information.

Insecure directory in %s

(F) You can't use system(), exec(), or a piped open in a setuid or setgid script if `$ENV{PATH}` contains a directory that is writable by the world. Also, the PATH must not contain any relative directory. See Section 70.1 [perlsec NAME], page 1198.

Insecure $ENV{%s} while running %s

(F) You can't use system(), exec(), or a piped open in a setuid or setgid script if any of `$ENV{PATH}`, `$ENV{IFS}`, `$ENV{CDPATH}`, `$ENV{ENV}`, `$ENV{BASH_ENV}` or `$ENV{TERM}` are derived from data supplied (or potentially supplied) by the user. The script must set the path to a known value, using trustworthy data. See Section 70.1 [perlsec NAME], page 1198.

Insecure user-defined property %s

(F) Perl detected tainted data when trying to compile a regular expression that contains a call to a user-defined character property function, i.e. `\p{IsFoo}` or `\p{InFoo}`. See Section 81.2.6 [perlunicode User-Defined Character Properties], page 1329 and Section 70.1 [perlsec NAME], page 1198.

Integer overflow in format string for %s

(F) The indexes and widths specified in the format string of `printf()` or `sprintf()` are too large. The numbers must not overflow the size of integers for your architecture.

Integer overflow in %s number

(S overflow) The hexadecimal, octal or binary number you have specified either as a literal or as an argument to hex() or oct() is too big for your architecture, and has been converted to a floating point number. On a 32-bit architecture the largest hexadecimal, octal or binary number representable without overflow is 0xFFFFFFFF, 037777777777, or 0b11111111111111111111111111111111 respectively. Note that Perl transparently promotes all numbers to a floating point representation internally–subject to loss of precision errors in subsequent operations.

Integer overflow in srand

(S overflow) The number you have passed to srand is too big to fit in your architecture's integer representation. The number has been replaced with the largest integer supported (0xFFFFFFFF on 32-bit architectures). This means you may be getting less randomness than you expect, because different random seeds above the maximum will return the same sequence of random numbers.

Integer overflow in version
Integer overflow in version %d

(W overflow) Some portion of a version initialization is too large for the size of integers for your architecture. This is not a warning because there is no rational reason for a version to try and use an element larger than typically 2**32. This is usually caused by trying to use some odd mathematical operation as a version, like 100/9.

Internal disaster in regex; marked by <– HERE in m/%s/

(P) Something went badly wrong in the regular expression parser. The <– HERE shows whereabouts in the regular expression the problem was discovered.

Internal inconsistency in tracking vforks

(S) A warning peculiar to VMS. Perl keeps track of the number of times you've called `fork` and `exec`, to determine whether the current call to `exec` should affect the current script or a subprocess (see [perlvms exec LIST], page 1419). Somehow, this count has become scrambled, so Perl is making a guess and treating this `exec` as a request to terminate the Perl script and execute the specified command.

internal %<num>p might conflict with future printf extensions

(S internal) Perl's internal routine that handles `printf` and `sprintf` formatting follows a slightly different set of rules when called from C or XS code. Specifically, formats consisting of digits followed by "p" (e.g., "%7p") are reserved for future use. If you see this message, then an XS module tried to call that routine with one such reserved format.

Internal urp in regex; marked by <– HERE in m/%s/

(P) Something went badly awry in the regular expression parser. The <– HERE shows whereabouts in the regular expression the problem was discovered.

%s (...) interpreted as function

(W syntax) You've run afoul of the rule that says that any list operator followed by parentheses turns into a function, with all the list operators arguments found inside the parentheses. See Section 48.2.2 [perlop Terms and List Operators (Leftward)], page 799.

In '(?...)', the '(' and '?' must be adjacent in regex; marked by <-- HERE in m/%s/

(F) The two-character sequence `"(?"` in this context in a regular expression pattern should be an indivisible token, with nothing intervening between the `"("` and the `"?"`, but you separated them with whitespace.

Invalid %s attribute: %s

(F) The indicated attribute for a subroutine or variable was not recognized by Perl or by a user-supplied handler. See `attributes`.

Invalid %s attributes: %s

(F) The indicated attributes for a subroutine or variable were not recognized by Perl or by a user-supplied handler. See `attributes`.

Invalid character in charnames alias definition; marked by <-- HERE in '%s

(F) You tried to create a custom alias for a character name, with the `:alias` option to `use charnames` and the specified character in the indicated name isn't valid. See Section "CUSTOM ALIASES" in `charnames`.

Invalid \0 character in %s for %s: %s\0%s

(W syscalls) Embedded `\0` characters in pathnames or other system call arguments produce a warning as of 5.20. The parts after the `\0` were formerly ignored by system calls.

Invalid character in \N{...}; marked by <-- HERE in \N{%s}

(F) Only certain characters are valid for character names. The indicated one isn't. See Section "CUSTOM ALIASES" in `charnames`.

Invalid conversion in %s: "%s"

(W printf) Perl does not understand the given format conversion. See ⟨undefined⟩ [perlfunc sprintf], page ⟨undefined⟩.

Invalid escape in the specified encoding in regex; marked by <-- HERE in m/%s/

(W regexp)(F) The numeric escape (for example `\xHH`) of value < 256 didn't correspond to a single character through the conversion from the encoding specified by the encoding pragma. The escape was replaced with REPLACEMENT CHARACTER (U+FFFD) instead, except within `(?[ ])`, where it is a fatal error. The <-- HERE shows whereabouts in the regular expression the escape was discovered.

Invalid hexadecimal number in \N{U+...}
Invalid hexadecimal number in \N{U+...} in regex; marked by <-- HERE in m/%s/

(F) The character constant represented by `...` is not a valid hexadecimal number. Either it is empty, or you tried to use a character other than 0 - 9 or A - F, a - f in a hexadecimal number.

Invalid module name %s with -%c option: contains single ':'

(F) The module argument to perl's **-m** and **-M** command-line options cannot contain single colons in the module name, but only in the arguments after `"="`. In other words, **-MFoo::Bar=:baz** is ok, but **-MFoo:Bar=baz** is not.

Invalid mro name: '%s'

(F) You tried to `mro::set_mro("classname", "foo")` or `use mro 'foo'`, where `foo` is not a valid method resolution order (MRO). Currently, the only valid ones supported are `dfs` and `c3`, unless you have loaded a module that is a MRO plugin. See `mro` and Section 43.1 [perlmroapi NAME], page 758.

Invalid negative number (%s) in chr

(W utf8) You passed a negative number to `chr`. Negative numbers are not valid character numbers, so it returns the Unicode replacement character (U+FFFD).

invalid option -D%c, use -D" to see choices

(S debugging) Perl was called with invalid debugger flags. Call perl with the **-D** option with no flags to see the list of acceptable values. See also [perlrun -Dletters], page 1181.

Invalid quantifier in {,} in regex; marked by <– HERE in m/%s/

(F) The pattern looks like a {min,max} quantifier, but the min or max could not be parsed as a valid number - either it has leading zeroes, or it represents too big a number to cope with. The <– HERE shows where in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Invalid [] range "%s" in regex; marked by <– HERE in m/%s/

(F) The range specified in a character class had a minimum character greater than the maximum character. One possibility is that you forgot the `{}` from your ending `\x{}` - `\x` without the curly braces can go only up to `ff`. The <– HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Invalid range "%s" in transliteration operator

(F) The range specified in the tr/// or y/// operator had a minimum character greater than the maximum character. See Section 48.1 [perlop NAME], page 798.

Invalid separator character %s in attribute list

(F) Something other than a colon or whitespace was seen between the elements of an attribute list. If the previous attribute had a parenthesised parameter list, perhaps that list was terminated too soon. See `attributes`.

Invalid separator character %s in PerlIO layer specification %s

(W layer) When pushing layers onto the Perl I/O system, something other than a colon or whitespace was seen between the elements of a layer list. If the previous attribute had a parenthesised parameter list, perhaps that list was terminated too soon.

Invalid strict version format (%s)

(F) A version number did not meet the "strict" criteria for versions. A "strict" version number is a positive decimal number (integer or decimal-fraction) without exponentiation or else a dotted-decimal v-string with a leading 'v' character and at least three components. The parenthesized text indicates which criteria were not met. See the `version` module for more details on allowed version formats.

Invalid type '%s' in %s

(F) The given character is not a valid pack or unpack type. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

(W) The given character is not a valid pack or unpack type but used to be silently ignored.

Invalid version format (%s)

(F) A version number did not meet the "lax" criteria for versions. A "lax" version number is a positive decimal number (integer or decimal-fraction) without exponentiation or else a dotted-decimal v-string. If the v-string has fewer than three components, it must have a leading 'v' character. Otherwise, the leading 'v' is optional. Both decimal and dotted-decimal versions may have a trailing "alpha" component separated by an underscore character after a fractional or dotted-decimal component. The parenthesized text indicates which criteria were not met. See the `version` module for more details on allowed version formats.

Invalid version object

(F) The internal structure of the version object was invalid. Perhaps the internals were modified directly in some way or an arbitrary reference was blessed into the "version" class.

In '(*VERB...)', the '(' and '*' must be adjacent in regex; marked by <– HERE in m/%s/

(F) The two-character sequence `"(*"` in this context in a regular expression pattern should be an indivisible token, with nothing intervening between the `"("` and the `"*"`, but you separated them.

ioctl is not implemented

(F) Your machine apparently doesn't implement ioctl(), which is pretty strange for a machine that supports C.

ioctl() on unopened %s

(W unopened) You tried ioctl() on a filehandle that was never opened. Check your control flow and number of arguments.

IO layers (like '%s') unavailable

(F) Your Perl has not been configured to have PerlIO, and therefore you cannot use IO layers. To have PerlIO, Perl must be configured with 'useperlio'.

IO::Socket::atmark not implemented on this architecture

(F) Your machine doesn't implement the sockatmark() functionality, neither as a system call nor an ioctl call (SIOCATMARK).

'%s' is an unknown bound type in regex; marked by <– HERE in m/%s/

(F) You used `\b{...}` or `\B{...}` and the `...` is not known to Perl. The current valid ones are given in [perlrebackslash \b{}, \b, \B{}, \B], page 1054.

"%s" is more clearly written simply as "%s" in regex; marked by <– HERE in m/%s/

(W regexp) (only under `use re 'strict'` or within `(?[...])`)

You specified a character that has the given plainer way of writing it, and which is also portable to platforms running with different character sets.

$* is no longer supported

(D deprecated, syntax) The special variable `$*`, deprecated in older perls, has been removed as of 5.10.0 and is no longer supported. In previous versions of perl the use of `$*` enabled or disabled multi-line matching within a string.

Instead of using `$*` you should use the `/m` (and maybe `/s`) regexp modifiers. You can enable `/m` for a lexical scope (even a whole file) with `use re '/m'`. (In older versions: when `$*` was set to a true value then all regular expressions behaved as if they were written using `/m`.)

$# is no longer supported

(D deprecated, syntax) The special variable `$#`, deprecated in older perls, has been removed as of 5.10.0 and is no longer supported. You should use the printf/sprintf functions instead.

'%s' is not a code reference

(W overload) The second (fourth, sixth, ...) argument of overload::constant needs to be a code reference. Either an anonymous subroutine, or a reference to a subroutine.

'%s' is not an overloadable type

(W overload) You tried to overload a constant type the overload package is unaware of.

-i used with no filenames on the command line, reading from STDIN

(S inplace) The `-i` option was passed on the command line, indicating that the script is intended to edit files in place, but no files were given. This is usually a mistake, since editing STDIN in place doesn't make sense, and can be confusing because it can make perl look like it is hanging when it is really just trying to read from STDIN. You should either pass a filename to edit, or remove `-i` from the command line. See Section 69.1 [perlrun NAME], page 1176 for more details.

Junk on end of regexp in regex m/%s/

(P) The regular expression parser is confused.

keys on reference is experimental

(S experimental::autoderef) `keys` with a scalar argument is experimental and may change or be removed in a future Perl version. If you want to take the risk of using this feature, simply disable this warning:

```
no warnings "experimental::autoderef";
```

Label not found for "last %s"

(F) You named a loop to break out of, but you're not currently in a loop of that name, not even if you count where you were called from. See [perlfunc last], page 399.

Label not found for "next %s"

(F) You named a loop to continue, but you're not currently in a loop of that name, not even if you count where you were called from. See [perlfunc last], page 399.

Label not found for "redo %s"

(F) You named a loop to restart, but you're not currently in a loop of that name, not even if you count where you were called from. See [perlfunc last], page 399.

leaving effective %s failed

(F) While under the `use filetest` pragma, switching the real and effective uids or gids failed.

length/code after end of string in unpack

(F) While unpacking, the string buffer was already used up when an unpack length/code combination tried to obtain more data. This results in an undefined value for the length. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

length() used on %s (did you mean "scalar(%s)"?)

(W syntax) You used length() on either an array or a hash when you probably wanted a count of the items.

Array size can be obtained by doing:

```
    scalar(@array);
```

The number of items in a hash can be obtained by doing:

```
    scalar(keys %hash);
```

Lexing code attempted to stuff non-Latin-1 character into Latin-1 input

(F) An extension is attempting to insert text into the current parse (using Section “lex_stuff_pvn” in `perlapi` or similar), but tried to insert a character that couldn’t be part of the current input. This is an inherent pitfall of the stuffing mechanism, and one of the reasons to avoid it. Where it is necessary to stuff, stuffing only plain ASCII is recommended.

Lexing code internal error (%s)

(F) Lexing code supplied by an extension violated the lexer’s API in a detectable way.

listen() on closed socket %s

(W closed) You tried to do a listen on a closed socket. Did you forget to check the return value of your socket() call? See ⟨undefined⟩ [perlfunc listen], page ⟨undefined⟩.

List form of piped open not implemented

(F) On some platforms, notably Windows, the three-or-more-arguments form of `open` does not support pipes, such as `open($pipe, ’|-’, @args)`. Use the two-argument `open($pipe, ’|prog arg1 arg2...’)` form instead.

%s: loadable library and perl binaries are mismatched (got handshake key %p, needed %p)

(P) A dynamic loading library `.so` or `.dll` was being loaded into the process that was built against a different build of perl than the said library was compiled against. Reinstalling the XS module will likely fix this error.

Locale ’%s’ may not work well.%s

(W locale) You are using the named locale, which is a non-UTF-8 one, and which perl has determined is not fully compatible with what it can handle. The second `%s` gives a reason.

By far the most common reason is that the locale has characters in it that are represented by more than one byte. The only such locales that Perl can handle are the UTF-8 locales. Most likely the specified locale is a non-UTF-8 one for an East Asian language such as Chinese or Japanese. If the locale is a superset of ASCII, the ASCII portion of it may work in Perl.

Some essentially obsolete locales that aren't supersets of ASCII, mainly those in ISO 646 or other 7-bit locales, such as ASMO 449, can also have problems, depending on what portions of the ASCII character set get changed by the locale and are also used by the program. The warning message lists the determinable conflicting characters.

Note that not all incompatibilities are found.

If this happens to you, there's not much you can do except switch to use a different locale or use `Encode` to translate from the locale into UTF-8; if that's impracticable, you have been warned that some things may break.

This message is output once each time a bad locale is switched into within the scope of `use locale`, or on the first possibly-affected operation if the `use locale` inherits a bad one. It is not raised for any operations from the `POSIX` module.

localtime(%f) failed

(W overflow) You called `localtime` with a number that it could not handle: too large, too small, or NaN. The returned value is `undef`.

localtime(%f) too large

(W overflow) You called `localtime` with a number that was larger than it can reliably handle and `localtime` probably returned the wrong date. This warning is also triggered with NaN (the special not-a-number value).

localtime(%f) too small

(W overflow) You called `localtime` with a number that was smaller than it can reliably handle and `localtime` probably returned the wrong date.

Lookbehind longer than %d not implemented in regex m/%s/

(F) There is currently a limit on the length of string which lookbehind can handle. This restriction may be eased in a future release.

Lost precision when %s %f by 1

(W imprecision) The value you attempted to increment or decrement by one is too large for the underlying floating point representation to store accurately, hence the target of `++` or `--` is unchanged. Perl issues this warning because it has already switched from integers to floating point when values are too large for integers, and now even floating point is insufficient. You may wish to switch to using `Math-BigInt` explicitly.

lstat() on filehandle%s

(W io) You tried to do an lstat on a filehandle. What did you mean by that? lstat() makes sense only on filenames. (Perl did a fstat() instead on the filehandle.)

lvalue attribute %s already-defined subroutine

(W misc) Although `attributes` allows this, turning the lvalue attribute on or off on a Perl subroutine that is already defined does not always work properly. It may or may not do what you want, depending on what code is inside the subroutine, with exact details subject to change between Perl versions. Only do this if you really know what you are doing.

lvalue attribute ignored after the subroutine has been defined

(W misc) Using the `:lvalue` declarative syntax to make a Perl subroutine an lvalue subroutine after it has been defined is not permitted. To make the subroutine an lvalue subroutine, add the lvalue attribute to the definition, or put the `sub foo :lvalue;` declaration before the definition.

See also `attributes`.

Magical list constants are not supported

(F) You assigned a magical array to a stash element, and then tried to use the subroutine from the same slot. You are asking Perl to do something it cannot do, details subject to change between Perl versions.

Malformed integer in [] in pack

(F) Between the brackets enclosing a numeric repeat count only digits are permitted. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

Malformed integer in [] in unpack

(F) Between the brackets enclosing a numeric repeat count only digits are permitted. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

Malformed PERLLIB_PREFIX

(F) An error peculiar to OS/2. PERLLIB_PREFIX should be of the form

```
prefix1;prefix2
```

or prefix1 prefix2

with nonempty prefix1 and prefix2. If `prefix1` is indeed a prefix of a builtin library search path, prefix2 is substituted. The error may appear if components are not found, or are too long. See "PERLLIB_PREFIX" in `perlos2`.

Malformed prototype for %s: %s

(F) You tried to use a function with a malformed prototype. The syntax of function prototypes is given a brief compile-time check for obvious errors like invalid characters. A more rigorous check is run when the function is called. Perhaps the function's author was trying to write a subroutine signature but didn't enable that feature first (`use feature 'signatures'`), so the signature was instead interpreted as a bad prototype.

Malformed UTF-8 character (%s)

(S utf8)(F) Perl detected a string that didn't comply with UTF-8 encoding rules, even though it had the UTF8 flag on.

One possible cause is that you set the UTF8 flag yourself for data that you thought to be in UTF-8 but it wasn't (it was for example legacy 8-bit data). To guard against this, you can use Encode::decode_utf8.

If you use the `:encoding(UTF-8)` PerlIO layer for input, invalid byte sequences are handled gracefully, but if you use `:utf8`, the flag is set without validating the data, possibly resulting in this error message.

See also Section "Handling Malformed Data" in `Encode`.

Malformed UTF-8 character immediately after '%s'

(F) You said `use utf8`, but the program file doesn't comply with UTF-8 encoding rules. The message prints out the properly encoded characters just before the first bad one. If `utf8` warnings are enabled, a warning is generated that gives more details about the type of malformation.

Malformed UTF-8 returned by \N{%s} immediately after '%s'

(F) The charnames handler returned malformed UTF-8.

Malformed UTF-8 string in '%c' format in unpack

(F) You tried to unpack something that didn't comply with UTF-8 encoding rules and perl was unable to guess how to make more progress.

Malformed UTF-8 string in pack

(F) You tried to pack something that didn't comply with UTF-8 encoding rules and perl was unable to guess how to make more progress.

Malformed UTF-8 string in unpack

(F) You tried to unpack something that didn't comply with UTF-8 encoding rules and perl was unable to guess how to make more progress.

Malformed UTF-16 surrogate

(F) Perl thought it was reading UTF-16 encoded character data but while doing it Perl met a malformed Unicode surrogate.

Mandatory parameter follows optional parameter

(F) In a subroutine signature, you wrote something like "$a = undef, $b", making an earlier parameter optional and a later one mandatory. Parameters are filled from left to right, so it's impossible for the caller to omit an earlier one and pass a later one. If you want to act as if the parameters are filled from right to left, declare the rightmost optional and then shuffle the parameters around in the subroutine's body.

Matched non-Unicode code point 0x%X against Unicode property; may not be portable

(S non_unicode) Perl allows strings to contain a superset of Unicode code points; each code point may be as large as what is storable in an unsigned integer on your system, but these may not be accepted by other languages/systems. This message occurs when you matched a string containing such a code point against a regular expression pattern, and the code point was matched against a Unicode property, `\p{...}` or `\P{...}`. Unicode properties are only defined on Unicode code points, so the result of this match is undefined by Unicode, but Perl (starting in v5.20) treats non-Unicode code points as if they were typical unassigned Unicode ones, and matched this one accordingly. Whether a given property matches these code points or not is specified in Section "Properties accessible through \p{} and \P{}" in `perluniprops`.

This message is suppressed (unless it has been made fatal) if it is immaterial to the results of the match if the code point is Unicode or not. For example, the property `\p{ASCII_Hex_Digit}` only can match the 22 characters `[0-9A-Fa-f]`, so obviously all other code points, Unicode or not, won't match it. (And `\P{ASCII_Hex_Digit}` will match every code point except these 22.)

Getting this message indicates that the outcome of the match arguably should have been the opposite of what actually happened. If you think that is the case, you may wish to make the `non_unicode` warnings category fatal; if you agree with Perl's decision, you may wish to turn off this category.

See Section 81.2.12 [perlunicode Beyond Unicode code points], page 1337 for more information.

%s matches null string many times in regex; marked by <– HERE in m/%s/

(W regexp) The pattern you've specified would be an infinite loop if the regular expression engine didn't specifically check for that. The <– HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Maximal count of pending signals (%u) exceeded

(F) Perl aborted due to too high a number of signals pending. This usually indicates that your operating system tried to deliver signals too fast (with a very high priority), starving the perl process from resources it would need to reach a point where it can process signals safely. (See Section 36.3.2 [perlipc Deferred Signals (Safe Signals)], page 671.)

"%s" may clash with future reserved word

(W) This warning may be due to running a perl5 script through a perl4 interpreter, especially if the word that is being warned about is "use" or "my".

'%' may not be used in pack

(F) You can't pack a string by supplying a checksum, because the checksumming process loses information, and you can't go the other way. See ⟨undefined⟩ [perlfunc unpack], page ⟨undefined⟩.

Method for operation %s not found in package %s during blessing

(F) An attempt was made to specify an entry in an overloading table that doesn't resolve to a valid subroutine. See `overload`.

Method %s not permitted

See Server error.

Might be a runaway multi-line %s string starting on line %d

(S) An advisory indicating that the previous error may have been caused by a missing delimiter on a string or pattern, because it eventually ended earlier on the current line.

Misplaced _ in number

(W syntax) An underscore (underbar) in a numeric constant did not separate two digits.

Missing argument in %s

(W missing) You called a function with fewer arguments than other arguments you supplied indicated would be needed.

Currently only emitted when a printf-type format required more arguments than were supplied, but might be used in the future for other cases where we can statically determine that arguments to functions are missing, e.g. for the ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩ function.

Missing argument to -%c

(F) The argument to the indicated command line switch must follow immediately after the switch, without intervening spaces.

Missing braces on \N{}
Missing braces on \N{} in regex; marked by <– HERE in m/%s/

(F) Wrong syntax of character name literal `\N{charname}` within double-quotish context. This can also happen when there is a space (or comment) between the `\N` and the `{` in a regex with the `/x` modifier. This modifier does not change the requirement that the brace immediately follow the `\N`.

Missing braces on \o{}

(F) A `\o` must be followed immediately by a `{` in double-quotish context.

Missing comma after first argument to %s function

(F) While certain functions allow you to specify a filehandle or an "indirect object" before the argument list, this ain't one of them.

Missing command in piped open

(W pipe) You used the `open(FH, "| command")` or `open(FH, "command |")` construction, but the command was missing or blank.

Missing control char name in \c

(F) A double-quoted string ended with "\c", without the required control character name.

Missing ']' in prototype for %s : %s

(W illegalproto) A grouping was started with `[` but never closed with `]`.

Missing name in "%s sub"

(F) The syntax for lexically scoped subroutines requires that they have a name with which they can be found.

Missing $ on loop variable

(F) Apparently you've been programming in **csh** too much. Variables are always mentioned with the $ in Perl, unlike in the shells, where it can vary from one line to the next.

(Missing operator before %s?)

(S syntax) This is an educated guess made in conjunction with the message "%s found where operator expected". Often the missing operator is a comma.

Missing or undefined argument to require

(F) You tried to call require with no argument or with an undefined value as an argument. Require expects either a package name or a file-specification as an argument. See [perlfunc require], page 437.

Missing right brace on \%c{} in regex; marked by <– HERE in m/%s/

(F) Missing right brace in `\x{...}`, `\p{...}`, `\P{...}`, or `\N{...}`.

Missing right brace on \N{}
Missing right brace on \N{} or unescaped left brace after \N

(F) `\N` has two meanings.

The traditional one has it followed by a name enclosed in braces, meaning the character (or sequence of characters) given by that name. Thus `\N{ASTERISK}` is another way of writing `*`, valid in both double-quoted strings and regular expression patterns. In patterns, it doesn't have the meaning an unescaped `*` does.

Starting in Perl 5.12.0, `\N` also can have an additional meaning (only) in patterns, namely to match a non-newline character. (This is short for `[^\n]`, and like `.` but is not affected by the `/s` regex modifier.)

This can lead to some ambiguities. When `\N` is not followed immediately by a left brace, Perl assumes the `[^\n]` meaning. Also, if the braces form a valid quantifier such as `\N{3}` or `\N{5,}`, Perl assumes that this means to match the given quantity of non-newlines (in these examples, 3; and 5 or more, respectively). In all other case, where there is a `\N{` and a matching `}`, Perl assumes that a character name is desired.

However, if there is no matching `}`, Perl doesn't know if it was mistakenly omitted, or if `[^\n]{` was desired, and raises this error. If you meant the former, add the right brace; if you meant the latter, escape the brace with a backslash, like so: `\N\{`

Missing right curly or square bracket

(F) The lexer counted more opening curly or square brackets than closing ones. As a general rule, you'll find it's missing near the place you were last editing.

(Missing semicolon on previous line?)

(S syntax) This is an educated guess made in conjunction with the message "%s found where operator expected". Don't automatically put a semicolon on the previous line just because you saw this message.

Modification of a read-only value attempted

(F) You tried, directly or indirectly, to change the value of a constant. You didn't, of course, try "2 = 1", because the compiler catches that. But an easy way to do the same thing is:

```
sub mod { $_[0] = 1 }
mod(2);
```

Another way is to assign to a substr() that's off the end of the string.

Yet another way is to assign to a `foreach` loop *VAR* when *VAR* is aliased to a constant in the look *LIST*:

```
$x = 1;
foreach my $n ($x, 2) {
    $n *= 2; # modifies the $x, but fails on attempt to
}            # modify the 2
```

Modification of non-creatable array value attempted, %s

(F) You tried to make an array value spring into existence, and the subscript was probably negative, even counting from end of the array backwards.

Modification of non-creatable hash value attempted, %s

(P) You tried to make a hash value spring into existence, and it couldn't be created for some peculiar reason.

Module name must be constant

(F) Only a bare module name is allowed as the first argument to a `"use"`.

Module name required with -%c option

(F) The `-M` or `-m` options say that Perl should load some module, but you omitted the name of the module. Consult Section 69.1 [perlrun NAME], page 1176 for full details about `-M` and `-m`.

More than one argument to '%s' open

(F) The `open` function has been asked to open multiple files. This can happen if you are trying to open a pipe to a command that takes a list of arguments, but have forgotten to specify a piped open mode. See ⟨undefined⟩ [perlfunc open], page ⟨undefined⟩ for details.

mprotect for COW string %p %u failed with %d

(S) You compiled perl with **-D**PERL_DEBUG_READONLY_COW (see Section 28.3.17 [perlguts Copy on Write], page 527), but a shared string buffer could not be made read-only.

mprotect for %p %u failed with %d

(S) You compiled perl with **-D**PERL_DEBUG_READONLY_OPS (see Section 30.1 [perlhacktips NAME], page 578), but an op tree could not be made read-only.

mprotect RW for COW string %p %u failed with %d

(S) You compiled perl with **-D**PERL_DEBUG_READONLY_COW (see Section 28.3.17 [perlguts Copy on Write], page 527), but a read-only shared string buffer could not be made mutable.

mprotect RW for %p %u failed with %d

(S) You compiled perl with **-D**PERL_DEBUG_READONLY_OPS (see Section 30.1 [perlhacktips NAME], page 578), but a read-only op tree could not be made mutable before freeing the ops.

msg%s not implemented

(F) You don't have System V message IPC on your system.

Multidimensional syntax %s not supported

(W syntax) Multidimensional arrays aren't written like `$foo[1,2,3]`. They're written like `$foo[1][2][3]`, as in C.

'/' must follow a numeric type in unpack

(F) You had an unpack template that contained a '/', but this did not follow some unpack specification producing a numeric value. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

`"my sub"` not yet implemented

(F) Lexically scoped subroutines are not yet implemented. Don't try that yet.

`"my"` subroutine %s can't be in a package

(F) Lexically scoped subroutines aren't in a package, so it doesn't make sense to try to declare one with a package qualifier on the front.

"my %s" used in sort comparison

(W syntax) The package variables $a and $b are used for sort comparisons. You used $a or $b in as an operand to the `<=>` or `cmp` operator inside a sort comparison block, and the variable had earlier been declared as a lexical variable. Either qualify the sort variable with the package name, or rename the lexical variable.

"my" variable %s can't be in a package

(F) Lexically scoped variables aren't in a package, so it doesn't make sense to try to declare one with a package qualifier on the front. Use local() if you want to localize a package variable.

Name "%s::%s" used only once: possible typo

(W once) Typographical errors often show up as unique variable names. If you had a good reason for having a unique name, then just mention it again somehow to suppress the message. The `our` declaration is also provided for this purpose.

NOTE: This warning detects package symbols that have been used only once. This means lexical variables will never trigger this warning. It also means that all of the package variables $c, @c, %c, as well as *c, &c, sub c{}, c(), and c (the filehandle or format) are considered the same; if a program uses $c only once but also uses any of the others it will not trigger this warning. Symbols beginning with an underscore and symbols using special identifiers (q.v. Section 11.1 [perldata NAME], page 70) are exempt from this warning.

Need exactly 3 octal digits in regex; marked by <– HERE in m/%s/

(F) Within `(?[ ])`, all constants interpreted as octal need to be exactly 3 digits long. This helps catch some ambiguities. If your constant is too short, add leading zeros, like

```
(?[ [ \078 ] ])     # Syntax error!
(?[ [ \0078 ] ])    # Works
(?[ [ \007 8 ] ])   # Clearer
```

The maximum number this construct can express is `\777`. If you need a larger one, you need to use Section 60.2.3.7 [\o{}], page 1049 instead. If you meant two separate things, you need to separate them:

```
(?[ [ \7776 ] ])        # Syntax error!
(?[ [ \o{7776} ] ])     # One meaning
(?[ [ \777 6 ] ])       # Another meaning
(?[ [ \777 \006 ] ])    # Still another
```

Negative '/' count in unpack

(F) The length count obtained from a length/code unpack operation was negative. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

Negative length

(F) You tried to do a read/write/send/recv operation with a buffer length that is less than 0. This is difficult to imagine.

Negative offset to vec in lvalue context

(F) When `vec` is called in an lvalue context, the second argument must be greater than or equal to zero.

Negative repeat count does nothing

(W numeric) You tried to execute the Section 48.2.8 [x], page 802 repetition operator fewer than 0 times, which doesn't make sense.

Nested quantifiers in regex; marked by <– HERE in m/%s/

(F) You can't quantify a quantifier without intervening parentheses. So things like ** or +* or ?* are illegal. The <– HERE shows whereabouts in the regular expression the problem was discovered.

Note that the minimal matching quantifiers, `*?`, `+?`, and `??` appear to be nested quantifiers, but aren't. See Section 58.1 [perlre NAME], page 989.

%s never introduced

(S internal) The symbol in question was declared but somehow went out of scope before it could possibly have been used.

next::method/next::can/maybe::next::method cannot find enclosing method

(F) `next::method` needs to be called within the context of a real method in a real package, and it could not find such a context. See `mro`.

\N in a character class must be a named character: \N{...} in regex; marked by <– HERE in m/%s/

(F) The new (as of Perl 5.12) meaning of `\N` as `[^\n]` is not valid in a bracketed character class, for the same reason that `.` in a character class loses its specialness: it matches almost everything, which is probably not what you want.

\N{} in inverted character class or as a range end-point is restricted to one character in regex; marked by <– HERE in m/%s/

(F) Named Unicode character escapes (`\N{...}`) may return a multi-character sequence. Even though a character class is supposed to match just one character of input, perl will match the whole thing correctly, except when the class is inverted (`[^...]`), or the escape is the beginning or final end point of a range. The mathematically logical behavior for what matches when inverting is very different from what people expect, so we have decided to forbid it. Similarly unclear is what should be generated when the `\N{...}` is used as one of the end points of the range, such as in

```
 [\x{41}-\N{ARABIC SEQUENCE YEH WITH HAMZA ABOVE WITH AE}]
```

What is meant here is unclear, as the `\N{...}` escape is a sequence of code points, so this is made an error.

\N{NAME} must be resolved by the lexer in regex; marked by <– HERE in m/%s/

(F) When compiling a regex pattern, an unresolved named character or sequence was encountered. This can happen in any of several ways that bypass the lexer, such as using single-quotish context, or an extra backslash in double-quotish:

```
    $re = '\N{SPACE}';  # Wrong!
    $re = "\\N{SPACE}"; # Wrong!
    /$re/;
```

Instead, use double-quotes with a single backslash:

```
$re = "\N{SPACE}";  # ok
/$re/;
```

The lexer can be bypassed as well by creating the pattern from smaller components:

```
$re = '\N';
/${re}{SPACE}/;     # Wrong!
```

It's not a good idea to split a construct in the middle like this, and it doesn't work here. Instead use the solution above.

Finally, the message also can happen under the `/x` regex modifier when the `\N` is separated by spaces from the `{`, in which case, remove the spaces.

```
/\N {SPACE}/x;      # Wrong!
/\N{SPACE}/x;       # ok
```

No %s allowed while running setuid

(F) Certain operations are deemed to be too insecure for a setuid or setgid script to even be allowed to attempt. Generally speaking there will be another way to do what you want that is, if not secure, at least securable. See Section 70.1 [perlsec NAME], page 1198.

NO-BREAK SPACE in a charnames alias definition is deprecated

(D deprecated) You defined a character name which contained a no-break space character. Change it to a regular space. Usually these names are defined in the `:alias` import argument to `use charnames`, but they could be defined by a translator installed into `$^H{charnames}`. See Section "CUSTOM ALIASES" in `charnames`.

No code specified for -%c

(F) Perl's **-e** and **-E** command-line options require an argument. If you want to run an empty program, pass the empty string as a separate argument or run a program consisting of a single 0 or 1:

```
perl -e ""
perl -e0
perl -e1
```

No comma allowed after %s

(F) A list operator that has a filehandle or "indirect object" is not allowed to have a comma between that and the following arguments. Otherwise it'd be just another one of the arguments.

One possible cause for this is that you expected to have imported a constant to your name space with **use** or **import** while no such importing took place, it may for example be that your operating system does not support that particular constant. Hopefully you did use an explicit import list for the constants you expect to see; please see ⟨undefined⟩ [perlfunc use], page ⟨undefined⟩ and ⟨undefined⟩ [perlfunc import], page ⟨undefined⟩. While an explicit import list would probably have caught this error earlier it naturally does not remedy the fact that your operating system still does not support that constant. Maybe you have a typo in the constants of the symbol import list of **use** or **import** or in the constant name at the line where this error was triggered?

No command into which to pipe on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a '`|`' at the end of the command line, so it doesn't know where you want to pipe the output from this command.

No DB::DB routine defined

(F) The currently executing code was compiled with the **-d** switch, but for some reason the current debugger (e.g. `perl5db.pl` or a `Devel::` module) didn't define a routine to be called at the beginning of each statement.

No dbm on this machine

(P) This is counted as an internal error, because every machine should supply dbm nowadays, because Perl comes with SDBM. See `SDBM_File`.

No DB::sub routine defined

(F) The currently executing code was compiled with the **-d** switch, but for some reason the current debugger (e.g. `perl5db.pl` or a `Devel::` module) didn't define a `DB::sub` routine to be called at the beginning of each ordinary subroutine call.

No directory specified for -I

(F) The **-I** command-line switch requires a directory name as part of the *same* argument. Use **-Ilib**, for instance. **-I lib** won't work.

No error file after 2> or 2>> on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a '2>' or a '2>>' on the command line, but can't find the name of the file to which to write data destined for stderr.

No group ending character '%c' found in template

(F) A pack or unpack template has an opening '(' or '[' without its matching counterpart. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

No input file after < on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a '<' on the command line, but can't find the name of the file from which to read data for stdin.

No next::method '%s' found for %s

(F) `next::method` found no further instances of this method name in the remaining packages of the MRO of this class. If you don't want it throwing an exception, use `maybe::next::method` or `next::can`. See `mro`.

Non-finite repeat count does nothing

(W numeric) You tried to execute the Section 48.2.8 [x], page 802 repetition operator `Inf` (or `-Inf`) or `NaN` times, which doesn't make sense.

Non-hex character in regex; marked by <– HERE in m/%s/

(F) In a regular expression, there was a non-hexadecimal character where a hex one was expected, like

```
(?[ [ \xDG ] ])
(?[ [ \x{DEKA} ] ])
```

Non-octal character in regex; marked by <– HERE in m/%s/

(F) In a regular expression, there was a non-octal character where an octal one was expected, like

```
(?[ [ \o{1278} ] ])
```

Non-octal character '%c'. Resolved as "%s"

(W digit) In parsing an octal numeric constant, a character was unexpectedly encountered that isn't octal. The resulting value is as indicated.

"no" not allowed in expression

(F) The "no" keyword is recognized and executed at compile time, and returns no useful value. See Section 40.1 [perlmod NAME], page 732.

Non-string passed as bitmask

(W misc) A number has been passed as a bitmask argument to select(). Use the vec() function to construct the file descriptor bitmasks for select. See [perlfunc select], page 443.

No output file after > on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a lone '>' at the end of the command line, so it doesn't know where you wanted to redirect stdout.

No output file after > or >> on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a '>' or a '>>' on the command line, but can't find the name of the file to which to write data destined for stdout.

No package name allowed for variable %s in "our"

(F) Fully qualified variable names are not allowed in "our" declarations, because that doesn't make much sense under existing rules. Such syntax is reserved for future extensions.

No Perl script found in input

(F) You called `perl -x`, but no line was found in the file beginning with #! and containing the word "perl".

No setregid available

(F) Configure didn't find anything resembling the setregid() call for your system.

No setreuid available

(F) Configure didn't find anything resembling the setreuid() call for your system.

No such class %s

(F) You provided a class qualifier in a "my", "our" or "state" declaration, but this class doesn't exist at this point in your program.

No such class field "%s" in variable %s of type %s

(F) You tried to access a key from a hash through the indicated typed variable but that key is not allowed by the package of the same type. The indicated package has restricted the set of allowed keys using the `fields` pragma.

No such hook: %s

(F) You specified a signal hook that was not recognized by Perl. Currently, Perl accepts `__DIE__` and `__WARN__` as valid signal hooks.

No such pipe open

(P) An error peculiar to VMS. The internal routine my_pclose() tried to close a pipe which hadn't been opened. This should have been caught earlier as an attempt to close an unopened filehandle.

No such signal: SIG%s

(W signal) You specified a signal name as a subscript to %SIG that was not recognized. Say `kill -l` in your shell to see the valid signal names on your system.

Not a CODE reference

(F) Perl was trying to evaluate a reference to a code value (that is, a subroutine), but found a reference to something else instead. You can use the ref() function to find out what kind of ref it really was. See also Section 62.1 [perlref NAME], page 1077.

Not a GLOB reference

(F) Perl was trying to evaluate a reference to a "typeglob" (that is, a symbol table entry that looks like `*foo`), but found a reference to something else instead. You can use the ref() function to find out what kind of ref it really was. See Section 62.1 [perlref NAME], page 1077.

Not a HASH reference

(F) Perl was trying to evaluate a reference to a hash value, but found a reference to something else instead. You can use the ref() function to find out what kind of ref it really was. See Section 62.1 [perlref NAME], page 1077.

Not an ARRAY reference

(F) Perl was trying to evaluate a reference to an array value, but found a reference to something else instead. You can use the ref() function to find out what kind of ref it really was. See Section 62.1 [perlref NAME], page 1077.

Not an unblessed ARRAY reference

(F) You passed a reference to a blessed array to `push`, `shift` or another array function. These only accept unblessed array references or arrays beginning explicitly with `@`.

Not a SCALAR reference

(F) Perl was trying to evaluate a reference to a scalar value, but found a reference to something else instead. You can use the ref() function to find out what kind of ref it really was. See Section 62.1 [perlref NAME], page 1077.

Not a subroutine reference

(F) Perl was trying to evaluate a reference to a code value (that is, a subroutine), but found a reference to something else instead. You can use the ref() function to find out what kind of ref it really was. See also Section 62.1 [perlref NAME], page 1077.

Not a subroutine reference in overload table

(F) An attempt was made to specify an entry in an overloading table that doesn't somehow point to a valid subroutine. See `overload`.

Not enough arguments for %s

(F) The function requires more arguments than you specified.

Not enough format arguments

(W syntax) A format specified more picture fields than the next line supplied. See Section 24.1 [perlform NAME], page 343.

%s: not found

(A) You've accidentally run your script through the Bourne shell instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

(?[...]) not valid in locale in regex; marked by <– HERE in m/%s/

(F) `(?[...])` cannot be used within the scope of a `use locale` or with an `/l` regular expression modifier, as that would require deferring to run-time the calculation of what it should evaluate to, and it is regex compile-time only.

no UTC offset information; assuming local time is UTC

(S) A warning peculiar to VMS. Perl was unable to find the local timezone offset, so it's assuming that local system time is equivalent to UTC. If it's not, define the logical name `SYS$TIMEZONE_DIFFERENTIAL` to translate to the number of seconds which need to be added to UTC to get local time.

NULL OP IN RUN

(S debugging) Some internal routine called run() with a null opcode pointer.

Null picture in formline

(F) The first argument to formline must be a valid format picture specification. It was found to be empty, which probably means you supplied it an uninitialized value. See Section 24.1 [perlform NAME], page 343.

Null realloc

(P) An attempt was made to realloc NULL.

NULL regexp argument

(P) The internal pattern matching routines blew it big time.

NULL regexp parameter

(P) The internal pattern matching routines are out of their gourd.

Number too long

(F) Perl limits the representation of decimal numbers in programs to about 250 characters. You've exceeded that length. Future versions of Perl are likely to eliminate this arbitrary limitation. In the meantime, try using scientific notation (e.g. "1e6" instead of "1_000_000").

Number with no digits

(F) Perl was looking for a number but found nothing that looked like a number. This happens, for example with `\o{}`, with no number between the braces.

Octal number > 037777777777 non-portable

(W portable) The octal number you specified is larger than 2**32-1 (4294967295) and therefore non-portable between systems. See Section 56.1 [perlport NAME], page 951 for more on portability concerns.

Odd name/value argument for subroutine

(F) A subroutine using a slurpy hash parameter in its signature received an odd number of arguments to populate the hash. It requires the arguments to be paired, with the same number of keys as values. The caller of the subroutine is presumably at fault. Inconveniently, this error will be reported at the location of the subroutine, not that of the caller.

Odd number of arguments for overload::constant

(W overload) The call to overload::constant contained an odd number of arguments. The arguments should come in pairs.

Odd number of elements in anonymous hash

(W misc) You specified an odd number of elements to initialize a hash, which is odd, because hashes come in key/value pairs.

Odd number of elements in hash assignment

(W misc) You specified an odd number of elements to initialize a hash, which is odd, because hashes come in key/value pairs.

Offset outside string

(F)(W layer) You tried to do a read/write/send/recv/seek operation with an offset pointing outside the buffer. This is difficult to imagine. The sole exceptions to this are that zero padding will take place when going past the end of the string when either `sysread()`ing a file, or when seeking past the end of a scalar opened for I/O (in anticipation of future reads and to imitate the behavior with real files).

%s() on unopened %s

(W unopened) An I/O operation was attempted on a filehandle that was never initialized. You need to do an open(), a sysopen(), or a socket() call, or call a constructor from the FileHandle package.

-%s on unopened filehandle %s

(W unopened) You tried to invoke a file test operator on a filehandle that isn't open. Check your control flow. See also [perlfunc -X], page 354.

oops: oopsAV

(S internal) An internal warning that the grammar is screwed up.

oops: oopsHV

(S internal) An internal warning that the grammar is screwed up.

Opening dirhandle %s also as a file

(D io, deprecated) You used open() to associate a filehandle to a symbol (glob or scalar) that already holds a dirhandle. Although legal, this idiom might render your code confusing and is deprecated.

Opening filehandle %s also as a directory

(D io, deprecated) You used opendir() to associate a dirhandle to a symbol (glob or scalar) that already holds a filehandle. Although legal, this idiom might render your code confusing and is deprecated.

Operand with no preceding operator in regex; marked by <– HERE in m/%s/

(F) You wrote something like

```
 (?[ \p{Digit} \p{Thai} ])
```

There are two operands, but no operator giving how you want to combine them.

Operation "%s": no method found, %s

(F) An attempt was made to perform an overloaded operation for which no handler was defined. While some handlers can be autogenerated in terms of other handlers, there is no default handler for any operation, unless the `fallback` overloading key is specified to be true. See `overload`.

Operation "%s" returns its argument for non-Unicode code point 0x%X

(S non_unicode) You performed an operation requiring Unicode rules on a code point that is not in Unicode, so what it should do is not defined. Perl has chosen to have it do nothing, and warn you.

If the operation shown is "ToFold", it means that case-insensitive matching in a regular expression was done on the code point.

If you know what you are doing you can turn off this warning by `no warnings 'non_unicode';`.

Operation "%s" returns its argument for UTF-16 surrogate U+%X

(S surrogate) You performed an operation requiring Unicode rules on a Unicode surrogate. Unicode frowns upon the use of surrogates for anything but storing strings in UTF-16, but rules are (reluctantly) defined for the surrogates, and they are to do nothing for this operation. Because the use of surrogates can be dangerous, Perl warns.

If the operation shown is "ToFold", it means that case-insensitive matching in a regular expression was done on the code point.

If you know what you are doing you can turn off this warning by `no warnings 'surrogate';`.

Operator or semicolon missing before %s

(S ambiguous) You used a variable or subroutine call where the parser was expecting an operator. The parser has assumed you really meant to use an operator, but this is highly likely to be incorrect. For example, if you say "*foo *foo" it will be interpreted as if you said "*foo * 'foo'".

Optional parameter lacks default expression

(F) In a subroutine signature, you wrote something like "$a =", making a named optional parameter without a default value. A nameless optional parameter is permitted to have no default value, but a named one must have a specific default. You probably want "$a = undef".

"our" variable %s redeclared

(W misc) You seem to have already declared the same global once before in the current lexical scope.

Out of memory!

(X) The malloc() function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request. Perl has no option but to exit immediately.

At least in Unix you may be able to get past this by increasing your process datasize limits: in csh/tcsh use `limit` and `limit datasize n` (where `n` is the number of kilobytes) to check the current limits and change them, and in ksh/bash/zsh use `ulimit -a` and `ulimit -d n`, respectively.

Out of memory during %s extend

(X) An attempt was made to extend an array, a list, or a string beyond the largest possible memory allocation.

Out of memory during "large" request for %s

(F) The malloc() function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request. However, the request was judged large enough (compile-time default is 64K), so a possibility to shut down by trapping this error is granted.

Out of memory during request for %s

(X)(F) The malloc() function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request.

The request was judged to be small, so the possibility to trap it depends on the way perl was compiled. By default it is not trappable. However, if compiled for this, Perl may use the contents of `$^M` as an emergency pool after die()ing with this message. In this case the error is trappable *once*, and the error message will include the line and file where the failed request happened.

Out of memory during ridiculously large request

(F) You can't allocate more than 2^31+"small amount" bytes. This error is most likely to be caused by a typo in the Perl program. e.g., `$arr[time]` instead of `$arr[$time]`.

Out of memory for yacc stack

(F) The yacc parser wanted to grow its stack so it could continue parsing, but realloc() wouldn't give it more memory, virtual or otherwise.

'.' outside of string in pack

(F) The argument to a '.' in your template tried to move the working position to before the start of the packed string being built.

'@' outside of string in unpack

(F) You had a template that specified an absolute position outside the string being unpacked. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

'@' outside of string with malformed UTF-8 in unpack

(F) You had a template that specified an absolute position outside the string being unpacked. The string being unpacked was also invalid UTF-8. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

overload arg '%s' is invalid

(W overload) The `overload` pragma was passed an argument it did not recognize. Did you mistype an operator?

Overloaded dereference did not return a reference

(F) An object with an overloaded dereference operator was dereferenced, but the overloaded operation did not return a reference. See `overload`.

Overloaded qr did not return a REGEXP

(F) An object with a `qr` overload was used as part of a match, but the overloaded operation didn't return a compiled regexp. See `overload`.

%s package attribute may clash with future reserved word: %s

(W reserved) A lowercase attribute name was used that had a package-specific handler. That name might have a meaning to Perl itself some day, even though it doesn't yet. Perhaps you should use a mixed-case attribute name, instead. See `attributes`.

pack/unpack repeat count overflow

(F) You can't specify a repeat count so large that it overflows your signed integers. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

page overflow

(W io) A single call to write() produced more lines than can fit on a page. See Section 24.1 [perlform NAME], page 343.

panic: %s

(P) An internal error.

panic: attempt to call %s in %s

(P) One of the file test operators entered a code branch that calls an ACL related-function, but that function is not available on this platform. Earlier checks mean that it should not be possible to enter this branch on this platform.

panic: child pseudo-process was never scheduled

(P) A child pseudo-process in the ithreads implementation on Windows was not scheduled within the time period allowed and therefore was not able to initialize properly.

panic: ck_grep, type=%u

(P) Failed an internal consistency check trying to compile a grep.

panic: ck_split, type=%u

(P) Failed an internal consistency check trying to compile a split.

panic: corrupt saved stack index %ld

(P) The savestack was requested to restore more localized values than there are in the savestack.

panic: del_backref

(P) Failed an internal consistency check while trying to reset a weak reference.

panic: die %s

(P) We popped the context stack to an eval context, and then discovered it wasn't an eval context.

panic: do_subst

(P) The internal pp_subst() routine was called with invalid operational data.

panic: do_trans_%s

(P) The internal do_trans routines were called with invalid operational data.

panic: fold_constants JMPENV_PUSH returned %d

(P) While attempting folding constants an exception other than an `eval` failure was caught.

panic: frexp: %f

(P) The library function frexp() failed, making printf("%f") impossible.

panic: goto, type=%u, ix=%ld

(P) We popped the context stack to a context with the specified label, and then discovered it wasn't a context we know how to do a goto in.

panic: gp_free failed to free glob pointer

(P) The internal routine used to clear a typeglob's entries tried repeatedly, but each time something re-created entries in the glob. Most likely the glob contains an object with a reference back to the glob and a destructor that adds a new object to the glob.

panic: INTERPCASEMOD, %s

(P) The lexer got into a bad state at a case modifier.

panic: INTERPCONCAT, %s

(P) The lexer got into a bad state parsing a string with brackets.

panic: kid popen errno read

(F) A forked child returned an incomprehensible message about its errno.

panic: last, type=%u

(P) We popped the context stack to a block context, and then discovered it wasn't a block context.

panic: leave_scope clearsv

(P) A writable lexical variable became read-only somehow within the scope.

panic: leave_scope inconsistency %u

(P) The savestack probably got out of sync. At least, there was an invalid enum on the top of it.

panic: magic_killbackrefs

(P) Failed an internal consistency check while trying to reset all weak references to an object.

panic: malloc, %s

(P) Something requested a negative number of bytes of malloc.

panic: memory wrap

(P) Something tried to allocate either more memory than possible or a negative amount.

panic: pad_alloc, %p!=%p

(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_free curpad, %p!=%p

(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_free po

(P) A zero scratch pad offset was detected internally. An attempt was made to free a target that had not been allocated to begin with.

panic: pad_reset curpad, %p!=%p

(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_sv po

(P) A zero scratch pad offset was detected internally. Most likely an operator needed a target but that target had not been allocated for whatever reason.

panic: pad_swipe curpad, %p!=%p

(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_swipe po

(P) An invalid scratch pad offset was detected internally.

panic: pp_iter, type=%u

(P) The foreach iterator got called in a non-loop context frame.

panic: pp_match%s

(P) The internal pp_match() routine was called with invalid operational data.

panic: pp_split, pm=%p, s=%p

(P) Something terrible went wrong in setting up for the split.

panic: realloc, %s

(P) Something requested a negative number of bytes of realloc.

panic: reference miscount on nsv in sv_replace() (%d != 1)

(P) The internal sv_replace() function was handed a new SV with a reference count other than 1.

panic: restartop in %s

(P) Some internal routine requested a goto (or something like it), and didn't supply the destination.

panic: return, type=%u

(P) We popped the context stack to a subroutine or eval context, and then discovered it wasn't a subroutine or eval context.

panic: scan_num, %s

(P) scan_num() got called on something that wasn't a number.

panic: Sequence (?{...}): no code block found in regex m/%s/

(P) While compiling a pattern that has embedded (?{}) or (??{}) code blocks, perl couldn't locate the code block that should have already been seen and compiled by perl before control passed to the regex compiler.

panic: strxfrm() gets absurd - a => %u, ab => %u

(P) The interpreter's sanity check of the C function strxfrm() failed. In your current locale the returned transformation of the string "ab" is shorter than that of the string "a", which makes no sense.

panic: sv_chop %s

(P) The sv_chop() routine was passed a position that is not within the scalar's string buffer.

panic: sv_insert, midend=%p, bigend=%p

(P) The sv_insert() routine was told to remove more string than there was string.

panic: top_env

(P) The compiler attempted to do a goto, or something weird like that.

panic: unimplemented op %s (#%d) called

(P) The compiler is screwed up and attempted to use an op that isn't permitted at run time.

panic: utf16_to_utf8: odd bytelen

(P) Something tried to call utf16_to_utf8 with an odd (as opposed to even) byte length.

panic: utf16_to_utf8_reversed: odd bytelen

(P) Something tried to call utf16_to_utf8_reversed with an odd (as opposed to even) byte length.

panic: yylex, %s

(P) The lexer got into a bad state while processing a case modifier.

Parentheses missing around "%s" list

(W parenthesis) You said something like

```
    my $foo, $bar = @_;
```

when you meant

```
    my ($foo, $bar) = @_;
```

Remember that "my", "our", "local" and "state" bind tighter than comma.

Parsing code internal error (%s)

(F) Parsing code supplied by an extension violated the parser's API in a detectable way.

Passing malformed UTF-8 to "%s" is deprecated

(D deprecated, utf8) This message indicates a bug either in the Perl core or in XS code. Such code was trying to find out if a character, allegedly stored internally encoded as UTF-8, was of a given type, such as being punctuation or a digit. But the character was not encoded in legal UTF-8. The `%s` is replaced by a string that can be used by knowledgeable people to determine what the type being checked against was. If `utf8` warnings are enabled, a further message is raised, giving details of the malformation.

Pattern subroutine nesting without pos change exceeded limit in regex

(F) You used a pattern that uses too many nested subpattern calls without consuming any text. Restructure the pattern so text is consumed before the nesting limit is exceeded.

`-p` destination: %s

(F) An error occurred during the implicit output invoked by the `-p` command-line switch. (This output goes to STDOUT unless you've redirected it with select().)

Perl API version %s of %s does not match %s

(F) The XS module in question was compiled against a different incompatible version of Perl than the one that has loaded the XS module.

Perl folding rules are not up-to-date for 0x%X; please use the perlbug utility to report; in regex; marked by <– HERE in m/%s/

(S regexp) You used a regular expression with case-insensitive matching, and there is a bug in Perl in which the built-in regular expression folding rules are not accurate. This may lead to incorrect results. Please report this as a bug using the `perlbug` utility.

PerlIO layer ':win32' is experimental

(S experimental::win32_perlio) The `:win32` PerlIO layer is experimental. If you want to take the risk of using this layer, simply disable this warning:

```
    no warnings "experimental::win32_perlio";
```

Perl_my_%s() not available

(F) Your platform has very uncommon byte-order and integer size, so it was not possible to set up some or all fixed-width byte-order conversion functions. This is only a problem when you're using the '<' or '>' modifiers in (un)pack templates. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

Perl %s required (did you mean %s?)–this is only %s, stopped

(F) The code you are trying to run has asked for a newer version of Perl than you are running. Perhaps `use 5.10` was written instead of `use 5.010` or `use v5.10`. Without the leading `v`, the number is interpreted as a decimal, with every three digits after the decimal point representing a part of the version number. So 5.10 is equivalent to v5.100.

Perl %s required–this is only %s, stopped

(F) The module in question uses features of a version of Perl more recent than the currently running version. How long has it been since you upgraded, anyway? See [perlfunc require], page 437.

PERL_SH_DIR too long

(F) An error peculiar to OS/2. PERL_SH_DIR is the directory to find the `sh`-shell in. See "PERL_SH_DIR" in `perlos2`.

PERL_SIGNALS illegal: "%s"

(X) See [perlrun PERL_SIGNALS], page 1196 for legal values.

Perls since %s too modern–this is %s, stopped

(F) The code you are trying to run claims it will not run on the version of Perl you are using because it is too new. Maybe the code needs to be updated, or maybe it is simply wrong and the version check should just be removed.

perl: warning: Non hex character in '$ENV{PERL_HASH_SEED}', seed only partially set

(S) PERL_HASH_SEED should match /^\s*(?:0x)?[0-9a-fA-F]+\s*\z/ but it contained a non hex character. This could mean you are not using the hash seed you think you are.

perl: warning: Setting locale failed.

(S) The whole warning message will look something like:

```
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
        LC_ALL = "En_US",
        LANG = (unset)
    are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

Exactly what were the failed locale settings varies. In the above the settings were that the LC_ALL was "En_US" and the LANG had no value. This error means that Perl detected that you and/or your operating system supplier and/or system administrator have set up the so-called locale system but Perl could not use those settings. This was not dead serious, fortunately: there is a "default locale" called "C" that Perl can and will use, and the script will be run. Before you really fix the problem, however, you will get the same error message each time you run Perl. How to really fix the problem can be found in Section 38.1 [perllocale NAME], page 701 section **LOCALE PROBLEMS**.

perl: warning: strange setting in '$ENV{PERL_PERTURB_KEYS}': '%s'

(S) Perl was run with the environment variable PERL_PERTURB_KEYS defined but containing an unexpected value. The legal values of this setting are as follows.

```
Numeric | String        | Result
--------+---------------+-----------------------------------------
0       | NO            | Disables key traversal randomization
1       | RANDOM        | Enables full key traversal randomization
2       | DETERMINISTIC | Enables repeatable key traversal
        |               | randomization
```

Both numeric and string values are accepted, but note that string values are case sensitive. The default for this setting is "RANDOM" or 1.

pid %x not a child

(W exec) A warning peculiar to VMS. Waitpid() was asked to wait for a process which isn't a subprocess of the current process. While this is fine from VMS' perspective, it's probably not what you intended.

'P' must have an explicit size in unpack

(F) The unpack format P must have an explicit size, not "*".

pop on reference is experimental

(S experimental::autoderef) `pop` with a scalar argument is experimental and may change or be removed in a future Perl version. If you want to take the risk of using this feature, simply disable this warning:

```
    no warnings "experimental::autoderef";
```

POSIX class [:%s:] unknown in regex; marked by <– HERE in m/%s/

(F) The class in the character class [: :] syntax is unknown. The <– HERE shows whereabouts in the regular expression the problem was discovered. Note that the POSIX character classes do **not** have the `is` prefix the corresponding C interfaces have: in other words, it's `[[:print:]]`, not `isprint`. See Section 58.1 [perlre NAME], page 989.

POSIX getpgrp can't take an argument

(F) Your system has POSIX getpgrp(), which takes no argument, unlike the BSD version, which takes a pid.

POSIX syntax [%c %c] belongs inside character classes in regex; marked by <– HERE in m/%s/

(W regexp) The character class constructs [: :], [= =], and [. .] go *inside* character classes, the [] are part of the construct, for example: /[012[:alpha:]345]/. Note that [= =] and [. .] are not currently implemented; they are simply placeholders for future extensions and will cause fatal errors. The <– HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

POSIX syntax [. .] is reserved for future extensions in regex; marked by <– HERE in m/%s/

(F) Within regular expression character classes ([]) the syntax beginning with "[." and ending with ".]" is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: "\[." and ".\]". The <– HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

POSIX syntax [= =] is reserved for future extensions in regex; marked by <– HERE in m/%s/

(F) Within regular expression character classes ([]) the syntax beginning with "[=" and ending with "=]" is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: "\[=" and "=\]". The <– HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Possible attempt to put comments in qw() list

(W qw) qw() lists contain items separated by whitespace; as with literal strings, comment characters are not ignored, but are instead treated as literal data.

(You may have used different delimiters than the parentheses shown here; braces are also frequently used.)

You probably wrote something like this:

```
@list = qw(
    a # a comment
    b # another comment
);
```

when you should have written this:

```
@list = qw(
    a
    b
);
```

If you really want comments, build your list the old-fashioned way, with quotes and commas:

```
@list = (
    ’a’,    # a comment
    ’b’,    # another comment
);
```

Possible attempt to separate words with commas

(W qw) qw() lists contain items separated by whitespace; therefore commas aren’t needed to separate the items. (You may have used different delimiters than the parentheses shown here; braces are also frequently used.)

You probably wrote something like this:

```
qw! a, b, c !;
```

which puts literal commas into some of the list items. Write it without commas if you don’t want them to appear in your data:

```
qw! a b c !;
```

Possible memory corruption: %s overflowed 3rd argument

(F) An ioctl() or fcntl() returned more than Perl was bargaining for. Perl guesses a reasonable buffer size, but puts a sentinel byte at the end of the buffer just in case. This sentinel byte got clobbered, and Perl assumes that memory is now corrupted. See ⟨undefined⟩ [perlfunc ioctl], page ⟨undefined⟩.

Possible precedence issue with control flow operator

(W syntax) There is a possible problem with the mixing of a control flow operator (e.g. `return`) and a low-precedence operator like `or`. Consider:

```
sub { return $a or $b; }
```

This is parsed as:

```
sub { (return $a) or $b; }
```

Which is effectively just:

```
sub { return $a; }
```

Either use parentheses or the high-precedence variant of the operator.

Note this may be also triggered for constructs like:

```
sub { 1 if die; }
```

Possible precedence problem on bitwise %s operator

(W precedence) Your program uses a bitwise logical operator in conjunction with a numeric comparison operator, like this :

```
if ($x & $y == 0) { ... }
```

This expression is actually equivalent to `$x & ($y == 0)`, due to the higher precedence of `==`. This is probably not what you want. (If you really meant to write this, disable the warning, or, better, put the parentheses explicitly and write `$x & ($y == 0)`).

Possible unintended interpolation of $\ in regex

(W ambiguous) You said something like `m/$\/` in a regex. The regex `m/foo$\s+bar/m` translates to: match the word 'foo', the output record separator (see [perlvar $\], page 1396) and the letter 's' (one time or more) followed by the word 'bar'.

If this is what you intended then you can silence the warning by using `m/${\}/` (for example: `m/foo${\}s+bar/`).

If instead you intended to match the word 'foo' at the end of the line followed by whitespace and the word 'bar' on the next line then you can use `m/$(?)\/` (for example: `m/foo$(?)\s+bar/`).

Possible unintended interpolation of %s in string

(W ambiguous) You said something like '@foo' in a double-quoted string but there was no array `@foo` in scope at the time. If you wanted a literal @foo, then write it as \@foo; otherwise find out what happened to the array you apparently lost track of.

Postfix dereference is experimental

(S experimental::postderef) This warning is emitted if you use the experimental postfix dereference syntax. Simply suppress the warning if you want to use the feature, but know that in doing so you are taking the risk of using an experimental feature which may change or be removed in a future Perl version:

```
no warnings "experimental::postderef";
use feature "postderef", "postderef_qq";
$ref->$*;
$aref->@*;
$aref->@[@indices];
... etc ...
```

Precedence problem: open %s should be open(%s)

(S precedence) The old irregular construct

```
open FOO || die;
```

is now misinterpreted as

```
open(FOO || die);
```

because of the strict regularization of Perl 5's grammar into unary and list operators. (The old open was a little of both.) You must put parentheses around the filehandle, or use the new `"or"` operator instead of `"||"`.

Premature end of script headers

See Server error.

printf() on closed filehandle %s

(W closed) The filehandle you're writing to got itself closed sometime before now. Check your control flow.

print() on closed filehandle %s

(W closed) The filehandle you're printing on got itself closed sometime before now. Check your control flow.

Process terminated by SIG%s

(W) This is a standard message issued by OS/2 applications, while *nix applications die in silence. It is considered a feature of the OS/2 port. One can easily disable this by appropriate sighandlers, see Section 36.3 [perlipc Signals], page 667. See also "Process terminated by SIGTERM/SIGINT" in `perlos2`.

Property '%s' is unknown in regex; marked by <– HERE in m/%s/

(F) The named property which you specified via `\p` or `\P` is not one known to Perl. Perhaps you misspelled the name? See Section "Properties accessible through \p{} and \P{}" in `perluniprops` for a complete list of available official properties. If it is a Section 81.2.6 [user-defined property], page 1329 it must have been defined by the time the regular expression is compiled.

Prototype after '%c' for %s : %s

(W illegalproto) A character follows % or @ in a prototype. This is useless, since % and @ gobble the rest of the subroutine arguments.

Prototype mismatch: %s vs %s

(S prototype) The subroutine being declared or defined had previously been declared or defined with a different function prototype.

Prototype not terminated

(F) You've omitted the closing parenthesis in a function prototype definition.

Prototype '%s' overridden by attribute 'prototype(%s)' in %s

(W prototype) A prototype was declared in both the parentheses after the sub name and via the prototype attribute. The prototype in parentheses is useless, since it will be replaced by the prototype from the attribute before it's ever used.

push on reference is experimental

(S experimental::autoderef) `push` with a scalar argument is experimental and may change or be removed in a future Perl version. If you want to take the risk of using this feature, simply disable this warning:

```
no warnings "experimental::autoderef";
```

Quantifier follows nothing in regex; marked by <– HERE in m/%s/

(F) You started a regular expression with a quantifier. Backslash it if you meant it literally. The <– HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Quantifier in {,} bigger than %d in regex; marked by <-- HERE in m/%s/

(F) There is currently a limit to the size of the min and max values of the {min,max} construct. The <-- HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Quantifier {n,m} with n > m can't match in regex
Quantifier {n,m} with n > m can't match in regex; marked by <-- HERE in m/%s/

(W regexp) Minima should be less than or equal to maxima. If you really want your regexp to match something 0 times, just put {0}.

Quantifier unexpected on zero-length expression in regex m/%s/

(W regexp) You applied a regular expression quantifier in a place where it makes no sense, such as on a zero-width assertion. Try putting the quantifier inside the assertion instead. For example, the way to match "abc" provided that it is followed by three repetitions of "xyz" is `/abc(?=(?:xyz){3})/`, not `/abc(?=xyz){3}/`.

Range iterator outside integer range

(F) One (or both) of the numeric arguments to the range operator ".." are outside the range which can be represented by integers internally. One possible workaround is to force Perl to use magical string increment by prepending "0" to your numbers.

Ranges of ASCII printables should be some subset of "0-9", "A-Z", or "a-z" in regex; marked by <-- HERE in m/%s/

(W regexp) (only under `use re 'strict'` or within `(?[...])`)

Stricter rules help to find typos and other errors. Perhaps you didn't even intend a range here, if the "-" was meant to be some other character, or should have been escaped (like "\-"). If you did intend a range, the one that was used is not portable between ASCII and EBCDIC platforms, and doesn't have an obvious meaning to a casual reader.

```
 [3-7]    # OK; Obvious and portable
 [d-g]    # OK; Obvious and portable
 [A-Y]    # OK; Obvious and portable
 [A-z]    # WRONG; Not portable; not clear what is meant
 [a-Z]    # WRONG; Not portable; not clear what is meant
 [%-.]    # WRONG; Not portable; not clear what is meant
 [\x41-Z] # WRONG; Not portable; not obvious to non-geek
```

(You can force portability by specifying a Unicode range, which means that the endpoints are specified by Section 61.2.3.2 [\N{...}], page 1067, but the meaning may still not be obvious.) The stricter rules require that ranges that start or stop with an ASCII character that is not a control have all their endpoints be the literal character, and not some escape sequence (like "\x41"), and the ranges must be all digits, or all uppercase letters, or all lowercase letters.

Ranges of digits should be from the same group in regex; marked by <-- HERE in m/%s/

(W regexp) (only under `use re 'strict'` or within `(?[...])`)

Stricter rules help to find typos and other errors. You included a range, and at least one of the end points is a decimal digit. Under the stricter rules, when this happens, both end points should be digits in the same group of 10 consecutive digits.

readdir() attempted on invalid dirhandle %s

(W io) The dirhandle you're reading from is either closed or not really a dirhandle. Check your control flow.

readline() on closed filehandle %s

(W closed) The filehandle you're reading from got itself closed sometime before now. Check your control flow.

read() on closed filehandle %s

(W closed) You tried to read from a closed filehandle.

read() on unopened filehandle %s

(W unopened) You tried to read from a filehandle that was never opened.

Reallocation too large: %x

(F) You can't allocate more than 64K on an MS-DOS machine.

realloc() of freed memory ignored

(S malloc) An internal routine called realloc() on something that had already been freed.

Recompile perl with **-D**DEBUGGING to use **-D** switch

(S debugging) You can't use the **-D** option unless the code to produce the desired output is compiled into Perl, which entails some overhead, which is why it's currently left out of your copy.

Recursive call to Perl_load_module in PerlIO_find_layer

(P) It is currently not permitted to load modules when creating a filehandle inside an %INC hook. This can happen with `open my $fh, '<', \$scalar`, which implicitly loads PerlIO::scalar. Try loading PerlIO::scalar explicitly first.

Recursive inheritance detected in package '%s'

(F) While calculating the method resolution order (MRO) of a package, Perl believes it found an infinite loop in the `@ISA` hierarchy. This is a crude check that bails out after 100 levels of `@ISA` depth.

Redundant argument in %s

(W redundant) You called a function with more arguments than other arguments you supplied indicated would be needed. Currently only emitted when a printf-type format required fewer arguments than were supplied, but might be used in the future for e.g. ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

refcnt_dec: fd %d%s
refcnt: fd %d%s
refcnt_inc: fd %d%s

(P) Perl's I/O implementation failed an internal consistency check. If you see this message, something is very wrong.

Reference found where even-sized list expected

(W misc) You gave a single reference where Perl was expecting a list with an even number of elements (for assignment to a hash). This usually means that you used the anon hash constructor when you meant to use parens. In any case, a hash requires key/value **pairs**.

```
%hash = { one => 1, two => 2, };     # WRONG
%hash = [ qw/ an anon array / ];     # WRONG
%hash = ( one => 1, two => 2, );     # right
%hash = qw( one 1 two 2 );                   # also fine
```

Reference is already weak

(W misc) You have attempted to weaken a reference that is already weak. Doing so has no effect.

Reference to invalid group 0 in regex; marked by <– HERE in m/%s/

(F) You used `\g0` or similar in a regular expression. You may refer to capturing parentheses only with strictly positive integers (normal backreferences) or with strictly negative integers (relative backreferences). Using 0 does not make sense.

Reference to nonexistent group in regex; marked by <– HERE in m/%s/

(F) You used something like `\7` in your regular expression, but there are not at least seven sets of capturing parentheses in the expression. If you wanted to have the character with ordinal 7 inserted into the regular expression, prepend zeroes to make it three digits long: `\007`

The <– HERE shows whereabouts in the regular expression the problem was discovered.

Reference to nonexistent named group in regex; marked by <– HERE in m/%s/

(F) You used something like `\k'NAME'` or `\k<NAME>` in your regular expression, but there is no corresponding named capturing parentheses such as `(?'NAME'...)` or `(?<NAME>...)`. Check if the name has been spelled correctly both in the backreference and the declaration.

The <– HERE shows whereabouts in the regular expression the problem was discovered.

Reference to nonexistent or unclosed group in regex; marked by <– HERE in m/%s/

(F) You used something like `\g{-7}` in your regular expression, but there are not at least seven sets of closed capturing parentheses in the expression before where the `\g{-7}` was located.

The <– HERE shows whereabouts in the regular expression the problem was discovered.

regexp memory corruption

(P) The regular expression engine got confused by what the regular expression compiler gave it.

Regexp modifier "/%c" may appear a maximum of twice
Regexp modifier "%c" may appear a maximum of twice in regex; marked by <– HERE in m/%s/

(F) The regular expression pattern had too many occurrences of the specified modifier. Remove the extraneous ones.

Regexp modifier "%c" may not appear after the "-" in regex; marked by <– HERE in m/%s/

(F) Turning off the given modifier has the side effect of turning on another one. Perl currently doesn't allow this. Reword the regular expression to use the modifier you want to turn on (and place it before the minus), instead of the one you want to turn off.

Regexp modifier "/%c" may not appear twice
Regexp modifier "%c" may not appear twice in regex; marked by <– HERE in m/%s/

(F) The regular expression pattern had too many occurrences of the specified modifier. Remove the extraneous ones.

Regexp modifiers "/%c" and "/%c" are mutually exclusive
Regexp modifiers "%c" and "%c" are mutually exclusive in regex; marked by <– HERE in m/%s/

(F) The regular expression pattern had more than one of these mutually exclusive modifiers. Retain only the modifier that is supposed to be there.

Regexp out of space in regex m/%s/

(P) A "can't happen" error, because safemalloc() should have caught it earlier.

Repeated format line will never terminate (~~ and @#)

(F) Your format contains the ~~ repeat-until-blank sequence and a numeric field that will never go blank so that the repetition never terminates. You might use ^# instead. See Section 24.1 [perlform NAME], page 343.

Replacement list is longer than search list

(W misc) You have used a replacement list that is longer than the search list. So the additional elements in the replacement list are meaningless.

'%s' resolved to '\o{%s}%d'

(W misc, regexp) You wrote something like \08, or \179 in a double-quotish string. All but the last digit is treated as a single character, specified in octal. The last digit is the next character in the string. To tell Perl that this is indeed what you want, you can use the \o{ } syntax, or use exactly three digits to specify the octal for the character.

Reversed %s= operator

(W syntax) You wrote your assignment operator backwards. The = must always come last, to avoid ambiguity with subsequent unary operators.

rewinddir() attempted on invalid dirhandle %s

(W io) The dirhandle you tried to do a rewinddir() on is either closed or not really a dirhandle. Check your control flow.

Scalars leaked: %d

(S internal) Something went wrong in Perl's internal bookkeeping of scalars: not all scalar variables were deallocated by the time Perl exited. What this usually indicates is a memory leak, which is of course bad, especially if the Perl program is intended to be long-running.

Scalar value @%s[%s] better written as $%s[%s]

(W syntax) You've used an array slice (indicated by @) to select a single element of an array. Generally it's better to ask for a scalar value (indicated by $). The difference is that `$foo[&bar]` always behaves like a scalar, both when assigning to it and when evaluating its argument, while `@foo[&bar]` behaves like a list when you assign to it, and provides a list context to its subscript, which can do weird things if you're expecting only one subscript.

On the other hand, if you were actually hoping to treat the array element as a list, you need to look into how references work, because Perl will not magically convert between scalars and lists for you. See Section 62.1 [perlref NAME], page 1077.

Scalar value @%s{%s} better written as $%s{%s}

(W syntax) You've used a hash slice (indicated by @) to select a single element of a hash. Generally it's better to ask for a scalar value (indicated by $). The difference is that `$foo{&bar}` always behaves like a scalar, both when assigning to it and when evaluating its argument, while `@foo{&bar}` behaves like a list when you assign to it, and provides a list context to its subscript, which can do weird things if you're expecting only one subscript.

On the other hand, if you were actually hoping to treat the hash element as a list, you need to look into how references work, because Perl will not magically convert between scalars and lists for you. See Section 62.1 [perlref NAME], page 1077.

Search pattern not terminated

(F) The lexer couldn't find the final delimiter of a // or m{} construct. Remember that bracketing delimiters count nesting level. Missing the leading `$` from a variable `$m` may cause this error.

Note that since Perl 5.10.0 a // can also be the *defined-or* construct, not just the empty search pattern. Therefore code written in Perl 5.10.0 or later that uses the // as the *defined-or* can be misparsed by pre-5.10.0 Perls as a non-terminated search pattern.

seekdir() attempted on invalid dirhandle %s

(W io) The dirhandle you are doing a seekdir() on is either closed or not really a dirhandle. Check your control flow.

%sseek() on unopened filehandle

(W unopened) You tried to use the seek() or sysseek() function on a filehandle that was either never opened or has since been closed.

select not implemented

(F) This machine doesn't implement the select() system call.

Self-ties of arrays and hashes are not supported

(F) Self-ties are of arrays and hashes are not supported in the current implementation.

Semicolon seems to be missing

(W semicolon) A nearby syntax error was probably caused by a missing semicolon, or possibly some other missing operator, such as a comma.

semi-panic: attempt to dup freed string

(S internal) The internal newSVsv() routine was called to duplicate a scalar that had previously been marked as free.

sem%s not implemented

(F) You don't have System V semaphore IPC on your system.

send() on closed socket %s

(W closed) The socket you're sending to got itself closed sometime before now. Check your control flow.

Sequence "\c{" invalid

(F) These three characters may not appear in sequence in a double-quotish context. This message is raised only on non-ASCII platforms (a different error message is output on ASCII ones). If you were intending to specify a control character with this sequence, you'll have to use a different way to specify it.

Sequence (? incomplete in regex; marked by <– HERE in m/%s/

(F) A regular expression ended with an incomplete extension (?. The <– HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Sequence (?%c...) not implemented in regex; marked by <– HERE in m/%s/

(F) A proposed regular expression extension has the character reserved but has not yet been written. The <– HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Sequence (?%s...) not recognized in regex; marked by <– HERE in m/%s/

(F) You used a regular expression extension that doesn't make sense. The <– HERE shows whereabouts in the regular expression the problem was discovered. This may happen when using the `(?^...)` construct to tell Perl to use the default regular expression modifiers, and you redundantly specify a default modifier. For other causes, see Section 58.1 [perlre NAME], page 989.

Sequence (?#... not terminated in regex m/%s/

(F) A regular expression comment must be terminated by a closing parenthesis. Embedded parentheses aren't allowed. See Section 58.1 [perlre NAME], page 989.

Sequence (?&... not terminated in regex; marked by <– HERE in m/%s/

(F) A named reference of the form `(?&...)` was missing the final closing parenthesis after the name. The <– HERE shows whereabouts in the regular expression the problem was discovered.

Sequence (?%c... not terminated in regex; marked by <– HERE in m/%s/

(F) A named group of the form `(?'...')` or `(?<...>)` was missing the final closing quote or angle bracket. The <– HERE shows whereabouts in the regular expression the problem was discovered.

Sequence (?(%c... not terminated in regex; marked by <– HERE in m/%s/

(F) A named reference of the form `(?('...')...)` or `(?(<...>)...)` was missing the final closing quote or angle bracket after the name. The <– HERE shows whereabouts in the regular expression the problem was discovered.

Sequence \%s... not terminated in regex; marked by <– HERE in m/%s/

(F) The regular expression expects a mandatory argument following the escape sequence and this has been omitted or incorrectly written.

Sequence (?{...}) not terminated with ')'

(F) The end of the perl code contained within the {...} must be followed immediately by a ')'.

Sequence ?P=... not terminated in regex; marked by <– HERE in m/%s/

(F) A named reference of the form `(?P=...)` was missing the final closing parenthesis after the name. The <– HERE shows whereabouts in the regular expression the problem was discovered.

Sequence (?R) not terminated in regex m/%s/

(F) An `(?R)` or `(?0)` sequence in a regular expression was missing the final parenthesis.

Server error (a.k.a. "500 Server error")

(A) This is the error message generally seen in a browser window when trying to run a CGI program (including SSI) over the web. The actual error text varies widely from server to server. The most frequently-seen variants are "500 Server error", "Method (something) not permitted", "Document contains no data", "Premature end of script headers", and "Did not produce a valid header".

This is a CGI error, not a Perl error.

You need to make sure your script is executable, is accessible by the user CGI is running the script under (which is probably not the user account you tested it under), does not rely on any environment variables (like PATH) from the user it isn't running under, and isn't in a location where the CGI server can't find it, basically, more or less. Please see the following for more information:

```
http://www.perl.org/CGI_MetaFAQ.html
http://www.htmlhelp.org/faq/cgifaq.html
http://www.w3.org/Security/Faq/
```

You should also look at `perlfaq9`.

setegid() not implemented

(F) You tried to assign to `$)`, and your operating system doesn't support the setegid() system call (or equivalent), or at least Configure didn't think so.

seteuid() not implemented

(F) You tried to assign to `$>`, and your operating system doesn't support the seteuid() system call (or equivalent), or at least Configure didn't think so.

setpgrp can't take arguments

(F) Your system has the setpgrp() from BSD 4.2, which takes no arguments, unlike POSIX setpgid(), which takes a process ID and process group ID.

setrgid() not implemented

(F) You tried to assign to `$(`, and your operating system doesn't support the setrgid() system call (or equivalent), or at least Configure didn't think so.

setruid() not implemented

(F) You tried to assign to `$<`, and your operating system doesn't support the setruid() system call (or equivalent), or at least Configure didn't think so.

setsockopt() on closed socket %s

(W closed) You tried to set a socket option on a closed socket. Did you forget to check the return value of your socket() call? See ⟨undefined⟩ [perlfunc setsockopt], page ⟨undefined⟩.

Setting ${^ENCODING} is deprecated

(D deprecated) You assigned a non-`undef` value to `${^ENCODING}`. This is deprecated; see `[perlvar ${^ENCODING}], page 1402` for details.

Setting $/ to a reference to %s as a form of slurp is deprecated, treating as undef

(D deprecated) You assigned a reference to a scalar to `$/` where the referenced item is not a positive integer. In older perls this **appeared** to work the same as setting it to `undef` but was in fact internally different, less efficient and with very bad luck could have resulted in your file being split by a stringified form of the reference.

In Perl 5.20.0 this was changed so that it would be **exactly** the same as setting `$/` to undef, with the exception that this warning would be thrown.

You are recommended to change your code to set `$/` to `undef` explicitly if you wish to slurp the file. In future versions of Perl assigning a reference to will throw a fatal error.

Setting $/ to %s reference is forbidden

(F) You tried to assign a reference to a non integer to `$/`. In older Perls this would have behaved similarly to setting it to a reference to a positive integer, where the integer was the address of the reference. As of Perl 5.20.0 this is a fatal error, to allow future versions of Perl to use non-integer refs for more interesting purposes.

shift on reference is experimental

(S experimental::autoderef) `shift` with a scalar argument is experimental and may change or be removed in a future Perl version. If you want to take the risk of using this feature, simply disable this warning:

```
    no warnings "experimental::autoderef";
```

shm%s not implemented

(F) You don't have System V shared memory IPC on your system.

!=~ should be !~

(W syntax) The non-matching operator is !~, not !=~. !=~ will be interpreted as the != (numeric not equal) and ~ (1's complement) operators: probably not what you intended.

/%s/ should probably be written as "%s"

(W syntax) You have used a pattern where Perl expected to find a string, as in the first argument to `join`. Perl will treat the true or false result of matching the pattern against $_ as the string, which is probably not what you had in mind.

shutdown() on closed socket %s

(W closed) You tried to do a shutdown on a closed socket. Seems a bit superfluous.

SIG%s handler "%s" not defined

(W signal) The signal handler named in %SIG doesn't, in fact, exist. Perhaps you put it into the wrong package?

Slab leaked from cv %p

(S) If you see this message, then something is seriously wrong with the internal bookkeeping of op trees. An op tree needed to be freed after a compilation error, but could not be found, so it was leaked instead.

sleep(%u) too large

(W overflow) You called `sleep` with a number that was larger than it can reliably handle and `sleep` probably slept for less time than requested.

Slurpy parameter not last

(F) In a subroutine signature, you put something after a slurpy (array or hash) parameter. The slurpy parameter takes all the available arguments, so there can't be any left to fill later parameters.

Smart matching a non-overloaded object breaks encapsulation

(F) You should not use the `~~` operator on an object that does not overload it: Perl refuses to use the object's underlying structure for the smart match.

Smartmatch is experimental

(S experimental::smartmatch) This warning is emitted if you use the smartmatch (`~~`) operator. This is currently an experimental feature, and its details are subject to change in future releases of Perl. Particularly, its current behavior is noticed for being unnecessarily complex and unintuitive, and is very likely to be overhauled.

sort is now a reserved word

(F) An ancient error message that almost nobody ever runs into anymore. But before sort was a keyword, people sometimes used it as a filehandle.

Sort subroutine didn't return single value

(F) A sort comparison subroutine written in XS must return exactly one item. See ⟨undefined⟩ [perlfunc sort], page ⟨undefined⟩.

Source filters apply only to byte streams

(F) You tried to activate a source filter (usually by loading a source filter module) within a string passed to `eval`. This is not permitted under the `unicode_eval` feature. Consider using `evalbytes` instead. See `feature`.

splice() offset past end of array

(W misc) You attempted to specify an offset that was past the end of the array passed to splice(). Splicing will instead commence at the end of the array, rather than past it. If this isn't what you want, try explicitly pre-extending the array by assigning $#array = $offset. See ⟨undefined⟩ [perlfunc splice], page ⟨undefined⟩.

splice on reference is experimental

(S experimental::autoderef) `splice` with a scalar argument is experimental and may change or be removed in a future Perl version. If you want to take the risk of using this feature, simply disable this warning:

```
    no warnings "experimental::autoderef";
```

Split loop

(P) The split was looping infinitely. (Obviously, a split shouldn't iterate more times than there are characters of input, which is what happened.) See [perlfunc split], page 453.

Statement unlikely to be reached

(W exec) You did an exec() with some statement after it other than a die(). This is almost always an error, because exec() never returns unless there was a failure. You probably wanted to use system() instead, which does return. To suppress this warning, put the exec() in a block by itself.

"state" subroutine %s can't be in a package

(F) Lexically scoped subroutines aren't in a package, so it doesn't make sense to try to declare one with a package qualifier on the front.

"state %s" used in sort comparison

(W syntax) The package variables $a and $b are used for sort comparisons. You used $a or $b in as an operand to the `<=>` or `cmp` operator inside a sort comparison block, and the variable had earlier been declared as a lexical variable. Either qualify the sort variable with the package name, or rename the lexical variable.

"state" variable %s can't be in a package

(F) Lexically scoped variables aren't in a package, so it doesn't make sense to try to declare one with a package qualifier on the front. Use local() if you want to localize a package variable.

stat() on unopened filehandle %s

(W unopened) You tried to use the stat() function on a filehandle that was either never opened or has since been closed.

Strings with code points over 0xFF may not be mapped into in-memory file handles

(W utf8) You tried to open a reference to a scalar for read or append where the scalar contained code points over 0xFF. In-memory files model on-disk files and can only contain bytes.

Stub found while resolving method "%s" overloading "%s" in package "%s"

(P) Overloading resolution over @ISA tree may be broken by importation stubs. Stubs should never be implicitly created, but explicit calls to `can` may break this.

Subroutine "&%s" is not available

(W closure) During compilation, an inner named subroutine or eval is attempting to capture an outer lexical subroutine that is not currently available. This can happen for one of two reasons. First, the lexical subroutine may be declared in an outer anonymous subroutine that has not yet been created. (Remember that named subs are created at compile time, while anonymous subs are created at run-time.) For example,

```
sub { my sub a {...} sub f { \&a } }
```

At the time that f is created, it can't capture the current "a" sub, since the anonymous subroutine hasn't been created yet. Conversely, the following won't give a warning since the anonymous subroutine has by now been created and is live:

```
sub { my sub a {...} eval 'sub f { \&a }' }->();
```

The second situation is caused by an eval accessing a lexical subroutine that has gone out of scope, for example,

```
sub f {
    my sub a {...}
    sub { eval '\&a' }
}
f()->();
```

Here, when the '\&a' in the eval is being compiled, f() is not currently being executed, so its &a is not available for capture.

"%s" subroutine &%s masks earlier declaration in same %s

(W misc) A "my" or "state" subroutine has been redeclared in the current scope or statement, effectively eliminating all access to the previous instance. This is almost always a typographical error. Note that the earlier subroutine will still exist until the end of the scope or until all closure references to it are destroyed.

Subroutine %s redefined

(W redefine) You redefined a subroutine. To suppress this warning, say

```
{
    no warnings 'redefine';
    eval "sub name { ... }";
}
```

Subroutine "%s" will not stay shared

(W closure) An inner (nested) *named* subroutine is referencing a "my" subroutine defined in an outer named subroutine.

When the inner subroutine is called, it will see the value of the outer subroutine's lexical subroutine as it was before and during the *first* call to the outer subroutine; in this case, after the first call to the outer subroutine is complete, the inner and outer subroutines will no longer share a common value for the lexical subroutine. In other words, it will no longer be shared. This will especially make a difference if the lexical subroutines accesses lexical variables declared in its surrounding scope.

This problem can usually be solved by making the inner subroutine anonymous, using the `sub {}` syntax. When inner anonymous subs that reference lexical subroutines in outer subroutines are created, they are automatically rebound to the current values of such lexical subs.

Substitution loop

(P) The substitution was looping infinitely. (Obviously, a substitution shouldn't iterate more times than there are characters of input, which is what happened.) See the discussion of substitution in Section 48.2.30 [perlop Regexp Quote-Like Operators], page 823.

Substitution pattern not terminated

(F) The lexer couldn't find the interior delimiter of an s/// or s{}{} construct. Remember that bracketing delimiters count nesting level. Missing the leading `$` from variable `$s` may cause this error.

Substitution replacement not terminated

(F) The lexer couldn't find the final delimiter of an s/// or s{}{} construct. Remember that bracketing delimiters count nesting level. Missing the leading `$` from variable `$s` may cause this error.

substr outside of string

(W substr)(F) You tried to reference a substr() that pointed outside of a string. That is, the absolute value of the offset was larger than the length of the string. See ⟨undefined⟩ [perlfunc substr], page ⟨undefined⟩. This warning is fatal if substr is used in an lvalue context (as the left hand side of an assignment or as a subroutine argument for example).

sv_upgrade from type %d down to type %d

(P) Perl tried to force the upgrade of an SV to a type which was actually inferior to its current type.

SWASHNEW didn't return an HV ref

(P) Something went wrong internally when Perl was trying to look up Unicode characters.

Switch (?(condition)... contains too many branches in regex; marked by <– HERE in m/%s/

(F) A (?(condition)if-clause|else-clause) construct can have at most two branches (the if-clause and the else-clause). If you want one or both to contain alternation, such as using `this|that|other`, enclose it in clustering parentheses:

```
(?(condition)(?:this|that|other)|else-clause)
```

The <– HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Switch condition not recognized in regex; marked by <– HERE in m/%s/

(F) The condition part of a (?(condition)if-clause|else-clause) construct is not known. The condition must be one of the following:

```
(1) (2) ...         true if 1st, 2nd, etc., capture matched
(<NAME>) ('NAME')   true if named capture matched
(?=...) (?<=...)    true if subpattern matches
(?!...) (?<!...)    true if subpattern fails to match
(?{ CODE })         true if code returns a true value
(R)                 true if evaluating inside recursion
```

```
 (R1) (R2) ...       true if directly inside capture group 1, 2, etc.
 (R&NAME)            true if directly inside named capture
 (DEFINE)            always false; for defining named subpatterns
```

The <– HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Switch (?(condition)... not terminated in regex; marked by <– HERE in m/%s/

(F) You omitted to close a (?(condition)...) block somewhere in the pattern. Add a closing parenthesis in the appropriate position. See Section 58.1 [perlre NAME], page 989.

switching effective %s is not implemented

(F) While under the `use filetest` pragma, we cannot switch the real and effective uids or gids.

syntax error

(F) Probably means you had a syntax error. Common reasons include:

```
A keyword is misspelled.
A semicolon is missing.
A comma is missing.
An opening or closing parenthesis is missing.
An opening or closing brace is missing.
A closing quote is missing.
```

Often there will be another error message associated with the syntax error giving more information. (Sometimes it helps to turn on **-w**.) The error message itself often tells you where it was in the line when it decided to give up. Sometimes the actual error is several tokens before this, because Perl is good at understanding random input. Occasionally the line number may be misleading, and once in a blue moon the only way to figure out what's triggering the error is to call `perl -c` repeatedly, chopping away half the program each time to see if the error went away. Sort of the cybernetic version of 20 questions.

syntax error at line %d: '%s' unexpected

(A) You've accidentally run your script through the Bourne shell instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

syntax error in file %s at line %d, next 2 tokens "%s"

(F) This error is likely to occur if you run a perl5 script through a perl4 interpreter, especially if the next 2 tokens are "use strict" or "my $var" or "our $var".

Syntax error in (?[...]) in regex m/%s/

(F) Perl could not figure out what you meant inside this construct; this notifies you that it is giving up trying.

%s syntax OK

(F) The final summary message when a `perl -c` succeeds.

sysread() on closed filehandle %s

(W closed) You tried to read from a closed filehandle.

sysread() on unopened filehandle %s

(W unopened) You tried to read from a filehandle that was never opened.

System V %s is not implemented on this machine

(F) You tried to do something with a function beginning with "sem", "shm", or "msg" but that System V IPC is not implemented in your machine. In some machines the functionality can exist but be unconfigured. Consult your system support.

syswrite() on closed filehandle %s

(W closed) The filehandle you're writing to got itself closed sometime before now. Check your control flow.

`-T` and `-B` not implemented on filehandles

(F) Perl can't peek at the stdio buffer of filehandles when it doesn't know about your kind of stdio. You'll have to use a filename instead.

Target of goto is too deeply nested

(F) You tried to use `goto` to reach a label that was too deeply nested for Perl to reach. Perl is doing you a favor by refusing.

telldir() attempted on invalid dirhandle %s

(W io) The dirhandle you tried to telldir() is either closed or not really a dirhandle. Check your control flow.

tell() on unopened filehandle

(W unopened) You tried to use the tell() function on a filehandle that was either never opened or has since been closed.

That use of $[is unsupported

(F) Assignment to `$[` is now strictly circumscribed, and interpreted as a compiler directive. You may say only one of

```
$[ = 0;
$[ = 1;
...
local $[ = 0;
local $[ = 1;
...
```

This is to prevent the problem of one module changing the array base out from under another module inadvertently. See [perlvar $[], page 1408 and `arybase`.

The bitwise feature is experimental

(S experimental::bitwise) This warning is emitted if you use bitwise operators (`& | ^ ~ &. |. ^. ~.`) with the "bitwise" feature enabled. Simply suppress the warning if you want to use the feature, but know that in doing so you are taking the risk of using an experimental feature which may change or be removed in a future Perl version:

```
no warnings "experimental::bitwise";
use feature "bitwise";
$x |.= $y;
```

The crypt() function is unimplemented due to excessive paranoia.

(F) Configure couldn't find the crypt() function on your machine, probably because your vendor didn't supply it, probably because they think the U.S. Government thinks it's a secret, or at least that they will continue to pretend that it is. And if you quote me on that, I will deny it.

The %s function is unimplemented

(F) The function indicated isn't implemented on this architecture, according to the probings of Configure.

The lexical_subs feature is experimental

(S experimental::lexical_subs) This warning is emitted if you declare a sub with `my` or `state`. Simply suppress the warning if you want to use the feature, but know that in doing so you are taking the risk of using an experimental feature which may change or be removed in a future Perl version:

```
    no warnings "experimental::lexical_subs";
    use feature "lexical_subs";
    my sub foo { ... }
```

The regex_sets feature is experimental

(S experimental::regex_sets) This warning is emitted if you use the syntax `(?[ ])` in a regular expression. The details of this feature are subject to change. if you want to use it, but know that in doing so you are taking the risk of using an experimental feature which may change in a future Perl version, you can do this to silence the warning:

```
    no warnings "experimental::regex_sets";
```

The signatures feature is experimental

(S experimental::signatures) This warning is emitted if you unwrap a subroutine's arguments using a signature. Simply suppress the warning if you want to use the feature, but know that in doing so you are taking the risk of using an experimental feature which may change or be removed in a future Perl version:

```
    no warnings "experimental::signatures";
    use feature "signatures";
    sub foo ($left, $right) { ... }
```

The stat preceding %s wasn't an lstat

(F) It makes no sense to test the current stat buffer for symbolic linkhood if the last stat that wrote to the stat buffer already went past the symlink to get to the real file. Use an actual filename instead.

The 'unique' attribute may only be applied to 'our' variables

(F) This attribute was never supported on `my` or `sub` declarations.

This Perl can't reset CRTL environ elements (%s)
This Perl can't set CRTL environ elements (%s=%s)

(W internal) Warnings peculiar to VMS. You tried to change or delete an element of the CRTL's internal environ array, but your copy of Perl wasn't built with a CRTL that contained the setenv() function. You'll need to rebuild Perl with a CRTL that does, or redefine `PERL_ENV_TABLES` (see Section 87.1 [perlvms NAME], page 1409) so that the environ array isn't the target of the change to %ENV which produced the warning.

This Perl has not been built with support for randomized hash key traversal but something called Perl_hv_rand_set().

(F) Something has attempted to use an internal API call which depends on Perl being compiled with the default support for randomized hash key traversal, but this Perl has been compiled without it. You should report this warning to the relevant upstream party, or recompile perl with default options.

times not implemented

(F) Your version of the C library apparently doesn't do times(). I suspect you're not running on Unix.

"-T" is on the #! line, it must also be used on the command line

(X) The #! line (or local equivalent) in a Perl script contains the **-T** option (or the **-t** option), but Perl was not invoked with **-T** in its command line. This is an error because, by the time Perl discovers a **-T** in a script, it's too late to properly taint everything from the environment. So Perl gives up.

If the Perl script is being executed as a command using the #! mechanism (or its local equivalent), this error can usually be fixed by editing the #! line so that the **-%c** option is a part of Perl's first argument: e.g. change `perl -n -%c` to `perl -%c -n`.

If the Perl script is being executed as `perl scriptname`, then the **-%c** option must appear on the command line: `perl -%c scriptname`.

To%s: illegal mapping '%s'

(F) You tried to define a customized To-mapping for lc(), lcfirst, uc(), or ucfirst() (or their string-inlined versions), but you specified an illegal mapping. See Section 81.2.6 [perlunicode User-Defined Character Properties], page 1329.

Too deeply nested ()-groups

(F) Your template contains ()-groups with a ridiculously deep nesting level.

Too few args to syscall

(F) There has to be at least one argument to syscall() to specify the system call to call, silly dilly.

Too few arguments for subroutine

(F) A subroutine using a signature received fewer arguments than required by the signature. The caller of the subroutine is presumably at fault. Inconveniently, this error will be reported at the location of the subroutine, not that of the caller.

Too late for "-%s" option

(X) The #! line (or local equivalent) in a Perl script contains the **-M**, **-m** or **-C** option.

In the case of **-M** and **-m**, this is an error because those options are not intended for use inside scripts. Use the `use` pragma instead.

The **-C** option only works if it is specified on the command line as well (with the same sequence of letters or numbers following). Either specify this option on the command line, or, if your system supports it, make your script executable and run it directly instead of passing it to perl.

Too late to run %s block

(W void) A CHECK or INIT block is being defined during run time proper, when the opportunity to run them has already passed. Perhaps you are loading a file with `require` or `do` when you should be using `use` instead. Or perhaps you should put the `require` or `do` inside a BEGIN block.

Too many args to syscall

(F) Perl supports a maximum of only 14 args to syscall().

Too many arguments for %s

(F) The function requires fewer arguments than you specified.

Too many arguments for subroutine

(F) A subroutine using a signature received more arguments than required by the signature. The caller of the subroutine is presumably at fault. Inconveniently, this error will be reported at the location of the subroutine, not that of the caller.

Too many)'s

(A) You've accidentally run your script through **csh** instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

Too many ('s

(A) You've accidentally run your script through **csh** instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

Trailing \ in regex m/%s/

(F) The regular expression ends with an unbackslashed backslash. Backslash it. See Section 58.1 [perlre NAME], page 989.

Transliteration pattern not terminated

(F) The lexer couldn't find the interior delimiter of a tr/// or tr[][] or y/// or y[][] construct. Missing the leading `$` from variables `$tr` or `$y` may cause this error.

Transliteration replacement not terminated

(F) The lexer couldn't find the final delimiter of a tr///, tr[][], y/// or y[][] construct.

'%s' trapped by operation mask

(F) You tried to use an operator from a Safe compartment in which it's disallowed. See `Safe`.

truncate not implemented

(F) Your machine doesn't implement a file truncation mechanism that Configure knows about.

Type of arg %d to &CORE::%s must be %s

(F) The subroutine in question in the CORE package requires its argument to be a hard reference to data of the specified type. Overloading is ignored, so a reference to an object that is not the specified type, but nonetheless has overloading to handle it, will still not be accepted.

Type of arg %d to %s must be %s (not %s)

(F) This function requires the argument in that position to be of a certain type. Arrays must be @NAME or `@{EXPR}`. Hashes must be %NAME or `%{EXPR}`. No implicit dereferencing is allowed–use the {EXPR} forms as an explicit dereference. See Section 62.1 [perlref NAME], page 1077.

Type of argument to %s must be unblessed hashref or arrayref

(F) You called `keys`, `values` or `each` with a scalar argument that was not a reference to an unblessed hash or array.

umask not implemented

(F) Your machine doesn't implement the umask function and you tried to use it to restrict permissions for yourself (EXPR & 0700).

Unbalanced context: %d more PUSHes than POPs

(S internal) The exit code detected an internal inconsistency in how many execution contexts were entered and left.

Unbalanced saves: %d more saves than restores

(S internal) The exit code detected an internal inconsistency in how many values were temporarily localized.

Unbalanced scopes: %d more ENTERs than LEAVEs

(S internal) The exit code detected an internal inconsistency in how many blocks were entered and left.

Unbalanced string table refcount: (%d) for "%s"

(S internal) On exit, Perl found some strings remaining in the shared string table used for copy on write and for hash keys. The entries should have been freed, so this indicates a bug somewhere.

Unbalanced tmps: %d more allocs than frees

(S internal) The exit code detected an internal inconsistency in how many mortal scalars were allocated and freed.

Undefined format "%s" called

(F) The format indicated doesn't seem to exist. Perhaps it's really in another package? See Section 24.1 [perlform NAME], page 343.

Undefined sort subroutine "%s" called

(F) The sort comparison routine specified doesn't seem to exist. Perhaps it's in a different package? See ⟨undefined⟩ [perlfunc sort], page ⟨undefined⟩.

Undefined subroutine &%s called

(F) The subroutine indicated hasn't been defined, or if it was, it has since been undefined.

Undefined subroutine called

(F) The anonymous subroutine you're trying to call hasn't been defined, or if it was, it has since been undefined.

Undefined subroutine in sort

(F) The sort comparison routine specified is declared but doesn't seem to have been defined yet. See ⟨undefined⟩ [perlfunc sort], page ⟨undefined⟩.

Undefined top format "%s" called

(F) The format indicated doesn't seem to exist. Perhaps it's really in another package? See Section 24.1 [perlform NAME], page 343.

Undefined value assigned to typeglob

(W misc) An undefined value was assigned to a typeglob, a la `*foo = undef`. This does nothing. It's possible that you really mean `undef *foo`.

%s: Undefined variable

(A) You've accidentally run your script through **csh** instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

Unescaped left brace in regex is deprecated, passed through in regex; marked by <– HERE in m/%s/

(D deprecated, regexp) You used a literal `"{"` character in a regular expression pattern. You should change to use `"\{"` instead, because a future version of Perl (tentatively v5.26) will consider this to be a syntax error. If the pattern delimiters are also braces, any matching right brace (`"}"`) should also be escaped to avoid confusing the parser, for example,

```
    qr{abc\{def\}ghi}
```

unexec of %s into %s failed!

(F) The unexec() routine failed for some reason. See your local FSF representative, who probably put it there in the first place.

Unexpected binary operator '%c' with no preceding operand in regex; marked by <– HERE in m/%s/

(F) You had something like this:

```
 (?[ | \p{Digit} ])
```

where the `"|"` is a binary operator with an operand on the right, but no operand on the left.

Unexpected character in regex; marked by <– HERE in m/%s/

(F) You had something like this:

```
 (?[ z ])
```

Within `(?[ ])`, no literal characters are allowed unless they are within an inner pair of square brackets, like

```
 (?[ [ z ] ])
```

Another possibility is that you forgot a backslash. Perl isn't smart enough to figure out what you really meant.

Unexpected constant lvalue entersub entry via type/targ %d:%d

(P) When compiling a subroutine call in lvalue context, Perl failed an internal consistency check. It encountered a malformed op tree.

Unexpected exit %u

(S) exit() was called or the script otherwise finished gracefully when `PERL_EXIT_WARN` was set in `PL_exit_flags`.

Unexpected exit failure %d

(S) An uncaught die() was called when `PERL_EXIT_WARN` was set in `PL_exit_flags`.

Unexpected ')' in regex; marked by <– HERE in m/%s/

(F) You had something like this:

```
 (?[ ( \p{Digit} + ) ])
```

The `")"` is out-of-place. Something apparently was supposed to be combined with the digits, or the `"+"` shouldn't be there, or something like that. Perl can't figure out what was intended.

Unexpected '(' with no preceding operator in regex; marked by <– HERE in m/%s/

(F) You had something like this:

```
 (?[ \p{Digit} ( \p{Lao} + \p{Thai} ) ])
```

There should be an operator before the `"("`, as there's no indication as to how the digits are to be combined with the characters in the Lao and Thai scripts.

Unicode non-character U+%X is illegal for open interchange

(S nonchar) Certain codepoints, such as U+FFFE and U+FFFF, are defined by the Unicode standard to be non-characters. Those are legal codepoints, but are reserved for internal use; so, applications shouldn't attempt to exchange them. An application may not be expecting any of these characters at all, and receiving them may lead to bugs. If you know what you are doing you can turn off this warning by `no warnings 'nonchar';`.

This is not really a "severe" error, but it is supposed to be raised by default even if warnings are not enabled, and currently the only way to do that in Perl is to mark it as serious.

Unicode surrogate U+%X is illegal in UTF-8

(S surrogate) You had a UTF-16 surrogate in a context where they are not considered acceptable. These code points, between U+D800 and U+DFFF (inclusive), are used by Unicode only for UTF-16. However, Perl internally allows all unsigned integer code points (up to the size limit available on your platform), including surrogates. But these can cause problems when being input or output, which is likely where this message came from. If you really really know what you are doing you can turn off this warning by `no warnings 'surrogate';`.

Unknown charname '%s'

(F) The name you used inside `\N{}` is unknown to Perl. Check the spelling. You can say `use charnames ":loose"` to not have to be so precise about spaces, hyphens, and capitalization on standard Unicode names. (Any custom aliases that have been created must be specified exactly, regardless of whether `:loose` is used or not.) This error may also happen if the `\N{}` is not in the scope of the corresponding `use charnames`.

Unknown error

(P) Perl was about to print an error message in `$@`, but the `$@` variable did not exist, even after an attempt to create it.

Unknown open() mode '%s'

(F) The second argument of 3-argument open() is not among the list of valid modes: <, >, >>, +<, +>, +>>, -|, |-, <&, >&.

Unknown PerlIO layer "%s"

(W layer) An attempt was made to push an unknown layer onto the Perl I/O system. (Layers take care of transforming data between external and internal representations.) Note that some layers, such as mmap, are not supported in all environments. If your program didn't explicitly request the failing operation, it may be the result of the value of the environment variable PERLIO.

Unknown process %x sent message to prime_env_iter: %s

(P) An error peculiar to VMS. Perl was reading values for %ENV before iterating over it, and someone else stuck a message in the stream of data Perl expected. Someone's very confused, or perhaps trying to subvert Perl's population of %ENV for nefarious purposes.

Unknown regex modifier "%s"

(F) Alphanumerics immediately following the closing delimiter of a regular expression pattern are interpreted by Perl as modifier flags for the regex. One of the ones you specified is invalid. One way this can happen is if you didn't put in white space between the end of the regex and a following alphanumeric operator:

```
 if ($a =~ /foo/and $bar == 3) { ... }
```

The "a" is a valid modifier flag, but the "n" is not, and raises this error. Likely what was meant instead was:

```
 if ($a =~ /foo/ and $bar == 3) { ... }
```

Unknown "re" subpragma '%s' (known ones are: %s)

(W) You tried to use an unknown subpragma of the "re" pragma.

Unknown switch condition (?(...)) in regex; marked by <– HERE in m/%s/

(F) The condition part of a (?(condition)if-clause|else-clause) construct is not known. The condition must be one of the following:

```
 (1) (2) ...         true if 1st, 2nd, etc., capture matched
 (<NAME>) ('NAME')   true if named capture matched
 (?=...) (?<=...)    true if subpattern matches
 (?!...) (?<!...)    true if subpattern fails to match
 (?{ CODE })         true if code returns a true value
 (R)                 true if evaluating inside recursion
 (R1) (R2) ...       true if directly inside capture group 1, 2, etc.
 (R&NAME)            true if directly inside named capture
 (DEFINE)            always false; for defining named subpatterns
```

The <– HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Unknown Unicode option letter '%c'

(F) You specified an unknown Unicode option. See Section 69.1 [perlrun NAME], page 1176 documentation of the -C switch for the list of known options.

Unknown Unicode option value %d

(F) You specified an unknown Unicode option. See Section 69.1 [perlrun NAME], page 1176 documentation of the -C switch for the list of known options.

Unknown verb pattern '%s' in regex; marked by <– HERE in m/%s/

(F) You either made a typo or have incorrectly put a * quantifier after an open brace in your pattern. Check the pattern and review Section 58.1 [perlre NAME], page 989 for details on legal verb patterns.

Unknown warnings category '%s'

(F) An error issued by the warnings pragma. You specified a warnings category that is unknown to perl at this point.

Note that if you want to enable a warnings category registered by a module (e.g. use warnings 'File::Find'), you must have loaded this module first.

Unmatched '[' in POSIX class in regex; marked by <– HERE in m/%s/

(F) You had something like this:

```
(?[ [:digit: ])
```

That should be written:

```
(?[ [:digit:] ])
```

Unmatched '%c' in POSIX class in regex; marked by <– HERE in m/%s/

(F) You had something like this:

```
(?[ [:alnum] ])
```

There should be a second ":", like this:

```
(?[ [:alnum:] ])
```

Unmatched [in regex; marked by <– HERE in m/%s/

(F) The brackets around a character class must match. If you wish to include a closing bracket in a character class, backslash it or put it first. The <– HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Unmatched (in regex; marked by <– HERE in m/%s/
Unmatched) in regex; marked by <– HERE in m/%s/

(F) Unbackslashed parentheses must always be balanced in regular expressions. If you're a vi user, the % key is valuable for finding the matching parenthesis. The <– HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Unmatched right %s bracket

(F) The lexer counted more closing curly or square brackets than opening ones, so you're probably missing a matching opening bracket. As a general rule, you'll find the missing one (so to speak) near the place you were last editing.

Unquoted string "%s" may clash with future reserved word

(W reserved) You used a bareword that might someday be claimed as a reserved word. It's best to put such a word in quotes, or capitalize it somehow, or insert an underbar into it. You might also declare it as a subroutine.

Unrecognized character %s; marked by <– HERE after %s near column %d

(F) The Perl parser has no idea what to do with the specified character in your Perl script (or eval) near the specified column. Perhaps you tried to run a compressed script, a binary program, or a directory as a Perl program.

Unrecognized escape \%c in character class in regex; marked by <– HERE in m/%s/

(F) You used a backslash-character combination which is not recognized by Perl inside character classes. This is a fatal error when the character class is used within `(?[ ])`.

Unrecognized escape \%c in character class passed through in regex; marked by <– HERE in m/%s/

(W regexp) You used a backslash-character combination which is not recognized by Perl inside character classes. The character was understood literally, but this may change in a future version of Perl. The <– HERE shows whereabouts in the regular expression the escape was discovered.

Unrecognized escape \%c passed through

(W misc) You used a backslash-character combination which is not recognized by Perl. The character was understood literally, but this may change in a future version of Perl.

Unrecognized escape \%s passed through in regex; marked by <– HERE in m/%s/

(W regexp) You used a backslash-character combination which is not recognized by Perl. The character(s) were understood literally, but this may change in a future version of Perl. The <– HERE shows whereabouts in the regular expression the escape was discovered.

Unrecognized signal name "%s"

(F) You specified a signal name to the kill() function that was not recognized. Say `kill -l` in your shell to see the valid signal names on your system.

Unrecognized switch: -%s (-h will show valid options)

(F) You specified an illegal option to Perl. Don't do that. (If you think you didn't do that, check the #! line to see if it's supplying the bad switch on your behalf.)

unshift on reference is experimental

(S experimental::autoderef) `unshift` with a scalar argument is experimental and may change or be removed in a future Perl version. If you want to take the risk of using this feature, simply disable this warning:

```
    no warnings "experimental::autoderef";
```

Unsuccessful %s on filename containing newline

(W newline) A file operation was attempted on a filename, and that operation failed, PROBABLY because the filename contained a newline, PROBABLY because you forgot to chomp() it off. See [perlfunc chomp], page 363.

Unsupported directory function "%s" called

(F) Your machine doesn't support opendir() and readdir().

Unsupported function %s

(F) This machine doesn't implement the indicated function, apparently. At least, Configure doesn't think so.

Unsupported function fork

(F) Your version of executable does not support forking.

Note that under some systems, like OS/2, there may be different flavors of Perl executables, some of which may support fork, some not. Try changing the name you call Perl by to `perl_`, `perl__`, and so on.

Unsupported script encoding %s

(F) Your program file begins with a Unicode Byte Order Mark (BOM) which declares it to be in a Unicode encoding that Perl cannot read.

Unsupported socket function "%s" called

(F) Your machine doesn't support the Berkeley socket mechanism, or at least that's what Configure thought.

Unterminated attribute list

(F) The lexer found something other than a simple identifier at the start of an attribute, and it wasn't a semicolon or the start of a block. Perhaps you terminated the parameter list of the previous attribute too soon. See `attributes`.

Unterminated attribute parameter in attribute list

(F) The lexer saw an opening (left) parenthesis character while parsing an attribute list, but the matching closing (right) parenthesis character was not found. You may need to add (or remove) a backslash character to get your parentheses to balance. See `attributes`.

Unterminated compressed integer

(F) An argument to unpack("w",...) was incompatible with the BER compressed integer format and could not be converted to an integer. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

Unterminated delimiter for here document

(F) This message occurs when a here document label has an initial quotation mark but the final quotation mark is missing. Perhaps you wrote:

```
<<"foo
```

instead of:

```
<<"foo"
```

Unterminated \g... pattern in regex; marked by <-- HERE in m/%s/
Unterminated \g{...} pattern in regex; marked by <-- HERE in m/%s/

(F) In a regular expression, you had a `\g` that wasn't followed by a proper group reference. In the case of `\g{`, the closing brace is missing; otherwise the `\g` must be followed by an integer. Fix the pattern and retry.

Unterminated `<>` operator

(F) The lexer saw a left angle bracket in a place where it was expecting a term, so it's looking for the corresponding right angle bracket, and not finding it. Chances are you left some needed parentheses out earlier in the line, and you really meant a "less than".

Unterminated verb pattern argument in regex; marked by <– HERE in m/%s/

(F) You used a pattern of the form `(*VERB:ARG)` but did not terminate the pattern with a `)`. Fix the pattern and retry.

Unterminated verb pattern in regex; marked by <– HERE in m/%s/

(F) You used a pattern of the form `(*VERB)` but did not terminate the pattern with a `)`. Fix the pattern and retry.

untie attempted while %d inner references still exist

(W untie) A copy of the object returned from `tie` (or `tied`) was still valid when `untie` was called.

Usage: POSIX::%s(%s)

(F) You called a POSIX function with incorrect arguments. See Section "FUNCTIONS" in `POSIX` for more information.

Usage: Win32::%s(%s)

(F) You called a Win32 function with incorrect arguments. See `Win32` for more information.

$[used in %s (did you mean $] ?)

(W syntax) You used `$[` in a comparison, such as:

```
if ($[ > 5.006) {
    ...
}
```

You probably meant to use `$]` instead. `$[` is the base for indexing arrays. `$]` is the Perl version number in decimal.

Use "%s" instead of "%s"

(F) The second listed construct is no longer legal. Use the first one instead.

Useless assignment to a temporary

(W misc) You assigned to an lvalue subroutine, but what the subroutine returned was a temporary scalar about to be discarded, so the assignment had no effect.

Useless (?-%s) - don't use /%s modifier in regex; marked by <– HERE in m/%s/

(W regexp) You have used an internal modifier such as (?-o) that has no meaning unless removed from the entire regexp:

```
if ($string =~ /(?-o)$pattern/o) { ... }
```

must be written as

```
if ($string =~ /$pattern/) { ... }
```

The <– HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Useless localization of %s

(W syntax) The localization of lvalues such as `local($x=10)` is legal, but in fact the local() currently has no effect. This may change at some point in the future, but in the meantime such code is discouraged.

Useless (?%s) - use /%s modifier in regex; marked by <-- HERE in m/%s/

(W regexp) You have used an internal modifier such as (?o) that has no meaning unless applied to the entire regexp:

```
    if ($string =~ /(?o)$pattern/) { ... }
```

must be written as

```
    if ($string =~ /$pattern/o) { ... }
```

The <-- HERE shows whereabouts in the regular expression the problem was discovered. See Section 58.1 [perlre NAME], page 989.

Useless use of attribute "const"

(W misc) The "const" attribute has no effect except on anonymous closure prototypes. You applied it to a subroutine via `attributes`. This is only useful inside an attribute handler for an anonymous subroutine.

Useless use of /d modifier in transliteration operator

(W misc) You have used the /d modifier where the searchlist has the same length as the replacelist. See Section 48.1 [perlop NAME], page 798 for more information about the /d modifier.

Useless use of \E

(W misc) You have a \E in a double-quotish string without a `\U`, `\L` or `\Q` preceding it.

Useless use of greediness modifier '%c' in regex; marked by <-- HERE in m/%s/

(W regexp) You specified something like these:

```
qr/a{3}?/
qr/b{1,1}+/
```

The "?" and "+" don't have any effect, as they modify whether to match more or fewer when there is a choice, and by specifying to match exactly a given numer, there is no room left for a choice.

Useless use of %s in void context

(W void) You did something without a side effect in a context that does nothing with the return value, such as a statement that doesn't return a value from a block, or the left side of a scalar comma operator. Very often this points not to stupidity on your part, but a failure of Perl to parse your program the way you thought it would. For example, you'd get this if you mixed up your C precedence with Python precedence and said

```
    $one, $two = 1, 2;
```

when you meant to say

```
    ($one, $two) = (1, 2);
```

Another common error is to use ordinary parentheses to construct a list reference when you should be using square or curly brackets, for example, if you say

```
    $array = (1,2);
```

when you should have said

```
$array = [1,2];
```

The square brackets explicitly turn a list value into a scalar value, while parentheses do not. So when a parenthesized list is evaluated in a scalar context, the comma is treated like C's comma operator, which throws away the left argument, which is not what you want. See Section 62.1 [perlref NAME], page 1077 for more on this.

This warning will not be issued for numerical constants equal to 0 or 1 since they are often used in statements like

```
1 while sub_with_side_effects();
```

String constants that would normally evaluate to 0 or 1 are warned about.

Useless use of (?-p) in regex; marked by <– HERE in m/%s/

(W regexp) The `p` modifier cannot be turned off once set. Trying to do so is futile.

Useless use of "re" pragma

(W) You did `use re;` without any arguments. That isn't very useful.

Useless use of sort in scalar context

(W void) You used sort in scalar context, as in :

```
my $x = sort @y;
```

This is not very useful, and perl currently optimizes this away.

Useless use of %s with no values

(W syntax) You used the push() or unshift() function with no arguments apart from the array, like `push(@x)` or `unshift(@foo)`. That won't usually have any effect on the array, so is completely useless. It's possible in principle that push(@tied_array) could have some effect if the array is tied to a class which implements a PUSH method. If so, you can write it as `push(@tied_array,())` to avoid this warning.

"use" not allowed in expression

(F) The "use" keyword is recognized and executed at compile time, and returns no useful value. See Section 40.1 [perlmod NAME], page 732.

Use of assignment to $[is deprecated

(D deprecated) The `$[` variable (index of the first element in an array) is deprecated. See [perlvar $[], page 1408.

Use of bare << to mean <<"" is deprecated

(D deprecated) You are now encouraged to use the explicitly quoted form if you wish to use an empty line as the terminator of the here-document.

Use of \b{} for non-UTF-8 locale is wrong. Assuming a UTF-8 locale

(W locale) You are matching a regular expression using locale rules, and a Unicode boundary is being matched, but the locale is not a Unicode one. This doesn't make sense. Perl will continue, assuming a Unicode (UTF-8) locale, but the results could well be wrong except if the locale happens to be ISO-8859-1 (Latin1) where this message is spurious and can be ignored.

Use of chdir(”) or chdir(undef) as chdir() deprecated

(D deprecated) chdir() with no arguments is documented to change to $ENV{HOME} or $ENV{LOGDIR}. chdir(undef) and chdir(”) share this behavior, but that has been deprecated. In future versions they will simply fail.

Be careful to check that what you pass to chdir() is defined and not blank, else you might find yourself in your home directory.

Use of /c modifier is meaningless in s///

(W regexp) You used the /c modifier in a substitution. The /c modifier is not presently meaningful in substitutions.

Use of /c modifier is meaningless without /g

(W regexp) You used the /c modifier with a regex operand, but didn’t use the /g modifier. Currently, /c is meaningful only when /g is used. (This may change in the future.)

Use of comma-less variable list is deprecated

(D deprecated) The values you give to a format should be separated by commas, not just aligned on a line.

Use of each() on hash after insertion without resetting hash iterator results in undefined behavior

(S internal) The behavior of `each()` after insertion is undefined; it may skip items, or visit items more than once. Consider using `keys()` instead of `each()`.

Use of := for an empty attribute list is not allowed

(F) The construction `my $x := 42` used to parse as equivalent to `my $x : = 42` (applying an empty attribute list to `$x`). This construct was deprecated in 5.12.0, and has now been made a syntax error, so `:=` can be reclaimed as a new operator in the future.

If you need an empty attribute list, for example in a code generator, add a space before the `=`.

Use of freed value in iteration

(F) Perhaps you modified the iterated array within the loop? This error is typically caused by code like the following:

```
@a = (3,4);
@a = () for (1,2,@a);
```

You are not supposed to modify arrays while they are being iterated over. For speed and efficiency reasons, Perl internally does not do full reference-counting of iterated items, hence deleting such an item in the middle of an iteration causes Perl to see a freed value.

Use of *glob{FILEHANDLE} is deprecated

(D deprecated) You are now encouraged to use the shorter *glob{IO} form to access the filehandle slot within a typeglob.

Use of /g modifier is meaningless in split

(W regexp) You used the /g modifier on the pattern for a `split` operator. Since `split` always tries to match the pattern repeatedly, the `/g` has no effect.

Use of "goto" to jump into a construct is deprecated

> (D deprecated) Using `goto` to jump from an outer scope into an inner scope is deprecated and should be avoided.

Use of inherited AUTOLOAD for non-method %s() is deprecated

> (D deprecated) As an (ahem) accidental feature, `AUTOLOAD` subroutines are looked up as methods (using the `@ISA` hierarchy) even when the subroutines to be autoloaded were called as plain functions (e.g. `Foo::bar()`), not as methods (e.g. `Foo->bar()` or `$obj->bar()`).
>
> This bug will be rectified in future by using method lookup only for methods' `AUTOLOAD`s. However, there is a significant base of existing code that may be using the old behavior. So, as an interim step, Perl currently issues an optional warning when non-methods use inherited `AUTOLOAD`s.
>
> The simple rule is: Inheritance will not work when autoloading non-methods. The simple fix for old code is: In any module that used to depend on inheriting `AUTOLOAD` for non-methods from a base class named `BaseClass`, execute `*AUTOLOAD = \&BaseClass::AUTOLOAD` during startup.
>
> In code that currently says `use AutoLoader; @ISA = qw(AutoLoader);` you should remove AutoLoader from @ISA and change `use AutoLoader;` to `use AutoLoader 'AUTOLOAD';`.

Use of %s in printf format not supported

> (F) You attempted to use a feature of printf that is accessible from only C. This usually means there's a better way to do it in Perl.

Use of %s is deprecated

> (D deprecated) The construct indicated is no longer recommended for use, generally because there's a better way to do it, and also because the old way has bad side effects.

Use of literal control characters in variable names is deprecated
Use of literal non-graphic characters in variable names is deprecated

> (D deprecated) Using literal non-graphic (including control) characters in the source to refer to the ^FOO variables, like `$^X` and `${^GLOBAL_PHASE}` is now deprecated. (We use `^X` and `^G` here for legibility. They actually represent the non-printable control characters, code points 0x18 and 0x07, respectively; `^A` would mean the control character whose code point is 0x01.) This only affects code like `$\cT`, where `\cT` is a control in the source code; `${"\cT"}` and `$^T` remain valid. Things that are non-controls and also not graphic are NO-BREAK SPACE and SOFT HYPHEN, which were previously only allowed for historical reasons.

Use of -l on filehandle%s

> (W io) A filehandle represents an opened file, and when you opened the file it already went past any symlink you are presumably trying to look for. The operation returned `undef`. Use a filename instead.

Use of my $_ is experimental

(S experimental::lexical_topic) Lexical $_ is an experimental feature and its behavior may change or even be removed in any future release of perl. See the explanation under [perlvar $_], page 1376.

Use of %s on a handle without * is deprecated

(D deprecated) You used `tie`, `tied` or `untie` on a scalar but that scalar happens to hold a typeglob, which means its filehandle will be tied. If you mean to tie a handle, use an explicit * as in `tie *$handle`.

This was a long-standing bug that was removed in Perl 5.16, as there was no way to tie the scalar itself when it held a typeglob, and no way to untie a scalar that had had a typeglob assigned to it. If you see this message, you must be using an older version.

Use of reference "%s" as array index

(W misc) You tried to use a reference as an array index; this probably isn't what you mean, because references in numerical context tend to be huge numbers, and so usually indicates programmer error.

If you really do mean it, explicitly numify your reference, like so: `$array[0+$ref]`. This warning is not given for overloaded objects, however, because you can overload the numification and stringification operators and then you presumably know what you are doing.

Use of state $_ is experimental

(S experimental::lexical_topic) Lexical $_ is an experimental feature and its behavior may change or even be removed in any future release of perl. See the explanation under [perlvar $_], page 1376.

Use of tainted arguments in %s is deprecated

(W taint, deprecated) You have supplied `system()` or `exec()` with multiple arguments and at least one of them is tainted. This used to be allowed but will become a fatal error in a future version of perl. Untaint your arguments. See Section 70.1 [perlsec NAME], page 1198.

Use of uninitialized value%s

(W uninitialized) An undefined value was used as if it were already defined. It was interpreted as a `""` or a 0, but maybe it was a mistake. To suppress this warning assign a defined value to your variables.

To help you figure out what was undefined, perl will try to tell you the name of the variable (if any) that was undefined. In some cases it cannot do this, so it also tells you what operation you used the undefined value in. Note, however, that perl optimizes your program and the operation displayed in the warning may not necessarily appear literally in your program. For example, `"that $foo"` is usually optimized into `"that " . $foo`, and the warning will refer to the `concatenation (.)` operator, even though there is no `.` in your program.

"use re 'strict'" is experimental

(S experimental::re_strict) The things that are different when a regular expression pattern is compiled under `'strict'` are subject to change in future Perl releases in incompatible ways. This means that a pattern that compiles today may not in a future Perl release. This warning is to alert you to that risk.

Use \x{...} for more than two hex characters in regex; marked by <– HERE in m/%s/

(F) In a regular expression, you said something like

```
 (?[ [ \xBEEF ] ])
```

Perl isn't sure if you meant this

```
 (?[ [ \x{BEEF} ] ])
```

or if you meant this

```
 (?[ [ \x{BE} E F ] ])
```

You need to add either braces or blanks to disambiguate.

Using just the first character returned by \N{} in character class in regex; marked by <– HERE in m/%s/

(W regexp) Named Unicode character escapes (`\N{...}`) may return a multicharacter sequence. Even though a character class is supposed to match just one character of input, perl will match the whole thing correctly, except when the class is inverted (`[^...]`), or the escape is the beginning or final end point of a range. For these, what should happen isn't clear at all. In these circumstances, Perl discards all but the first character of the returned sequence, which is not likely what you want.

Using /u for '%s' instead of /%s in regex; marked by <– HERE in m/%s/

(W regexp) You used a Unicode boundary (`\b{...}` or `\B{...}`) in a portion of a regular expression where the character set modifiers `/a` or `/aa` are in effect. These two modifiers indicate an ASCII interpretation, and this doesn't make sense for a Unicode defintion. The generated regular expression will compile so that the boundary uses all of Unicode. No other portion of the regular expression is affected.

Using !~ with %s doesn't make sense

(F) Using the `!~` operator with `s///r`, `tr///r` or `y///r` is currently reserved for future use, as the exact behavior has not been decided. (Simply returning the boolean opposite of the modified string is usually not particularly useful.)

UTF-16 surrogate U+%X

(S surrogate) You had a UTF-16 surrogate in a context where they are not considered acceptable. These code points, between U+D800 and U+DFFF (inclusive), are used by Unicode only for UTF-16. However, Perl internally allows all unsigned integer code points (up to the size limit available on your platform), including surrogates. But these can cause problems when being input or output, which is likely where this message came from. If you really really know what you are doing you can turn off this warning by `no warnings 'surrogate';`.

Value of %s can be "0"; test with defined()

(W misc) In a conditional expression, you used <HANDLE>, <*> (glob), `each()`, or `readdir()` as a boolean value. Each of these constructs can return a value of "0"; that would make the conditional expression false, which is probably not what you intended. When using these constructs in conditional expressions, test their values with the `defined` operator.

Value of CLI symbol "%s" too long

(W misc) A warning peculiar to VMS. Perl tried to read the value of an %ENV element from a CLI symbol table, and found a resultant string longer than 1024 characters. The return value has been truncated to 1024 characters.

values on reference is experimental

(S experimental::autoderef) `values` with a scalar argument is experimental and may change or be removed in a future Perl version. If you want to take the risk of using this feature, simply disable this warning:

```
    no warnings "experimental::autoderef";
```

Variable "%s" is not available

(W closure) During compilation, an inner named subroutine or eval is attempting to capture an outer lexical that is not currently available. This can happen for one of two reasons. First, the outer lexical may be declared in an outer anonymous subroutine that has not yet been created. (Remember that named subs are created at compile time, while anonymous subs are created at runtime.) For example,

```
    sub { my $a; sub f { $a } }
```

At the time that f is created, it can't capture the current value of $a, since the anonymous subroutine hasn't been created yet. Conversely, the following won't give a warning since the anonymous subroutine has by now been created and is live:

```
    sub { my $a; eval 'sub f { $a }' }->();
```

The second situation is caused by an eval accessing a variable that has gone out of scope, for example,

```
    sub f {
        my $a;
        sub { eval '$a' }
    }
    f()->();
```

Here, when the '$a' in the eval is being compiled, f() is not currently being executed, so its $a is not available for capture.

Variable "%s" is not imported%s

(S misc) With "use strict" in effect, you referred to a global variable that you apparently thought was imported from another module, because something else of the same name (usually a subroutine) is exported by that module. It usually means you put the wrong funny character on the front of your variable.

Variable length lookbehind not implemented in regex m/%s/

(F) Lookbehind is allowed only for subexpressions whose length is fixed and known at compile time. For positive lookbehind, you can use the `\K` regex construct as a way to get the equivalent functionality. See [perlre (?<=pattern) \K], page 1008.

There are non-obvious Unicode rules under `/i` that can match variably, but which you might not think could. For example, the substring `"ss"` can match the single character LATIN SMALL LETTER SHARP S. There are other sequences of ASCII characters that can match single ligature characters, such as LATIN SMALL LIGATURE FFI matching `qr/ffi/i`. Starting in Perl v5.16, if you only care about ASCII matches, adding the `/aa` modifier to the regex will exclude all these non-obvious matches, thus getting rid of this message. You can also say `use re qw(/aa)` to apply `/aa` to all regular expressions compiled within its scope. See `re`.

"%s" variable %s masks earlier declaration in same %s

(W misc) A "my", "our" or "state" variable has been redeclared in the current scope or statement, effectively eliminating all access to the previous instance. This is almost always a typographical error. Note that the earlier variable will still exist until the end of the scope or until all closure references to it are destroyed.

Variable syntax

(A) You've accidentally run your script through **csh** instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

Variable "%s" will not stay shared

(W closure) An inner (nested) *named* subroutine is referencing a lexical variable defined in an outer named subroutine.

When the inner subroutine is called, it will see the value of the outer subroutine's variable as it was before and during the *first* call to the outer subroutine; in this case, after the first call to the outer subroutine is complete, the inner and outer subroutines will no longer share a common value for the variable. In other words, the variable will no longer be shared.

This problem can usually be solved by making the inner subroutine anonymous, using the `sub {}` syntax. When inner anonymous subs that reference variables in outer subroutines are created, they are automatically rebound to the current values of such variables.

vector argument not supported with alpha versions

(S printf) The %vd (s)printf format does not support version objects with alpha parts.

Verb pattern '%s' has a mandatory argument in regex; marked by <– HERE in m/%s/

(F) You used a verb pattern that requires an argument. Supply an argument or check that you are using the right verb.

Verb pattern '%s' may not have an argument in regex; marked by <– HERE in m/%s/

(F) You used a verb pattern that is not allowed an argument. Remove the argument or check that you are using the right verb.

Version number must be a constant number

(P) The attempt to translate a `use Module n.n LIST` statement into its equivalent `BEGIN` block found an internal inconsistency with the version number.

Version string '%s' contains invalid data; ignoring: '%s'

(W misc) The version string contains invalid characters at the end, which are being ignored.

Warning: something's wrong

(W) You passed warn() an empty string (the equivalent of `warn ""`) or you called it with no args and `$@` was empty.

Warning: unable to close filehandle %s properly

(S) The implicit close() done by an open() got an error indication on the close(). This usually indicates your file system ran out of disk space.

Warning: unable to close filehandle properly: %s
Warning: unable to close filehandle %s properly: %s

(S io) An error occurred when Perl implicitly closed a filehandle. This usually indicates your file system ran out of disk space.

Warning: Use of "%s" without parentheses is ambiguous

(S ambiguous) You wrote a unary operator followed by something that looks like a binary operator that could also have been interpreted as a term or unary operator. For instance, if you know that the rand function has a default argument of 1.0, and you write

```
rand + 5;
```

you may THINK you wrote the same thing as

```
rand() + 5;
```

but in actual fact, you got

```
rand(+5);
```

So put in parentheses to say what you really mean.

when is experimental

(S experimental::smartmatch) `when` depends on smartmatch, which is experimental. Additionally, it has several special cases that may not be immediately obvious, and their behavior may change or even be removed in any future release of perl. See the explanation under Section 74.2.16 [perlsyn Experimental Details on given and when], page 1263.

Wide character in %s

(S utf8) Perl met a wide character (>255) when it wasn't expecting one. This warning is by default on for I/O (like print). The easiest way to quiet this warning is simply to add the `:utf8` layer to the output, e.g. `binmode STDOUT, ':utf8'`. Another way to turn off the warning is to add `no warnings 'utf8';` but that is often closer to cheating. In general, you are supposed to explicitly mark the filehandle with an encoding, see `open` and ⟨undefined⟩ [perlfunc binmode], page ⟨undefined⟩.

Wide character (U+%X) in %s

(W locale) While in a single-byte locale (*i.e.*, a non-UTF-8 one), a multi-byte character was encountered. Perl considers this character to be the specified Unicode code point. Combining non-UTF-8 locales and Unicode is dangerous. Almost certainly some characters will have two different representations. For example, in the ISO 8859-7 (Greek) locale, the code point 0xC3 represents a Capital Gamma. But so also does 0x393. This will make string comparisons unreliable.

You likely need to figure out how this multi-byte character got mixed up with your single-byte locale (or perhaps you thought you had a UTF-8 locale, but Perl disagrees).

Within []-length '%c' not allowed

(F) The count in the (un)pack template may be replaced by `[TEMPLATE]` only if `TEMPLATE` always matches the same amount of packed bytes that can be determined from the template alone. This is not possible if it contains any of the codes @, /, U, u, w or a *-length. Redesign the template.

write() on closed filehandle %s

(W closed) The filehandle you're writing to got itself closed sometime before now. Check your control flow.

%s "\x%X" does not map to Unicode

(S utf8) When reading in different encodings, Perl tries to map everything into Unicode characters. The bytes you read in are not legal in this encoding. For example

```
utf8 "\xE4" does not map to Unicode
```

if you try to read in the a-diaereses Latin-1 as UTF-8.

'X' outside of string

(F) You had a (un)pack template that specified a relative position before the beginning of the string being (un)packed. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

'x' outside of string in unpack

(F) You had a pack template that specified a relative position after the end of the string being unpacked. See ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩.

YOU HAVEN'T DISABLED SET-ID SCRIPTS IN THE KERNEL YET!

(F) And you probably never will, because you probably don't have the sources to your kernel, and your vendor probably doesn't give a rip about what you want. Your best bet is to put a setuid C wrapper around your script.

You need to quote "%s"

(W syntax) You assigned a bareword as a signal handler name. Unfortunately, you already have a subroutine of that name declared, which means that Perl 5 will try to call the subroutine when the assignment is executed, which is probably not what you want. (If it IS what you want, put an & in front.)

Your random numbers are not that random

(F) When trying to initialize the random seed for hashes, Perl could not get any randomness out of your system. This usually indicates Something Very Wrong.

Zero length \N{} in regex; marked by <– HERE in m/%s/

(F) Named Unicode character escapes (`\N{...}`) may return a zero-length sequence. Such an escape was used in an extended character class, i.e. `(?[...])`, which is not permitted. Check that the correct escape has been used, and the correct charnames handler is in scope. The <– HERE shows whereabouts in the regular expression the problem was discovered.

16.3 SEE ALSO

`warnings`, `diagnostics`.

17 perldsc

17.1 NAME

perldsc - Perl Data Structures Cookbook

17.2 DESCRIPTION

Perl lets us have complex data structures. You can write something like this and all of a sudden, you'd have an array with three dimensions!

```
for my $x (1 .. 10) {
    for my $y (1 .. 10) {
        for my $z (1 .. 10) {
            $AoA[$x][$y][$z] =
                $x ** $y + $z;
        }
    }
}
```

Alas, however simple this may appear, underneath it's a much more elaborate construct than meets the eye!

How do you print it out? Why can't you say just `print @AoA`? How do you sort it? How can you pass it to a function or get one of these back from a function? Is it an object? Can you save it to disk to read back later? How do you access whole rows or columns of that matrix? Do all the values have to be numeric?

As you see, it's quite easy to become confused. While some small portion of the blame for this can be attributed to the reference-based implementation, it's really more due to a lack of existing documentation with examples designed for the beginner.

This document is meant to be a detailed but understandable treatment of the many different sorts of data structures you might want to develop. It should also serve as a cookbook of examples. That way, when you need to create one of these complex data structures, you can just pinch, pilfer, or purloin a drop-in example from here.

Let's look at each of these possible constructs in detail. There are separate sections on each of the following:

- arrays of arrays
- hashes of arrays
- arrays of hashes
- hashes of hashes
- more elaborate constructs

But for now, let's look at general issues common to all these types of data structures.

17.3 REFERENCES

The most important thing to understand about all data structures in Perl–including multidimensional arrays–is that even though they might appear otherwise, Perl @ARRAYs and %HASHes are all internally one-dimensional. They can hold only scalar values (meaning a string, number, or a reference). They cannot directly contain other arrays or hashes, but instead contain *references* to other arrays or hashes.

You can't use a reference to an array or hash in quite the same way that you would a real array or hash. For C or C++ programmers unused to distinguishing between arrays and pointers to the same, this can be confusing. If so, just think of it as the difference between a structure and a pointer to a structure.

You can (and should) read more about references in Section 62.1 [perlref NAME], page 1077. Briefly, references are rather like pointers that know what they point to. (Objects are also a kind of reference, but we won't be needing them right away–if ever.) This means that when you have something which looks to you like an access to a two-or-more-dimensional array and/or hash, what's really going on is that the base type is merely a one-dimensional entity that contains references to the next level. It's just that you can *use* it as though it were a two-dimensional one. This is actually the way almost all C multidimensional arrays work as well.

```
    $array[7][12]                       # array of arrays
    $array[7]{string}                   # array of hashes
    $hash{string}[7]                    # hash of arrays
    $hash{string}{'another string'}     # hash of hashes
```

Now, because the top level contains only references, if you try to print out your array in with a simple print() function, you'll get something that doesn't look very nice, like this:

```
    my @AoA = ( [2, 3], [4, 5, 7], [0] );
    print $AoA[1][2];
  7
    print @AoA;
  ARRAY(0x83c38)ARRAY(0x8b194)ARRAY(0x8b1d0)
```

That's because Perl doesn't (ever) implicitly dereference your variables. If you want to get at the thing a reference is referring to, then you have to do this yourself using either prefix typing indicators, like `${$blah}`, `@{$blah}`, `@{$blah[$i]}`, or else postfix pointer arrows, like `$a->[3]`, `$h->{fred}`, or even `$ob->method()->[3]`.

17.4 COMMON MISTAKES

The two most common mistakes made in constructing something like an array of arrays is either accidentally counting the number of elements or else taking a reference to the same memory location repeatedly. Here's the case where you just get the count instead of a nested array:

```
    for my $i (1..10) {
        my @array = somefunc($i);
        $AoA[$i] = @array;      # WRONG!
    }
```

That's just the simple case of assigning an array to a scalar and getting its element count. If that's what you really and truly want, then you might do well to consider being a tad more explicit about it, like this:

```
for my $i (1..10) {
    my @array = somefunc($i);
    $counts[$i] = scalar @array;
}
```

Here's the case of taking a reference to the same memory location again and again:

```
# Either without strict or having an outer-scope my @array;
# declaration.

for my $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = \@array;     # WRONG!
}
```

So, what's the big problem with that? It looks right, doesn't it? After all, I just told you that you need an array of references, so by golly, you've made me one!

Unfortunately, while this is true, it's still broken. All the references in @AoA refer to the *very same place*, and they will therefore all hold whatever was last in @array! It's similar to the problem demonstrated in the following C program:

```
#include <pwd.h>
main() {
    struct passwd *getpwnam(), *rp, *dp;
    rp = getpwnam("root");
    dp = getpwnam("daemon");

    printf("daemon name is %s\nroot name is %s\n",
            dp->pw_name, rp->pw_name);
}
```

Which will print

```
daemon name is daemon
root name is daemon
```

The problem is that both `rp` and `dp` are pointers to the same location in memory! In C, you'd have to remember to malloc() yourself some new memory. In Perl, you'll want to use the array constructor `[]` or the hash constructor `{}` instead. Here's the right way to do the preceding broken code fragments:

```
# Either without strict or having an outer-scope my @array;
# declaration.

for my $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = [ @array ];
}
```

The square brackets make a reference to a new array with a *copy* of what's in @array at the time of the assignment. This is what you want.

Note that this will produce something similar, but it's much harder to read:

```
# Either without strict or having an outer-scope my @array;
# declaration.
for my $i (1..10) {
    @array = 0 .. $i;
    @{$AoA[$i]} = @array;
}
```

Is it the same? Well, maybe so–and maybe not. The subtle difference is that when you assign something in square brackets, you know for sure it's always a brand new reference with a new *copy* of the data. Something else could be going on in this new case with the `@{$AoA[$i]}` dereference on the left-hand-side of the assignment. It all depends on whether `$AoA[$i]` had been undefined to start with, or whether it already contained a reference. If you had already populated @AoA with references, as in

```
$AoA[3] = \@another_array;
```

Then the assignment with the indirection on the left-hand-side would use the existing reference that was already there:

```
@{$AoA[3]} = @array;
```

Of course, this *would* have the "interesting" effect of clobbering @another_array. (Have you ever noticed how when a programmer says something is "interesting", that rather than meaning "intriguing", they're disturbingly more apt to mean that it's "annoying", "difficult", or both? :-)

So just remember always to use the array or hash constructors with `[]` or `{}`, and you'll be fine, although it's not always optimally efficient.

Surprisingly, the following dangerous-looking construct will actually work out fine:

```
for my $i (1..10) {
    my @array = somefunc($i);
    $AoA[$i] = \@array;
}
```

That's because my() is more of a run-time statement than it is a compile-time declaration *per se*. This means that the my() variable is remade afresh each time through the loop. So even though it *looks* as though you stored the same variable reference each time, you actually did not! This is a subtle distinction that can produce more efficient code at the risk of misleading all but the most experienced of programmers. So I usually advise against teaching it to beginners. In fact, except for passing arguments to functions, I seldom like to see the gimme-a-reference operator (backslash) used much at all in code. Instead, I advise beginners that they (and most of the rest of us) should try to use the much more easily understood constructors `[]` and `{}` instead of relying upon lexical (or dynamic) scoping and hidden reference-counting to do the right thing behind the scenes.

In summary:

```
$AoA[$i] = [ @array ];     # usually best
$AoA[$i] = \@array;        # perilous; just how my() was that array?
@{ $AoA[$i] } = @array;    # way too tricky for most programmers
```

17.5 CAVEAT ON PRECEDENCE

Speaking of things like `@{$AoA[$i]}`, the following are actually the same thing: >>

```
$aref->[2][2]       # clear
$$aref[2][2]        # confusing
```

That's because Perl's precedence rules on its five prefix dereferencers (which look like someone swearing: `$ @ * % &`) make them bind more tightly than the postfix subscripting brackets or braces! This will no doubt come as a great shock to the C or C++ programmer, who is quite accustomed to using `*a[i]` to mean what's pointed to by the *i'th* element of `a`. That is, they first take the subscript, and only then dereference the thing at that subscript. That's fine in C, but this isn't C.

The seemingly equivalent construct in Perl, `$$aref[$i]` first does the deref of $aref, making it take $aref as a reference to an array, and then dereference that, and finally tell you the *i'th* value of the array pointed to by $AoA. If you wanted the C notion, you'd have to write `${$AoA[$i]}` to force the `$AoA[$i]` to get evaluated first before the leading `$` dereferencer.

17.6 WHY YOU SHOULD ALWAYS `use strict`

If this is starting to sound scarier than it's worth, relax. Perl has some features to help you avoid its most common pitfalls. The best way to avoid getting confused is to start every program like this:

```
#!/usr/bin/perl -w
use strict;
```

This way, you'll be forced to declare all your variables with my() and also disallow accidental "symbolic dereferencing". Therefore if you'd done this:

```
my $aref = [
    [ "fred", "barney", "pebbles", "bambam", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "elroy", "judy", ],
];

print $aref[2][2];
```

The compiler would immediately flag that as an error *at compile time*, because you were accidentally accessing `@aref`, an undeclared variable, and it would thereby remind you to write instead:

```
print $aref->[2][2]
```

17.7 DEBUGGING

You can use the debugger's `x` command to dump out complex data structures. For example, given the assignment to $AoA above, here's the debugger output:

```
DB<1> x $AoA
$AoA = ARRAY(0x13b5a0)
   0  ARRAY(0x1f0a24)
      0  'fred'
      1  'barney'
      2  'pebbles'
      3  'bambam'
      4  'dino'
   1  ARRAY(0x13b558)
      0  'homer'
      1  'bart'
      2  'marge'
      3  'maggie'
   2  ARRAY(0x13b540)
      0  'george'
      1  'jane'
      2  'elroy'
      3  'judy'
```

17.8 CODE EXAMPLES

Presented with little comment (these will get their own manpages someday) here are short code examples illustrating access of various types of data structures.

17.9 ARRAYS OF ARRAYS

17.9.1 Declaration of an ARRAY OF ARRAYS

```
@AoA = (
       [ "fred", "barney" ],
       [ "george", "jane", "elroy" ],
       [ "homer", "marge", "bart" ],
     );
```

17.9.2 Generation of an ARRAY OF ARRAYS

```
# reading from file
while ( <> ) {
    push @AoA, [ split ];
}

# calling a function
for $i ( 1 .. 10 ) {
    $AoA[$i] = [ somefunc($i) ];
}

# using temp vars
for $i ( 1 .. 10 ) {
    @tmp = somefunc($i);
    $AoA[$i] = [ @tmp ];
}

# add to an existing row
```

```
push @{ $AoA[0] }, "wilma", "betty";
```

17.9.3 Access and Printing of an ARRAY OF ARRAYS

```
# one element
$AoA[0][0] = "Fred";

# another element
$AoA[1][1] =~ s/(\w)/\u$1/;

# print the whole thing with refs
for $aref ( @AoA ) {
    print "\t [ @$aref ],\n";
}

# print the whole thing with indices
for $i ( 0 .. $#AoA ) {
    print "\t [ @{$AoA[$i]} ],\n";
}

# print the whole thing one at a time
for $i ( 0 .. $#AoA ) {
    for $j ( 0 .. $#{ $AoA[$i] } ) {
        print "elt $i $j is $AoA[$i][$j]\n";
    }
}
```

17.10 HASHES OF ARRAYS

17.10.1 Declaration of a HASH OF ARRAYS

```
%HoA = (
       flintstones        => [ "fred", "barney" ],
       jetsons            => [ "george", "jane", "elroy" ],
       simpsons           => [ "homer", "marge", "bart" ],
     );
```

17.10.2 Generation of a HASH OF ARRAYS

```
# reading from file
# flintstones: fred barney wilma dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $HoA{$1} = [ split ];
}

# reading from file; more temps
# flintstones: fred barney wilma dino
while ( $line = <> ) {
    ($who, $rest) = split /:\s*/, $line, 2;
    @fields = split ’ ’, $rest;
    $HoA{$who} = [ @fields ];
}

# calling a function that returns a list
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoA{$group} = [ get_family($group) ];
}

# likewise, but using temps
for $group ( "simpsons", "jetsons", "flintstones" ) {
    @members = get_family($group);
    $HoA{$group} = [ @members ];
}

# append new members to an existing family
push @{ $HoA{"flintstones"} }, "wilma", "betty";
```

17.10.3 Access and Printing of a HASH OF ARRAYS

```
# one element
$HoA{flintstones}[0] = "Fred";

# another element
$HoA{simpsons}[1] =~ s/(\w)/\u$1/;

# print the whole thing
foreach $family ( keys %HoA ) {
    print "$family: @{ $HoA{$family} }\n"
}

# print the whole thing with indices
foreach $family ( keys %HoA ) {
    print "family: ";
    foreach $i ( 0 .. $#{ $HoA{$family} } ) {
        print " $i = $HoA{$family}[$i]";
    }
    print "\n";
}

# print the whole thing sorted by number of members
foreach $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {
    print "$family: @{ $HoA{$family} }\n"
}

# print the whole thing sorted by number of members and name
```

```
foreach $family ( sort {
                           @{$HoA{$b}} <=> @{$HoA{$a}}
                                        ||
                                   $a cmp $b
           } keys %HoA )
{
    print "$family: ", join(", ", sort @{ $HoA{$family} }), "\n";
}
```

17.11 ARRAYS OF HASHES

17.11.1 Declaration of an ARRAY OF HASHES

```
@AoH = (
       {
           Lead     => "fred",
           Friend   => "barney",
       },
       {
           Lead     => "george",
           Wife     => "jane",
           Son      => "elroy",
       },
       {
           Lead     => "homer",
           Wife     => "marge",
           Son      => "bart",
       }
 );
```

17.11.2 Generation of an ARRAY OF HASHES

```
# reading from file
# format: LEAD=fred FRIEND=barney
while ( <> ) {
    $rec = {};
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
    push @AoH, $rec;
}


# reading from file
# format: LEAD=fred FRIEND=barney
# no temp
while ( <> ) {
    push @AoH, { split /[\s+=]/ };
}

# calling a function  that returns a key/value pair list, like
# "lead","fred","daughter","pebbles"
while ( %fields = getnextpairset() ) {
    push @AoH, { %fields };
}

# likewise, but using no temp vars
while (<>) {
    push @AoH, { parsepairs($_) };
}

# add key/value to an element
$AoH[0]{pet} = "dino";
$AoH[2]{pet} = "santa’s little helper";
```

17.11.3 Access and Printing of an ARRAY OF HASHES

```
# one element
$AoH[0]{lead} = "fred";

# another element
$AoH[1]{lead} =˜ s/(\w)/\u$1/;

# print the whole thing with refs
for $href ( @AoH ) {
    print "{ ";
    for $role ( keys %$href ) {
        print "$role=$href->{$role} ";
    }
    print "}\n";
}

# print the whole thing with indices
for $i ( 0 .. $#AoH ) {
    print "$i is { ";
    for $role ( keys %{ $AoH[$i] } ) {
        print "$role=$AoH[$i]{$role} ";
    }
    print "}\n";
}

# print the whole thing one at a time
for $i ( 0 .. $#AoH ) {
    for $role ( keys %{ $AoH[$i] } ) {
        print "elt $i $role is $AoH[$i]{$role}\n";
    }
}
```

17.12 HASHES OF HASHES

17.12.1 Declaration of a HASH OF HASHES

```
%HoH = (
       flintstones => {
               lead      => "fred",
               pal       => "barney",
       },
       jetsons     => {
               lead      => "george",
               wife      => "jane",
               "his boy" => "elroy",
       },
       simpsons    => {
               lead      => "homer",
               wife      => "marge",
               kid       => "bart",
       },
);
```

17.12.2 Generation of a HASH OF HASHES

```
# reading from file
# flintstones: lead=fred pal=barney wife=wilma pet=dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $HoH{$who}{$key} = $value;
    }


# reading from file; more temps
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    $rec = {};
    $HoH{$who} = $rec;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
}

# calling a function  that returns a key,value hash
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoH{$group} = { get_family($group) };
}

# likewise, but using temps
for $group ( "simpsons", "jetsons", "flintstones" ) {
    %members = get_family($group);
    $HoH{$group} = { %members };
}

# append new members to an existing family
%new_folks = (
    wife => "wilma",
    pet  => "dino",
);

for $what (keys %new_folks) {
    $HoH{flintstones}{$what} = $new_folks{$what};
}
```

17.12.3 Access and Printing of a HASH OF HASHES

```
# one element
$HoH{flintstones}{wife} = "wilma";

# another element
$HoH{simpsons}{lead} =~ s/(\w)/\u$1/;

# print the whole thing
foreach $family ( keys %HoH ) {
    print "$family: { ";
    for $role ( keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# print the whole thing  somewhat sorted
foreach $family ( sort keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}


# print the whole thing sorted by number of members
foreach $family ( sort { keys %{$HoH{$b}} <=> keys %{$HoH{$a}} }
                                                            keys %HoH )
{
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# establish a sort order (rank) for each role
$i = 0;
for ( qw(lead wife son daughter pal pet) ) { $rank{$_} = ++$i }

# now print the whole thing sorted by number of members
foreach $family ( sort { keys %{ $HoH{$b} } <=> keys %{ $HoH{$a} } }
                                                            keys %HoH )
{
    print "$family: { ";
    # and print these according to rank order
    for $role ( sort { $rank{$a} <=> $rank{$b} }
                                              keys %{ $HoH{$family} } )
    {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}
```

17.13 MORE ELABORATE RECORDS

17.13.1 Declaration of MORE ELABORATE RECORDS

Here's a sample showing how to create and use a record whose fields are of many different sorts:

```
    $rec = {
        TEXT      => $string,
        SEQUENCE  => [ @old_values ],
        LOOKUP    => { %some_table },
        THATCODE  => \&some_function,
        THISCODE  => sub { $_[0] ** $_[1] },
        HANDLE    => \*STDOUT,
    };
```

```
print $rec->{TEXT};

print $rec->{SEQUENCE}[0];
$last = pop @ { $rec->{SEQUENCE} };

print $rec->{LOOKUP}{"key"};
($first_k, $first_v) = each %{ $rec->{LOOKUP} };

$answer = $rec->{THATCODE}->($arg);
$answer = $rec->{THISCODE}->($arg1, $arg2);

# careful of extra block braces on fh ref
print { $rec->{HANDLE} } "a string\n";

use FileHandle;
$rec->{HANDLE}->autoflush(1);
$rec->{HANDLE}->print(" a string\n");
```

17.13.2 Declaration of a HASH OF COMPLEX RECORDS

```
%TV = (
   flintstones => {
       series   => "flintstones",
       nights   => [ qw(monday thursday friday) ],
       members  => [
           { name => "fred",    role => "lead", age  => 36, },
           { name => "wilma",   role => "wife", age  => 31, },
           { name => "pebbles", role => "kid",  age  =>  4, },
       ],
   },

   jetsons     => {
       series   => "jetsons",
       nights   => [ qw(wednesday saturday) ],
       members  => [
           { name => "george",  role => "lead", age  => 41, },
           { name => "jane",    role => "wife", age  => 39, },
           { name => "elroy",   role => "kid",  age  =>  9, },
       ],
    },

   simpsons    => {
       series   => "simpsons",
       nights   => [ qw(monday) ],
       members  => [
           { name => "homer", role => "lead", age  => 34, },
           { name => "marge", role => "wife", age => 37, },
           { name => "bart",  role => "kid",  age  =>  11, },
       ],
    },
 );
```

17.13.3 Generation of a HASH OF COMPLEX RECORDS

```
# reading from file
# this is most easily done by having the file itself be
# in the raw data format as shown above.  perl is happy
# to parse complex data structures if declared as data, so
# sometimes it's easiest to do that

# here's a piece by piece build up
$rec = {};
$rec->{series} = "flintstones";
$rec->{nights} = [ find_days() ];

@members = ();
# assume this file in field=value syntax
while (<>) {
    %fields = split /[\s=]+/;
    push @members, { %fields };
}
$rec->{members} = [ @members ];

# now remember the whole thing
$TV{ $rec->{series} } = $rec;

###########################################################
# now, you might want to make interesting extra fields that
# include pointers back into the same data structure so if
# change one piece, it changes everywhere, like for example
# if you wanted a {kids} field that was a reference
# to an array of the kids' records without having duplicate
# records and thus update problems.
###########################################################
foreach $family (keys %TV) {
    $rec = $TV{$family}; # temp pointer
    @kids = ();
    for $person ( @{ $rec->{members} } ) {
        if ($person->{role} =~ /kid|son|daughter/) {
            push @kids, $person;
        }
    }
    # REMEMBER: $rec and $TV{$family} point to same data!!
    $rec->{kids} = [ @kids ];
}

# you copied the array, but the array itself contains pointers
# to uncopied objects. this means that if you make bart get
# older via

$TV{simpsons}{kids}[0]{age}++;

# then this would also change in
print $TV{simpsons}{members}[2]{age};

# because $TV{simpsons}{kids}[0] and $TV{simpsons}{members}[2]
# both point to the same underlying anonymous hash table

# print the whole thing
foreach $family ( keys %TV ) {
    print "the $family";
    print " is on during @{ $TV{$family}{nights} }\n";
    print "its members are:\n";
    for $who ( @{ $TV{$family}{members} } ) {
        print " $who->{name} ($who->{role}), age $who->{age}\n";
    }
    print "it turns out that $TV{$family}{lead} has ";
    print scalar ( @{ $TV{$family}{kids} } ), " kids named ";
    print join (", ", map { $_->{name} } @{ $TV{$family}{kids} } );
    print "\n";
}
```

17.14 Database Ties

You cannot easily tie a multilevel data structure (such as a hash of hashes) to a dbm file. The first problem is that all but GDBM and Berkeley DB have size limitations, but beyond that, you also have problems with how references are to be represented on disk. One experimental module that does partially attempt to address this need is the MLDBM module. Check your nearest CPAN site as described in `perlmodlib` for source code to MLDBM.

17.15 SEE ALSO

Section 62.1 [perlref NAME], page 1077, Section 39.1 [perllol NAME], page 725, Section 11.1 [perldata NAME], page 70, Section 46.1 [perlobj NAME], page 769

17.16 AUTHOR

Tom Christiansen `<tchrist@perl.com>`

18 perldtrace

18.1 NAME

perldtrace - Perl's support for DTrace

18.2 SYNOPSIS

```
# dtrace -Zn 'perl::sub-entry, perl::sub-return { trace(copyinstr(arg0)) }'
dtrace: description 'perl::sub-entry, perl::sub-return ' matched 10 probes

# perl -E 'sub outer { inner(@_) } sub inner { say shift } outer("hello")'
hello

(dtrace output)
CPU     ID                    FUNCTION:NAME
  0  75915       Perl_pp_entersub:sub-entry   BEGIN
  0  75915       Perl_pp_entersub:sub-entry   import
  0  75922      Perl_pp_leavesub:sub-return   import
  0  75922      Perl_pp_leavesub:sub-return   BEGIN
  0  75915       Perl_pp_entersub:sub-entry   outer
  0  75915       Perl_pp_entersub:sub-entry   inner
  0  75922      Perl_pp_leavesub:sub-return   inner
  0  75922      Perl_pp_leavesub:sub-return   outer
```

18.3 DESCRIPTION

DTrace is a framework for comprehensive system- and application-level tracing. Perl is a DTrace *provider*, meaning it exposes several *probes* for instrumentation. You can use these in conjunction with kernel-level probes, as well as probes from other providers such as MySQL, in order to diagnose software defects, or even just your application's bottlenecks.

Perl must be compiled with the `-Dusedtrace` option in order to make use of the provided probes. While DTrace aims to have no overhead when its instrumentation is not active, Perl's support itself cannot uphold that guarantee, so it is built without DTrace probes under most systems. One notable exception is that Mac OS X ships a `/usr/bin/perl` with DTrace support enabled.

18.4 HISTORY

5.10.1

Perl's initial DTrace support was added, providing `sub-entry` and `sub-return` probes.

5.14.0

The `sub-entry` and `sub-return` probes gain a fourth argument: the package name of the function.

5.16.0

The `phase-change` probe was added.

5.18.0

The `op-entry`, `loading-file`, and `loaded-file` probes were added.

18.5 PROBES

sub-entry(SUBNAME, FILE, LINE, PACKAGE)

Traces the entry of any subroutine. Note that all of the variables refer to the subroutine that is being invoked; there is currently no way to get ahold of any information about the subroutine's *caller* from a DTrace action.

```
:*perl*::sub-entry {
    printf("%s::%s entered at %s line %d\n",
           copyinstr(arg3), copyinstr(arg0), copyinstr(arg1), arg2);
}
```

sub-return(SUBNAME, FILE, LINE, PACKAGE)

Traces the exit of any subroutine. Note that all of the variables refer to the subroutine that is returning; there is currently no way to get ahold of any information about the subroutine's *caller* from a DTrace action.

```
:*perl*::sub-return {
    printf("%s::%s returned at %s line %d\n",
           copyinstr(arg3), copyinstr(arg0), copyinstr(arg1), arg2);
}
```

phase-change(NEWPHASE, OLDPHASE)

Traces changes to Perl's interpreter state. You can internalize this as tracing changes to Perl's `${^GLOBAL_PHASE}` variable, especially since the values for `NEWPHASE` and `OLDPHASE` are the strings that `${^GLOBAL_PHASE}` reports.

```
:*perl*::phase-change {
    printf("Phase changed from %s to %s\n",
        copyinstr(arg1), copyinstr(arg0));
}
```

op-entry(OPNAME)

Traces the execution of each opcode in the Perl runloop. This probe is fired before the opcode is executed. When the Perl debugger is enabled, the DTrace probe is fired *after* the debugger hooks (but still before the opcode itself is executed).

```
:*perl*::op-entry {
    printf("About to execute opcode %s\n", copyinstr(arg0));
}
```

loading-file(FILENAME)

Fires when Perl is about to load an individual file, whether from `use`, `require`, or `do`. This probe fires before the file is read from disk. The filename argument is converted to local filesystem paths instead of providing `Module::Name`-style names.

```
:*perl*:loading-file {
    printf("About to load %s\n", copyinstr(arg0));
}
```

loaded-file(FILENAME)

Fires when Perl has successfully loaded an individual file, whether from `use`, `require`, or `do`. This probe fires after the file is read from disk and its contents evaluated. The filename argument is converted to local filesystem paths instead of providing `Module::Name`-style names.

```
:*perl*:loaded-file {
    printf("Successfully loaded %s\n", copyinstr(arg0));
}
```

18.6 EXAMPLES

Most frequently called functions

```
# dtrace -qZn 'sub-entry { @[strjoin(strjoin(copyinstr(arg3),"::"),copyinstr(

Class::MOP::Attribute::slots                                      400
Try::Tiny::catch                                                  411
Try::Tiny::try                                                    411
Class::MOP::Instance::inline_slot_access                          451
Class::MOP::Class::Immutable::Trait:::around                      472
Class::MOP::Mixin::AttributeCore::has_initializer                 496
Class::MOP::Method::Wrapped::__ANON__                             544
Class::MOP::Package::_package_stash                               737
Class::MOP::Class::initialize                                    1128
Class::MOP::get_metaclass_by_name                                1204
```

Trace function calls

```
# dtrace -qFZn 'sub-entry, sub-return { trace(copyinstr(arg0)) }'

0  -> Perl_pp_entersub                        BEGIN
0  <- Perl_pp_leavesub                        BEGIN
0  -> Perl_pp_entersub                        BEGIN
0    -> Perl_pp_entersub                      import
0    <- Perl_pp_leavesub                      import
0  <- Perl_pp_leavesub                        BEGIN
0  -> Perl_pp_entersub                        BEGIN
0    -> Perl_pp_entersub                      dress
0    <- Perl_pp_leavesub                      dress
0    -> Perl_pp_entersub                      dirty
0    <- Perl_pp_leavesub                      dirty
0    -> Perl_pp_entersub                      whiten
0    <- Perl_pp_leavesub                      whiten
0  <- Perl_dounwind                           BEGIN
```

Function calls during interpreter cleanup

```
# dtrace -Zn 'phase-change /copyinstr(arg0) == "END"/ { self->ending = 1 } su
```

```
CPU     ID                    FUNCTION:NAME
  1  77214       Perl_pp_entersub:sub-entry   END
  1  77214       Perl_pp_entersub:sub-entry   END
  1  77214       Perl_pp_entersub:sub-entry   cleanup
  1  77214       Perl_pp_entersub:sub-entry   _force_writable
  1  77214       Perl_pp_entersub:sub-entry   _force_writable
```

System calls at compile time

```
# dtrace -qZn 'phase-change /copyinstr(arg0) == "START"/ { self->interesting

lseek                                                                 310
read                                                                  374
stat64                                                               1056
```

Perl functions that execute the most opcodes

```
# dtrace -qZn 'sub-entry { self->fqn = strjoin(copyinstr(arg3), strjoin("::",

warnings::unimport                                                   4589
Exporter::Heavy::_rebuild_cache                                      5039
Exporter::import                                                    14578
```

18.7 REFERENCES

DTrace Dynamic Tracing Guide
: http://dtrace.org/guide/preface.html

DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD
: http://www.amazon.com/DTrace-Dynamic-Tracing-Solaris-FreeBSD/dp/0132091518/

18.8 SEE ALSO

Devel-DTrace-Provider
: This CPAN module lets you create application-level DTrace probes written in Perl.

18.9 AUTHORS

Shawn M Moore sartak@gmail.com

19 perlebcdic

19.1 NAME

perlebcdic - Considerations for running Perl on EBCDIC platforms

19.2 DESCRIPTION

An exploration of some of the issues facing Perl programmers on EBCDIC based computers.

Portions of this document that are still incomplete are marked with XXX.

Early Perl versions worked on some EBCDIC machines, but the last known version that ran on EBCDIC was v5.8.7, until v5.22, when the Perl core again works on z/OS. Theoretically, it could work on OS/400 or Siemens' BS2000 (or their successors), but this is untested. In v5.22, not all the modules found on CPAN but shipped with core Perl work on z/OS.

If you want to use Perl on a non-z/OS EBCDIC machine, please let us know by sending mail to perlbug@perl.org

Writing Perl on an EBCDIC platform is really no different than writing on an Section 19.3.1 [ASCII], page 267 one, but with different underlying numbers, as we'll see shortly. You'll have to know something about those Section 19.3.1 [ASCII], page 267 platforms because the documentation is biased and will frequently use example numbers that don't apply to EBCDIC. There are also very few CPAN modules that are written for EBCDIC and which don't work on ASCII; instead the vast majority of CPAN modules are written for ASCII, and some may happen to work on EBCDIC, while a few have been designed to portably work on both.

If your code just uses the 52 letters A-Z and a-z, plus SPACE, the digits 0-9, and the punctuation characters that Perl uses, plus a few controls that are denoted by escape sequences like `\n` and `\t`, then there's nothing special about using Perl, and your code may very well work on an ASCII machine without change.

But if you write code that uses `\005` to mean a TAB or `\xC1` to mean an `"A"`, or `\xDF` to mean a `"ÿ"` (small `"y"` with a diaeresis), then your code may well work on your EBCDIC platform, but not on an ASCII one. That's fine to do if no one will ever want to run your code on an ASCII platform; but the bias in this document will be in writing code portable between EBCDIC and ASCII systems. Again, if every character you care about is easily enterable from your keyboard, you don't have to know anything about ASCII, but many keyboards don't easily allow you to directly enter, say, the character `\xDF`, so you have to specify it indirectly, such as by using the `"\xDF"` escape sequence. In those cases it's easiest to know something about the ASCII/Unicode character sets. If you know that the small `"ÿ"` is `U+00FF`, then you can instead specify it as `"\N{U+FF}"`, and have the computer automatically translate it to `\xDF` on your platform, and leave it as `\xFF` on ASCII ones. Or you could specify it by name, `\N{LATIN SMALL LETTER Y WITH DIAERESIS` and not have to know the numbers. Either way works, but require familiarity with Unicode.

19.3 COMMON CHARACTER CODE SETS

19.3.1 ASCII

The American Standard Code for Information Interchange (ASCII or US-ASCII) is a set of integers running from 0 to 127 (decimal) that have standardized interpretations by the computers which use ASCII. For example, 65 means the letter "A". The range 0..127 can be covered by setting the bits in a 7-bit binary digit, hence the set is sometimes referred to as "7-bit ASCII". ASCII was described by the American National Standards Institute document ANSI X3.4-1986. It was also described by ISO 646:1991 (with localization for currency symbols). The full ASCII set is given in the table [below], page 272 as the first 128 elements. Languages that can be written adequately with the characters in ASCII include English, Hawaiian, Indonesian, Swahili and some Native American languages.

Most non-EBCDIC character sets are supersets of ASCII. That is the integers 0-127 mean what ASCII says they mean. But integers 128 and above are specific to the character set.

Many of these fit entirely into 8 bits, using ASCII as 0-127, while specifying what 128-255 mean, and not using anything above 255. Thus, these are single-byte (or octet if you prefer) character sets. One important one (since Unicode is a superset of it) is the ISO 8859-1 character set.

19.3.2 ISO 8859

The ISO 8859-**$n** are a collection of character code sets from the International Organization for Standardization (ISO), each of which adds characters to the ASCII set that are typically found in various languages, many of which are based on the Roman, or Latin, alphabet. Most are for European languages, but there are also ones for Arabic, Greek, Hebrew, and Thai. There are good references on the web about all these.

19.3.3 Latin 1 (ISO 8859-1)

A particular 8-bit extension to ASCII that includes grave and acute accented Latin characters. Languages that can employ ISO 8859-1 include all the languages covered by ASCII as well as Afrikaans, Albanian, Basque, Catalan, Danish, Faroese, Finnish, Norwegian, Portuguese, Spanish, and Swedish. Dutch is covered albeit without the ij ligature. French is covered too but without the oe ligature. German can use ISO 8859-1 but must do so without German-style quotation marks. This set is based on Western European extensions to ASCII and is commonly encountered in world wide web work. In IBM character code set identification terminology, ISO 8859-1 is also known as CCSID 819 (or sometimes 0819 or even 00819).

19.3.4 EBCDIC

The Extended Binary Coded Decimal Interchange Code refers to a large collection of single- and multi-byte coded character sets that are quite different from ASCII and ISO 8859-1, and are all slightly different from each other; they typically run on host computers. The EBCDIC encodings derive from 8-bit byte extensions of Hollerith punched card encodings, which long predate ASCII. The layout on the cards was such that high bits were set for the upper and lower case alphabetic characters `[a-z]` and `[A-Z]`, but there were gaps within each Latin alphabet range, visible in the table [below], page 272. These gaps can cause complications.

Some IBM EBCDIC character sets may be known by character code set identification numbers (CCSID numbers) or code page numbers.

Perl can be compiled on platforms that run any of three commonly used EBCDIC character sets, listed below.

19.3.4.1 The 13 variant characters

Among IBM EBCDIC character code sets there are 13 characters that are often mapped to different integer values. Those characters are known as the 13 "variant" characters and are:

```
\ [ ] { } ^ ~ ! # | $ @ `
```

When Perl is compiled for a platform, it looks at all of these characters to guess which EBCDIC character set the platform uses, and adapts itself accordingly to that platform. If the platform uses a character set that is not one of the three Perl knows about, Perl will either fail to compile, or mistakenly and silently choose one of the three.

19.3.4.2 EBCDIC code sets recognized by Perl

0037

Character code set ID 0037 is a mapping of the ASCII plus Latin-1 characters (i.e. ISO 8859-1) to an EBCDIC set. 0037 is used in North American English locales on the OS/400 operating system that runs on AS/400 computers. CCSID 0037 differs from ISO 8859-1 in 236 places; in other words they agree on only 20 code point values.

1047

Character code set ID 1047 is also a mapping of the ASCII plus Latin-1 characters (i.e. ISO 8859-1) to an EBCDIC set. 1047 is used under Unix System Services for OS/390 or z/OS, and OpenEdition for VM/ESA. CCSID 1047 differs from CCSID 0037 in eight places, and from ISO 8859-1 in 236.

POSIX-BC

The EBCDIC code page in use on Siemens' BS2000 system is distinct from 1047 and 0037. It is identified below as the POSIX-BC set. Like 0037 and 1047, it is the same as ISO 8859-1 in 20 code point values.

19.3.5 Unicode code points versus EBCDIC code points

In Unicode terminology a *code point* is the number assigned to a character: for example, in EBCDIC the character "A" is usually assigned the number 193. In Unicode, the character "A" is assigned the number 65. All the code points in ASCII and Latin-1 (ISO 8859-1) have the same meaning in Unicode. All three of the recognized EBCDIC code sets have 256 code points, and in each code set, all 256 code points are mapped to equivalent Latin1 code points. Obviously, "A" will map to "A", "B" => "B", "%" => "%", etc., for all printable characters in Latin1 and these code pages.

It also turns out that EBCDIC has nearly precise equivalents for the ASCII/Latin1 C0 controls and the DELETE control. (The C0 controls are those whose ASCII code points are 0..0x1F; things like TAB, ACK, BEL, etc.) A mapping is set up between these ASCII/EBCDIC controls. There isn't such a precise mapping between the C1 controls on

ASCII platforms and the remaining EBCDIC controls. What has been done is to map these controls, mostly arbitrarily, to some otherwise unmatched character in the other character set. Most of these are very very rarely used nowadays in EBCDIC anyway, and their names have been dropped, without much complaint. For example the EO (Eight Ones) EBCDIC control (consisting of eight one bits = 0xFF) is mapped to the C1 APC control (0x9F), and you can't use the name "EO".

The EBCDIC controls provide three possible line terminator characters, CR (0x0D), LF (0x25), and NL (0x15). On ASCII platforms, the symbols "NL" and "LF" refer to the same character, but in strict EBCDIC terminology they are different ones. The EBCDIC NL is mapped to the C1 control called "NEL" ("Next Line"; here's a case where the mapping makes quite a bit of sense, and hence isn't just arbitrary). On some EBCDIC platforms, this NL or NEL is the typical line terminator. This is true of z/OS and BS2000. In these platforms, the C compilers will swap the LF and NEL code points, so that "\n" is 0x15, and refers to NL. Perl does that too; you can see it in the code chart [below], page 272. This makes things generally "just work" without you even having to be aware that there is a swap.

19.3.6 Unicode and UTF

UTF stands for "Unicode Transformation Format". UTF-8 is an encoding of Unicode into a sequence of 8-bit byte chunks, based on ASCII and Latin-1. The length of a sequence required to represent a Unicode code point depends on the ordinal number of that code point, with larger numbers requiring more bytes. UTF-EBCDIC is like UTF-8, but based on EBCDIC. They are enough alike that often, casual usage will conflate the two terms, and use "UTF-8" to mean both the UTF-8 found on ASCII platforms, and the UTF-EBCDIC found on EBCDIC ones.

You may see the term "invariant" character or code point. This simply means that the character has the same numeric value and representation when encoded in UTF-8 (or UTF-EBCDIC) as when not. (Note that this is a very different concept from Section 19.3.4.1 [The 13 variant characters], page 268 mentioned above. Careful prose will use the term "UTF-8 invariant" instead of just "invariant", but most often you'll see just "invariant".) For example, the ordinal value of "A" is 193 in most EBCDIC code pages, and also is 193 when encoded in UTF-EBCDIC. All UTF-8 (or UTF-EBCDIC) variant code points occupy at least two bytes when encoded in UTF-8 (or UTF-EBCDIC); by definition, the UTF-8 (or UTF-EBCDIC) invariant code points are exactly one byte whether encoded in UTF-8 (or UTF-EBCDIC), or not. (By now you see why people typically just say "UTF-8" when they also mean "UTF-EBCDIC". For the rest of this document, we'll mostly be casual about it too.) In ASCII UTF-8, the code points corresponding to the lowest 128 ordinal numbers (0 - 127: the ASCII characters) are invariant. In UTF-EBCDIC, there are 160 invariant characters. (If you care, the EBCDIC invariants are those characters which have ASCII equivalents, plus those that correspond to the C1 controls (128 - 159 on ASCII platforms).)

A string encoded in UTF-EBCDIC may be longer (but never shorter) than one encoded in UTF-8. Perl extends UTF-8 so that it can encode code points above the Unicode maximum of U+10FFFF. It extends UTF-EBCDIC as well, but due to the inherent limitations in UTF-EBCDIC, the maximum code point expressible is U+7FFF_FFFF, even if the word size is more than 32 bits.

UTF-EBCDIC is defined by Unicode Technical Report #16 (http://www.unicode.org/reports/tr16). It is defined based on CCSID 1047, not allowing for the differences for other code pages. This allows for easy interchange of text between computers running different code pages, but makes it unusable, without adaptation, for Perl on those other code pages.

The reason for this unusability is that a fundamental assumption of Perl is that the characters it cares about for parsing and lexical analysis are the same whether or not the text is in UTF-8. For example, Perl expects the character "[" to have the same representation, no matter if the string containing it (or program text) is UTF-8 encoded or not. To ensure this, Perl adapts UTF-EBCDIC to the particular code page so that all characters it expects to be UTF-8 invariant are in fact UTF-8 invariant. This means that text generated on a computer running one version of Perl's UTF-EBCDIC has to be translated to be intelligible to a computer running another.

19.3.7 Using Encode

Starting from Perl 5.8 you can use the standard module Encode to translate from EBCDIC to Latin-1 code points. Encode knows about more EBCDIC character sets than Perl can currently be compiled to run on.

```
use Encode 'from_to';

my %ebcdic = ( 176 => 'cp37', 95 => 'cp1047', 106 => 'posix-bc' );

# $a is in EBCDIC code points
from_to($a, $ebcdic{ord '^'}, 'latin1');
# $a is ISO 8859-1 code points
```

and from Latin-1 code points to EBCDIC code points

```
use Encode 'from_to';

my %ebcdic = ( 176 => 'cp37', 95 => 'cp1047', 106 => 'posix-bc' );

# $a is ISO 8859-1 code points
from_to($a, 'latin1', $ebcdic{ord '^'});
# $a is in EBCDIC code points
```

For doing I/O it is suggested that you use the autotranslating features of PerlIO, see Section 83.1 [perluniintro NAME], page 1352.

Since version 5.8 Perl uses the PerlIO I/O library. This enables you to use different encodings per IO channel. For example you may use

```
use Encode;
open($f, ">:encoding(ascii)", "test.ascii");
print $f "Hello World!\n";
open($f, ">:encoding(cp37)", "test.ebcdic");
print $f "Hello World!\n";
open($f, ">:encoding(latin1)", "test.latin1");
print $f "Hello World!\n";
open($f, ">:encoding(utf8)", "test.utf8");
```

```
    print $f "Hello World!\n";
```

to get four files containing "Hello World!\n" in ASCII, CP 0037 EBCDIC, ISO 8859-1 (Latin-1) (in this example identical to ASCII since only ASCII characters were printed), and UTF-EBCDIC (in this example identical to normal EBCDIC since only characters that don't differ between EBCDIC and UTF-EBCDIC were printed). See the documentation of `Encode-PerlIO` for details.

As the PerlIO layer uses raw IO (bytes) internally, all this totally ignores things like the type of your filesystem (ASCII or EBCDIC).

19.4 SINGLE OCTET TABLES

The following tables list the ASCII and Latin 1 ordered sets including the subsets: C0 controls (0..31), ASCII graphics (32..7e), delete (7f), C1 controls (80..9f), and Latin-1 (a.k.a. ISO 8859-1) (a0..ff). In the table names of the Latin 1 extensions to ASCII have been labelled with character names roughly corresponding to *The Unicode Standard, Version 6.1* albeit with substitutions such as `s/LATIN//` and `s/VULGAR//` in all cases; `s/CAPITAL LETTER//` in some cases; and `s/SMALL LETTER ([A-Z])/\l$1/` in some other cases. Controls are listed using their Unicode 6.2 abbreviations. The differences between the 0037 and 1047 sets are flagged with `**`. The differences between the 1047 and POSIX-BC sets are flagged with `##`. All `ord()` numbers listed are decimal. If you would rather see this table listing octal values, then run the table (that is, the pod source text of this document, since this recipe may not work with a pod2_other_format translation) through:

recipe 0

```
    perl -ne 'if(/(.{29})(\d+)\s+(\d+)\s+(\d+)\s+(\d+)/)' \
     -e '{printf("%s%-5.03o%-5.03o%-5.03o%.03o\n",$1,$2,$3,$4,$5)}' \
     perlebcdic.pod
```

If you want to retain the UTF-x code points then in script form you might want to write:

recipe 1

```
 open(FH,"<perlebcdic.pod") or die "Could not open perlebcdic.pod: $!";
 while (<FH>) {
     if (/(.{29})(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\.?(\d*)
                                                     \s+(\d+)\.?(\d*)/x)
     {
         if ($7 ne '' && $9 ne '') {
             printf(
                "%s%-5.03o%-5.03o%-5.03o%-5.03o%-3o.%-5o%-3o.%.03o\n",
                                            $1,$2,$3,$4,$5,$6,$7,$8,$9);
         }
         elsif ($7 ne '') {
             printf("%s%-5.03o%-5.03o%-5.03o%-5.03o%-3o.%-5o%.03o\n",
                                           $1,$2,$3,$4,$5,$6,$7,$8);
         }
         else {
             printf("%s%-5.03o%-5.03o%-5.03o%-5.03o%-5.03o%.03o\n",
                                                $1,$2,$3,$4,$5,$6,$8);
```

```
        }
    }
}
```

If you would rather see this table listing hexadecimal values then run the table through:

recipe 2

```
    perl -ne ’if(/(.{29})(\d+)\s+(\d+)\s+(\d+)\s+(\d+)/)’ \
     -e ’{printf("%s%-5.02X%-5.02X%-5.02X%.02X\n",$1,$2,$3,$4,$5)}’ \
     perlebcdic.pod
```

Or, in order to retain the UTF-x code points in hexadecimal:

recipe 3

```
open(FH,"<perlebcdic.pod") or die "Could not open perlebcdic.pod: $!";
while (<FH>) {
    if (/(.{29})(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\.?(\d*)
                                                  \s+(\d+)\.?(\d*)/x)
    {
        if ($7 ne ’’ && $9 ne ’’) {
            printf(
               "%s%-5.02X%-5.02X%-5.02X%-5.02X%-2X.%-6.02X%02X.%02X\n",
                                        $1,$2,$3,$4,$5,$6,$7,$8,$9);
        }
        elsif ($7 ne ’’) {
            printf("%s%-5.02X%-5.02X%-5.02X%-5.02X%-2X.%-6.02X%02X\n",
                                           $1,$2,$3,$4,$5,$6,$7,$8);
        }
        else {
            printf("%s%-5.02X%-5.02X%-5.02X%-5.02X%-5.02X%02X\n",
                                               $1,$2,$3,$4,$5,$6,$8);
        }
    }
}
```

```
                        ISO
                       8859-1            POS-        CCSID
                       CCSID  CCSID CCSID IX-         1047
 chr                    0819   0037 1047  BC  UTF-8  UTF-EBCDIC
------------------------------------------------------------------
<NUL>                      0    0    0    0    0        0
<SOH>                      1    1    1    1    1        1
<STX>                      2    2    2    2    2        2
<ETX>                      3    3    3    3    3        3
<EOT>                      4    55   55   55   4        55
<ENQ>                      5    45   45   45   5        45
<ACK>                      6    46   46   46   6        46
<BEL>                      7    47   47   47   7        47
```

<BS>	8	22	22	22	8	22	
<HT>	9	5	5	5	9	5	
<LF>	10	37	21	21	10	21	**
<VT>	11	11	11	11	11	11	
<FF>	12	12	12	12	12	12	
<CR>	13	13	13	13	13	13	
<SO>	14	14	14	14	14	14	
<SI>	15	15	15	15	15	15	
<DLE>	16	16	16	16	16	16	
<DC1>	17	17	17	17	17	17	
<DC2>	18	18	18	18	18	18	
<DC3>	19	19	19	19	19	19	
<DC4>	20	60	60	60	20	60	
<NAK>	21	61	61	61	21	61	
<SYN>	22	50	50	50	22	50	
<ETB>	23	38	38	38	23	38	
<CAN>	24	24	24	24	24	24	
<EOM>	25	25	25	25	25	25	
<SUB>	26	63	63	63	26	63	
<ESC>	27	39	39	39	27	39	
<FS>	28	28	28	28	28	28	
<GS>	29	29	29	29	29	29	
<RS>	30	30	30	30	30	30	
<US>	31	31	31	31	31	31	
<SPACE>	32	64	64	64	32	64	
!	33	90	90	90	33	90	
"	34	127	127	127	34	127	
#	35	123	123	123	35	123	
$	36	91	91	91	36	91	
%	37	108	108	108	37	108	
&	38	80	80	80	38	80	
'	39	125	125	125	39	125	
(	40	77	77	77	40	77	
)	41	93	93	93	41	93	
*	42	92	92	92	42	92	
+	43	78	78	78	43	78	
,	44	107	107	107	44	107	
-	45	96	96	96	45	96	
.	46	75	75	75	46	75	
/	47	97	97	97	47	97	
0	48	240	240	240	48	240	
1	49	241	241	241	49	241	
2	50	242	242	242	50	242	
3	51	243	243	243	51	243	
4	52	244	244	244	52	244	
5	53	245	245	245	53	245	
6	54	246	246	246	54	246	

7	55	247	247	247	55	247	
8	56	248	248	248	56	248	
9	57	249	249	249	57	249	
:	58	122	122	122	58	122	
;	59	94	94	94	59	94	
<	60	76	76	76	60	76	
=	61	126	126	126	61	126	
>	62	110	110	110	62	110	
?	63	111	111	111	63	111	
@	64	124	124	124	64	124	
A	65	193	193	193	65	193	
B	66	194	194	194	66	194	
C	67	195	195	195	67	195	
D	68	196	196	196	68	196	
E	69	197	197	197	69	197	
F	70	198	198	198	70	198	
G	71	199	199	199	71	199	
H	72	200	200	200	72	200	
I	73	201	201	201	73	201	
J	74	209	209	209	74	209	
K	75	210	210	210	75	210	
L	76	211	211	211	76	211	
M	77	212	212	212	77	212	
N	78	213	213	213	78	213	
O	79	214	214	214	79	214	
P	80	215	215	215	80	215	
Q	81	216	216	216	81	216	
R	82	217	217	217	82	217	
S	83	226	226	226	83	226	
T	84	227	227	227	84	227	
U	85	228	228	228	85	228	
V	86	229	229	229	86	229	
W	87	230	230	230	87	230	
X	88	231	231	231	88	231	
Y	89	232	232	232	89	232	
Z	90	233	233	233	90	233	
[	91	186	173	187	91	173	** ##
\	92	224	224	188	92	224	##
]	93	187	189	189	93	189	**
^	94	176	95	106	94	95	** ##
_	95	109	109	109	95	109	
‘	96	121	121	74	96	121	##
a	97	129	129	129	97	129	
b	98	130	130	130	98	130	
c	99	131	131	131	99	131	
d	100	132	132	132	100	132	
e	101	133	133	133	101	133	

f	102	134	134	134	102	134	
g	103	135	135	135	103	135	
h	104	136	136	136	104	136	
i	105	137	137	137	105	137	
j	106	145	145	145	106	145	
k	107	146	146	146	107	146	
l	108	147	147	147	108	147	
m	109	148	148	148	109	148	
n	110	149	149	149	110	149	
o	111	150	150	150	111	150	
p	112	151	151	151	112	151	
q	113	152	152	152	113	152	
r	114	153	153	153	114	153	
s	115	162	162	162	115	162	
t	116	163	163	163	116	163	
u	117	164	164	164	117	164	
v	118	165	165	165	118	165	
w	119	166	166	166	119	166	
x	120	167	167	167	120	167	
y	121	168	168	168	121	168	
z	122	169	169	169	122	169	
{	123	192	192	251	123	192	##
|	124	79	79	79	124	79	
}	125	208	208	253	125	208	##
~	126	161	161	255	126	161	##
<DEL>	127	7	7	7	127	7	
<PAD>	128	32	32	32	194.128	32	
<HOP>	129	33	33	33	194.129	33	
<BPH>	130	34	34	34	194.130	34	
<NBH>	131	35	35	35	194.131	35	
<IND>	132	36	36	36	194.132	36	
<NEL>	133	21	37	37	194.133	37	**
<SSA>	134	6	6	6	194.134	6	
<ESA>	135	23	23	23	194.135	23	
<HTS>	136	40	40	40	194.136	40	
<HTJ>	137	41	41	41	194.137	41	
<VTS>	138	42	42	42	194.138	42	
<PLD>	139	43	43	43	194.139	43	
<PLU>	140	44	44	44	194.140	44	
<RI>	141	9	9	9	194.141	9	
<SS2>	142	10	10	10	194.142	10	
<SS3>	143	27	27	27	194.143	27	
<DCS>	144	48	48	48	194.144	48	
<PU1>	145	49	49	49	194.145	49	
<PU2>	146	26	26	26	194.146	26	
<STS>	147	51	51	51	194.147	51	
<CCH>	148	52	52	52	194.148	52	

<MW>	149	53	53	53	194.149	53	
<SPA>	150	54	54	54	194.150	54	
<EPA>	151	8	8	8	194.151	8	
<SOS>	152	56	56	56	194.152	56	
<SGC>	153	57	57	57	194.153	57	
<SCI>	154	58	58	58	194.154	58	
<CSI>	155	59	59	59	194.155	59	
<ST>	156	4	4	4	194.156	4	
<OSC>	157	20	20	20	194.157	20	
<PM>	158	62	62	62	194.158	62	
<APC>	159	255	255	95	194.159	255	##
<NON-BREAKING SPACE>	160	65	65	65	194.160	128.65	
<INVERTED "!" >	161	170	170	170	194.161	128.66	
<CENT SIGN>	162	74	74	176	194.162	128.67	##
<POUND SIGN>	163	177	177	177	194.163	128.68	
<CURRENCY SIGN>	164	159	159	159	194.164	128.69	
<YEN SIGN>	165	178	178	178	194.165	128.70	
<BROKEN BAR>	166	106	106	208	194.166	128.71	##
<SECTION SIGN>	167	181	181	181	194.167	128.72	
<DIAERESIS>	168	189	187	121	194.168	128.73	** ##
<COPYRIGHT SIGN>	169	180	180	180	194.169	128.74	
<FEMININE ORDINAL>	170	154	154	154	194.170	128.81	
<LEFT POINTING GUILLEMET>	171	138	138	138	194.171	128.82	
<NOT SIGN>	172	95	176	186	194.172	128.83	** ##
<SOFT HYPHEN>	173	202	202	202	194.173	128.84	
<REGISTERED TRADE MARK>	174	175	175	175	194.174	128.85	
<MACRON>	175	188	188	161	194.175	128.86	##
<DEGREE SIGN>	176	144	144	144	194.176	128.87	
<PLUS-OR-MINUS SIGN>	177	143	143	143	194.177	128.88	
<SUPERSCRIPT TWO>	178	234	234	234	194.178	128.89	
<SUPERSCRIPT THREE>	179	250	250	250	194.179	128.98	
<ACUTE ACCENT>	180	190	190	190	194.180	128.99	
<MICRO SIGN>	181	160	160	160	194.181	128.100	
<PARAGRAPH SIGN>	182	182	182	182	194.182	128.101	
<MIDDLE DOT>	183	179	179	179	194.183	128.102	
<CEDILLA>	184	157	157	157	194.184	128.103	
<SUPERSCRIPT ONE>	185	218	218	218	194.185	128.104	
<MASC. ORDINAL INDICATOR>	186	155	155	155	194.186	128.105	
<RIGHT POINTING GUILLEMET>	187	139	139	139	194.187	128.106	
<FRACTION ONE QUARTER>	188	183	183	183	194.188	128.112	
<FRACTION ONE HALF>	189	184	184	184	194.189	128.113	
<FRACTION THREE QUARTERS>	190	185	185	185	194.190	128.114	
<INVERTED QUESTION MARK>	191	171	171	171	194.191	128.115	
<A WITH GRAVE>	192	100	100	100	195.128	138.65	
<A WITH ACUTE>	193	101	101	101	195.129	138.66	
<A WITH CIRCUMFLEX>	194	98	98	98	195.130	138.67	
<A WITH TILDE>	195	102	102	102	195.131	138.68	

<A WITH DIAERESIS>	196	99	99	99	195.132	138.69	
<A WITH RING ABOVE>	197	103	103	103	195.133	138.70	
<CAPITAL LIGATURE AE>	198	158	158	158	195.134	138.71	
<C WITH CEDILLA>	199	104	104	104	195.135	138.72	
<E WITH GRAVE>	200	116	116	116	195.136	138.73	
<E WITH ACUTE>	201	113	113	113	195.137	138.74	
<E WITH CIRCUMFLEX>	202	114	114	114	195.138	138.81	
<E WITH DIAERESIS>	203	115	115	115	195.139	138.82	
<I WITH GRAVE>	204	120	120	120	195.140	138.83	
<I WITH ACUTE>	205	117	117	117	195.141	138.84	
<I WITH CIRCUMFLEX>	206	118	118	118	195.142	138.85	
<I WITH DIAERESIS>	207	119	119	119	195.143	138.86	
<CAPITAL LETTER ETH>	208	172	172	172	195.144	138.87	
<N WITH TILDE>	209	105	105	105	195.145	138.88	
<O WITH GRAVE>	210	237	237	237	195.146	138.89	
<O WITH ACUTE>	211	238	238	238	195.147	138.98	
<O WITH CIRCUMFLEX>	212	235	235	235	195.148	138.99	
<O WITH TILDE>	213	239	239	239	195.149	138.100	
<O WITH DIAERESIS>	214	236	236	236	195.150	138.101	
<MULTIPLICATION SIGN>	215	191	191	191	195.151	138.102	
<O WITH STROKE>	216	128	128	128	195.152	138.103	
<U WITH GRAVE>	217	253	253	224	195.153	138.104	##
<U WITH ACUTE>	218	254	254	254	195.154	138.105	
<U WITH CIRCUMFLEX>	219	251	251	221	195.155	138.106	##
<U WITH DIAERESIS>	220	252	252	252	195.156	138.112	
<Y WITH ACUTE>	221	173	186	173	195.157	138.113	** ##
<CAPITAL LETTER THORN>	222	174	174	174	195.158	138.114	
<SMALL LETTER SHARP S>	223	89	89	89	195.159	138.115	
<a WITH GRAVE>	224	68	68	68	195.160	139.65	
<a WITH ACUTE>	225	69	69	69	195.161	139.66	
<a WITH CIRCUMFLEX>	226	66	66	66	195.162	139.67	
<a WITH TILDE>	227	70	70	70	195.163	139.68	
<a WITH DIAERESIS>	228	67	67	67	195.164	139.69	
<a WITH RING ABOVE>	229	71	71	71	195.165	139.70	
<SMALL LIGATURE ae>	230	156	156	156	195.166	139.71	
<c WITH CEDILLA>	231	72	72	72	195.167	139.72	
<e WITH GRAVE>	232	84	84	84	195.168	139.73	
<e WITH ACUTE>	233	81	81	81	195.169	139.74	
<e WITH CIRCUMFLEX>	234	82	82	82	195.170	139.81	
<e WITH DIAERESIS>	235	83	83	83	195.171	139.82	
<i WITH GRAVE>	236	88	88	88	195.172	139.83	
<i WITH ACUTE>	237	85	85	85	195.173	139.84	
<i WITH CIRCUMFLEX>	238	86	86	86	195.174	139.85	
<i WITH DIAERESIS>	239	87	87	87	195.175	139.86	
<SMALL LETTER eth>	240	140	140	140	195.176	139.87	
<n WITH TILDE>	241	73	73	73	195.177	139.88	
<o WITH GRAVE>	242	205	205	205	195.178	139.89	

```
<o WITH ACUTE>              243  206  206  206  195.179  139.98
<o WITH CIRCUMFLEX>         244  203  203  203  195.180  139.99
<o WITH TILDE>              245  207  207  207  195.181  139.100
<o WITH DIAERESIS>          246  204  204  204  195.182  139.101
<DIVISION SIGN>             247  225  225  225  195.183  139.102
<o WITH STROKE>             248  112  112  112  195.184  139.103
<u WITH GRAVE>              249  221  221  192  195.185  139.104  ##
<u WITH ACUTE>              250  222  222  222  195.186  139.105
<u WITH CIRCUMFLEX>         251  219  219  219  195.187  139.106
<u WITH DIAERESIS>          252  220  220  220  195.188  139.112
<y WITH ACUTE>              253  141  141  141  195.189  139.113
<SMALL LETTER thorn>        254  142  142  142  195.190  139.114
<y WITH DIAERESIS>          255  223  223  223  195.191  139.115
```

If you would rather see the above table in CCSID 0037 order rather than ASCII + Latin-1 order then run the table through:

recipe 4

```
perl \
   -ne 'if(/.{29}\d{1,3}\s{2,4}\d{1,3}\s{2,4}\d{1,3}\s{2,4}\d{1,3}/)'\
    -e '{push(@l,$_)}' \
    -e 'END{print map{$_->[0]}' \
    -e '          sort{$a->[1] <=> $b->[1]}' \
    -e '          map{[$_,substr($_,34,3)]}@l;}' perlebcdic.pod
```

If you would rather see it in CCSID 1047 order then change the number 34 in the last line to 39, like this:

recipe 5

```
perl \
   -ne 'if(/.{29}\d{1,3}\s{2,4}\d{1,3}\s{2,4}\d{1,3}\s{2,4}\d{1,3}/)'\
   -e '{push(@l,$_)}' \
   -e 'END{print map{$_->[0]}' \
   -e '          sort{$a->[1] <=> $b->[1]}' \
   -e '          map{[$_,substr($_,39,3)]}@l;}' perlebcdic.pod
```

If you would rather see it in POSIX-BC order then change the number 34 in the last line to 44, like this:

recipe 6

```
perl \
   -ne 'if(/.{29}\d{1,3}\s{2,4}\d{1,3}\s{2,4}\d{1,3}\s{2,4}\d{1,3}/)'\
    -e '{push(@l,$_)}' \
    -e 'END{print map{$_->[0]}' \
    -e '          sort{$a->[1] <=> $b->[1]}' \
    -e '          map{[$_,substr($_,44,3)]}@l;}' perlebcdic.pod
```

19.4.1 Table in hex, sorted in 1047 order

Since this document was first written, the convention has become more and more to use hexadecimal notation for code points. To do this with the recipes and to also sort is a multi-step process, so here, for convenience, is the table from above, re-sorted to be in Code Page 1047 order, and using hex notation.

	ISO						
	8859-1			POS-		CCSID	
	CCSID	CCSID	CCSID	IX-		1047	
chr	0819	0037	1047	BC	UTF-8	UTF-EBCDIC	
<NUL>	00	00	00	00	00	00	
<SOH>	01	01	01	01	01	01	
<STX>	02	02	02	02	02	02	
<ETX>	03	03	03	03	03	03	
<ST>	9C	04	04	04	C2.9C	04	
<HT>	09	05	05	05	09	05	
<SSA>	86	06	06	06	C2.86	06	
<DEL>	7F	07	07	07	7F	07	
<EPA>	97	08	08	08	C2.97	08	
<RI>	8D	09	09	09	C2.8D	09	
<SS2>	8E	0A	0A	0A	C2.8E	0A	
<VT>	0B	0B	0B	0B	0B	0B	
<FF>	0C	0C	0C	0C	0C	0C	
<CR>	0D	0D	0D	0D	0D	0D	
<SO>	0E	0E	0E	0E	0E	0E	
<SI>	0F	0F	0F	0F	0F	0F	
<DLE>	10	10	10	10	10	10	
<DC1>	11	11	11	11	11	11	
<DC2>	12	12	12	12	12	12	
<DC3>	13	13	13	13	13	13	
<OSC>	9D	14	14	14	C2.9D	14	
<LF>	0A	25	15	15	0A	15	**
<BS>	08	16	16	16	08	16	
<ESA>	87	17	17	17	C2.87	17	
<CAN>	18	18	18	18	18	18	
<EOM>	19	19	19	19	19	19	
<PU2>	92	1A	1A	1A	C2.92	1A	
<SS3>	8F	1B	1B	1B	C2.8F	1B	
<FS>	1C	1C	1C	1C	1C	1C	
<GS>	1D	1D	1D	1D	1D	1D	
<RS>	1E	1E	1E	1E	1E	1E	
<US>	1F	1F	1F	1F	1F	1F	
<PAD>	80	20	20	20	C2.80	20	
<HOP>	81	21	21	21	C2.81	21	
<BPH>	82	22	22	22	C2.82	22	
<NBH>	83	23	23	23	C2.83	23	
<IND>	84	24	24	24	C2.84	24	
<NEL>	85	15	25	25	C2.85	25	**
<ETB>	17	26	26	26	17	26	

<ESC>	1B	27	27	27	1B	27	
<HTS>	88	28	28	28	C2.88	28	
<HTJ>	89	29	29	29	C2.89	29	
<VTS>	8A	2A	2A	2A	C2.8A	2A	
<PLD>	8B	2B	2B	2B	C2.8B	2B	
<PLU>	8C	2C	2C	2C	C2.8C	2C	
<ENQ>	05	2D	2D	2D	05	2D	
<ACK>	06	2E	2E	2E	06	2E	
<BEL>	07	2F	2F	2F	07	2F	
<DCS>	90	30	30	30	C2.90	30	
<PU1>	91	31	31	31	C2.91	31	
<SYN>	16	32	32	32	16	32	
<STS>	93	33	33	33	C2.93	33	
<CCH>	94	34	34	34	C2.94	34	
<MW>	95	35	35	35	C2.95	35	
<SPA>	96	36	36	36	C2.96	36	
<EOT>	04	37	37	37	04	37	
<SOS>	98	38	38	38	C2.98	38	
<SGC>	99	39	39	39	C2.99	39	
<SCI>	9A	3A	3A	3A	C2.9A	3A	
<CSI>	9B	3B	3B	3B	C2.9B	3B	
<DC4>	14	3C	3C	3C	14	3C	
<NAK>	15	3D	3D	3D	15	3D	
<PM>	9E	3E	3E	3E	C2.9E	3E	
<SUB>	1A	3F	3F	3F	1A	3F	
<SPACE>	20	40	40	40	20	40	
<NON-BREAKING SPACE>	A0	41	41	41	C2.A0	80.41	
<a WITH CIRCUMFLEX>	E2	42	42	42	C3.A2	8B.43	
<a WITH DIAERESIS>	E4	43	43	43	C3.A4	8B.45	
<a WITH GRAVE>	E0	44	44	44	C3.A0	8B.41	
<a WITH ACUTE>	E1	45	45	45	C3.A1	8B.42	
<a WITH TILDE>	E3	46	46	46	C3.A3	8B.44	
<a WITH RING ABOVE>	E5	47	47	47	C3.A5	8B.46	
<c WITH CEDILLA>	E7	48	48	48	C3.A7	8B.48	
<n WITH TILDE>	F1	49	49	49	C3.B1	8B.58	
<CENT SIGN>	A2	4A	4A	B0	C2.A2	80.43	##
.	2E	4B	4B	4B	2E	4B	
<	3C	4C	4C	4C	3C	4C	
(	28	4D	4D	4D	28	4D	
+	2B	4E	4E	4E	2B	4E	
|	7C	4F	4F	4F	7C	4F	
&	26	50	50	50	26	50	
<e WITH ACUTE>	E9	51	51	51	C3.A9	8B.4A	
<e WITH CIRCUMFLEX>	EA	52	52	52	C3.AA	8B.51	
<e WITH DIAERESIS>	EB	53	53	53	C3.AB	8B.52	
<e WITH GRAVE>	E8	54	54	54	C3.A8	8B.49	
<i WITH ACUTE>	ED	55	55	55	C3.AD	8B.54	

<i WITH CIRCUMFLEX>	EE	56	56	56	C3.AE	8B.55	
<i WITH DIAERESIS>	EF	57	57	57	C3.AF	8B.56	
<i WITH GRAVE>	EC	58	58	58	C3.AC	8B.53	
<SMALL LETTER SHARP S>	DF	59	59	59	C3.9F	8A.73	
!	21	5A	5A	5A	21	5A	
$	24	5B	5B	5B	24	5B	
*	2A	5C	5C	5C	2A	5C	
)	29	5D	5D	5D	29	5D	
;	3B	5E	5E	5E	3B	5E	
ˆ	5E	B0	5F	6A	5E	5F	** ##
-	2D	60	60	60	2D	60	
/	2F	61	61	61	2F	61	
<A WITH CIRCUMFLEX>	C2	62	62	62	C3.82	8A.43	
<A WITH DIAERESIS>	C4	63	63	63	C3.84	8A.45	
<A WITH GRAVE>	C0	64	64	64	C3.80	8A.41	
<A WITH ACUTE>	C1	65	65	65	C3.81	8A.42	
<A WITH TILDE>	C3	66	66	66	C3.83	8A.44	
<A WITH RING ABOVE>	C5	67	67	67	C3.85	8A.46	
<C WITH CEDILLA>	C7	68	68	68	C3.87	8A.48	
<N WITH TILDE>	D1	69	69	69	C3.91	8A.58	
<BROKEN BAR>	A6	6A	6A	D0	C2.A6	80.47	##
,	2C	6B	6B	6B	2C	6B	
%	25	6C	6C	6C	25	6C	
_	5F	6D	6D	6D	5F	6D	
>	3E	6E	6E	6E	3E	6E	
?	3F	6F	6F	6F	3F	6F	
<o WITH STROKE>	F8	70	70	70	C3.B8	8B.67	
<E WITH ACUTE>	C9	71	71	71	C3.89	8A.4A	
<E WITH CIRCUMFLEX>	CA	72	72	72	C3.8A	8A.51	
<E WITH DIAERESIS>	CB	73	73	73	C3.8B	8A.52	
<E WITH GRAVE>	C8	74	74	74	C3.88	8A.49	
<I WITH ACUTE>	CD	75	75	75	C3.8D	8A.54	
<I WITH CIRCUMFLEX>	CE	76	76	76	C3.8E	8A.55	
<I WITH DIAERESIS>	CF	77	77	77	C3.8F	8A.56	
<I WITH GRAVE>	CC	78	78	78	C3.8C	8A.53	
‘	60	79	79	4A	60	79	##
:	3A	7A	7A	7A	3A	7A	
#	23	7B	7B	7B	23	7B	
@	40	7C	7C	7C	40	7C	
’	27	7D	7D	7D	27	7D	
=	3D	7E	7E	7E	3D	7E	
"	22	7F	7F	7F	22	7F	
<O WITH STROKE>	D8	80	80	80	C3.98	8A.67	
a	61	81	81	81	61	81	
b	62	82	82	82	62	82	
c	63	83	83	83	63	83	
d	64	84	84	84	64	84	

e	65	85	85	85	65	85	
f	66	86	86	86	66	86	
g	67	87	87	87	67	87	
h	68	88	88	88	68	88	
i	69	89	89	89	69	89	
<LEFT POINTING GUILLEMET>	AB	8A	8A	8A	C2.AB	80.52	
<RIGHT POINTING GUILLEMET>	BB	8B	8B	8B	C2.BB	80.6A	
<SMALL LETTER eth>	F0	8C	8C	8C	C3.B0	8B.57	
<y WITH ACUTE>	FD	8D	8D	8D	C3.BD	8B.71	
<SMALL LETTER thorn>	FE	8E	8E	8E	C3.BE	8B.72	
<PLUS-OR-MINUS SIGN>	B1	8F	8F	8F	C2.B1	80.58	
<DEGREE SIGN>	B0	90	90	90	C2.B0	80.57	
j	6A	91	91	91	6A	91	
k	6B	92	92	92	6B	92	
l	6C	93	93	93	6C	93	
m	6D	94	94	94	6D	94	
n	6E	95	95	95	6E	95	
o	6F	96	96	96	6F	96	
p	70	97	97	97	70	97	
q	71	98	98	98	71	98	
r	72	99	99	99	72	99	
<FEMININE ORDINAL>	AA	9A	9A	9A	C2.AA	80.51	
<MASC. ORDINAL INDICATOR>	BA	9B	9B	9B	C2.BA	80.69	
<SMALL LIGATURE ae>	E6	9C	9C	9C	C3.A6	8B.47	
<CEDILLA>	B8	9D	9D	9D	C2.B8	80.67	
<CAPITAL LIGATURE AE>	C6	9E	9E	9E	C3.86	8A.47	
<CURRENCY SIGN>	A4	9F	9F	9F	C2.A4	80.45	
<MICRO SIGN>	B5	A0	A0	A0	C2.B5	80.64	
~	7E	A1	A1	FF	7E	A1	##
s	73	A2	A2	A2	73	A2	
t	74	A3	A3	A3	74	A3	
u	75	A4	A4	A4	75	A4	
v	76	A5	A5	A5	76	A5	
w	77	A6	A6	A6	77	A6	
x	78	A7	A7	A7	78	A7	
y	79	A8	A8	A8	79	A8	
z	7A	A9	A9	A9	7A	A9	
<INVERTED "!" >	A1	AA	AA	AA	C2.A1	80.42	
<INVERTED QUESTION MARK>	BF	AB	AB	AB	C2.BF	80.73	
<CAPITAL LETTER ETH>	D0	AC	AC	AC	C3.90	8A.57	
[	5B	BA	AD	BB	5B	AD	** ##
<CAPITAL LETTER THORN>	DE	AE	AE	AE	C3.9E	8A.72	
<REGISTERED TRADE MARK>	AE	AF	AF	AF	C2.AE	80.55	
<NOT SIGN>	AC	5F	B0	BA	C2.AC	80.53	** ##
<POUND SIGN>	A3	B1	B1	B1	C2.A3	80.44	
<YEN SIGN>	A5	B2	B2	B2	C2.A5	80.46	
<MIDDLE DOT>	B7	B3	B3	B3	C2.B7	80.66	

<COPYRIGHT SIGN>	A9	B4	B4	B4	C2.A9	80.4A	
<SECTION SIGN>	A7	B5	B5	B5	C2.A7	80.48	
<PARAGRAPH SIGN>	B6	B6	B6	B6	C2.B6	80.65	
<FRACTION ONE QUARTER>	BC	B7	B7	B7	C2.BC	80.70	
<FRACTION ONE HALF>	BD	B8	B8	B8	C2.BD	80.71	
<FRACTION THREE QUARTERS>	BE	B9	B9	B9	C2.BE	80.72	
<Y WITH ACUTE>	DD	AD	BA	AD	C3.9D	8A.71	** ##
<DIAERESIS>	A8	BD	BB	79	C2.A8	80.49	** ##
<MACRON>	AF	BC	BC	A1	C2.AF	80.56	##
]	5D	BB	BD	BD	5D	BD	**
<ACUTE ACCENT>	B4	BE	BE	BE	C2.B4	80.63	
<MULTIPLICATION SIGN>	D7	BF	BF	BF	C3.97	8A.66	
{	7B	C0	C0	FB	7B	C0	##
A	41	C1	C1	C1	41	C1	
B	42	C2	C2	C2	42	C2	
C	43	C3	C3	C3	43	C3	
D	44	C4	C4	C4	44	C4	
E	45	C5	C5	C5	45	C5	
F	46	C6	C6	C6	46	C6	
G	47	C7	C7	C7	47	C7	
H	48	C8	C8	C8	48	C8	
I	49	C9	C9	C9	49	C9	
<SOFT HYPHEN>	AD	CA	CA	CA	C2.AD	80.54	
<o WITH CIRCUMFLEX>	F4	CB	CB	CB	C3.B4	8B.63	
<o WITH DIAERESIS>	F6	CC	CC	CC	C3.B6	8B.65	
<o WITH GRAVE>	F2	CD	CD	CD	C3.B2	8B.59	
<o WITH ACUTE>	F3	CE	CE	CE	C3.B3	8B.62	
<o WITH TILDE>	F5	CF	CF	CF	C3.B5	8B.64	
}	7D	D0	D0	FD	7D	D0	##
J	4A	D1	D1	D1	4A	D1	
K	4B	D2	D2	D2	4B	D2	
L	4C	D3	D3	D3	4C	D3	
M	4D	D4	D4	D4	4D	D4	
N	4E	D5	D5	D5	4E	D5	
O	4F	D6	D6	D6	4F	D6	
P	50	D7	D7	D7	50	D7	
Q	51	D8	D8	D8	51	D8	
R	52	D9	D9	D9	52	D9	
<SUPERSCRIPT ONE>	B9	DA	DA	DA	C2.B9	80.68	
<u WITH CIRCUMFLEX>	FB	DB	DB	DB	C3.BB	8B.6A	
<u WITH DIAERESIS>	FC	DC	DC	DC	C3.BC	8B.70	
<u WITH GRAVE>	F9	DD	DD	C0	C3.B9	8B.68	##
<u WITH ACUTE>	FA	DE	DE	DE	C3.BA	8B.69	
<y WITH DIAERESIS>	FF	DF	DF	DF	C3.BF	8B.73	
\	5C	E0	E0	BC	5C	E0	##
<DIVISION SIGN>	F7	E1	E1	E1	C3.B7	8B.66	
S	53	E2	E2	E2	53	E2	

T	54	E3	E3	E3	54	E3	
U	55	E4	E4	E4	55	E4	
V	56	E5	E5	E5	56	E5	
W	57	E6	E6	E6	57	E6	
X	58	E7	E7	E7	58	E7	
Y	59	E8	E8	E8	59	E8	
Z	5A	E9	E9	E9	5A	E9	
<SUPERSCRIPT TWO>	B2	EA	EA	EA	C2.B2	80.59	
<O WITH CIRCUMFLEX>	D4	EB	EB	EB	C3.94	8A.63	
<O WITH DIAERESIS>	D6	EC	EC	EC	C3.96	8A.65	
<O WITH GRAVE>	D2	ED	ED	ED	C3.92	8A.59	
<O WITH ACUTE>	D3	EE	EE	EE	C3.93	8A.62	
<O WITH TILDE>	D5	EF	EF	EF	C3.95	8A.64	
0	30	F0	F0	F0	30	F0	
1	31	F1	F1	F1	31	F1	
2	32	F2	F2	F2	32	F2	
3	33	F3	F3	F3	33	F3	
4	34	F4	F4	F4	34	F4	
5	35	F5	F5	F5	35	F5	
6	36	F6	F6	F6	36	F6	
7	37	F7	F7	F7	37	F7	
8	38	F8	F8	F8	38	F8	
9	39	F9	F9	F9	39	F9	
<SUPERSCRIPT THREE>	B3	FA	FA	FA	C2.B3	80.62	
<U WITH CIRCUMFLEX>	DB	FB	FB	DD	C3.9B	8A.6A	##
<U WITH DIAERESIS>	DC	FC	FC	FC	C3.9C	8A.70	
<U WITH GRAVE>	D9	FD	FD	E0	C3.99	8A.68	##
<U WITH ACUTE>	DA	FE	FE	FE	C3.9A	8A.69	
<APC>	9F	FF	FF	5F	C2.9F	FF	##

19.5 IDENTIFYING CHARACTER CODE SETS

It is possible to determine which character set you are operating under. But first you need to be really really sure you need to do this. Your code will be simpler and probably just as portable if you don't have to test the character set and do different things, depending. There are actually only very few circumstances where it's not easy to write straight-line code portable to all character sets. See Section 83.2.4 [perluniintro Unicode and EBCDIC], page 1355 for how to portably specify characters.

But there are some cases where you may want to know which character set you are running under. One possible example is doing Section 19.11 [sorting], page 293 in inner loops where performance is critical.

To determine if you are running under ASCII or EBCDIC, you can use the return value of `ord()` or `chr()` to test one or more character values. For example:

```
    $is_ascii  = "A" eq chr(65);
    $is_ebcdic = "A" eq chr(193);
    $is_ascii  = ord("A") == 65;
    $is_ebcdic = ord("A") == 193;
```

There's even less need to distinguish between EBCDIC code pages, but to do so try looking at one or more of the characters that differ between them.

```
$is_ascii           = ord('[') == 91;
$is_ebcdic_37       = ord('[') == 186;
$is_ebcdic_1047     = ord('[') == 173;
$is_ebcdic_POSIX_BC = ord('[') == 187;
```

However, it would be unwise to write tests such as:

```
$is_ascii = "\r" ne chr(13);  #  WRONG
$is_ascii = "\n" ne chr(10);  #  ILL ADVISED
```

Obviously the first of these will fail to distinguish most ASCII platforms from either a CCSID 0037, a 1047, or a POSIX-BC EBCDIC platform since `"\r" eq chr(13)` under all of those coded character sets. But note too that because `"\n"` is `chr(13)` and `"\r"` is `chr(10)` on old Macintosh (which is an ASCII platform) the second `$is_ascii` test will lead to trouble there.

To determine whether or not perl was built under an EBCDIC code page you can use the Config module like so:

```
use Config;
$is_ebcdic = $Config{'ebcdic'} eq 'define';
```

19.6 CONVERSIONS

19.6.1 utf8::unicode_to_native() and utf8::native_to_unicode()

These functions take an input numeric code point in one encoding and return what its equivalent value is in the other.

See `utf8`.

19.6.2 tr///

In order to convert a string of characters from one character set to another a simple list of numbers, such as in the right columns in the above table, along with Perl's `tr///` operator is all that is needed. The data in the table are in ASCII/Latin1 order, hence the EBCDIC columns provide easy-to-use ASCII/Latin1 to EBCDIC operations that are also easily reversed.

For example, to convert ASCII/Latin1 to code page 037 take the output of the second numbers column from the output of recipe 2 (modified to add `"\"` characters), and use it in `tr///` like so:

```
$cp_037 =
'\x00\x01\x02\x03\x37\x2D\x2E\x2F\x16\x05\x25\x0B\x0C\x0D\x0E\x0F' .
'\x10\x11\x12\x13\x3C\x3D\x32\x26\x18\x19\x3F\x27\x1C\x1D\x1E\x1F' .
'\x40\x5A\x7F\x7B\x5B\x6C\x50\x7D\x4D\x5D\x5C\x4E\x6B\x60\x4B\x61' .
'\xF0\xF1\xF2\xF3\xF4\xF5\xF6\xF7\xF8\xF9\x7A\x5E\x4C\x7E\x6E\x6F' .
'\x7C\xC1\xC2\xC3\xC4\xC5\xC6\xC7\xC8\xC9\xD1\xD2\xD3\xD4\xD5\xD6' .
'\xD7\xD8\xD9\xE2\xE3\xE4\xE5\xE6\xE7\xE8\xE9\xBA\xE0\xBB\xB0\x6D' .
'\x79\x81\x82\x83\x84\x85\x86\x87\x88\x89\x91\x92\x93\x94\x95\x96' .
'\x97\x98\x99\xA2\xA3\xA4\xA5\xA6\xA7\xA8\xA9\xC0\x4F\xD0\xA1\x07' .
```

```
'\x20\x21\x22\x23\x24\x15\x06\x17\x28\x29\x2A\x2B\x2C\x09\x0A\x1B' .
'\x30\x31\x1A\x33\x34\x35\x36\x08\x38\x39\x3A\x3B\x04\x14\x3E\xFF' .
'\x41\xAA\x4A\xB1\x9F\xB2\x6A\xB5\xBD\xB4\x9A\x8A\x5F\xCA\xAF\xBC' .
'\x90\x8F\xEA\xFA\xBE\xA0\xB6\xB3\x9D\xDA\x9B\x8B\xB7\xB8\xB9\xAB' .
'\x64\x65\x62\x66\x63\x67\x9E\x68\x74\x71\x72\x73\x78\x75\x76\x77' .
'\xAC\x69\xED\xEE\xEB\xEF\xEC\xBF\x80\xFD\xFE\xFB\xFC\xAD\xAE\x59' .
'\x44\x45\x42\x46\x43\x47\x9C\x48\x54\x51\x52\x53\x58\x55\x56\x57' .
'\x8C\x49\xCD\xCE\xCB\xCF\xCC\xE1\x70\xDD\xDE\xDB\xDC\x8D\x8E\xDF';

my $ebcdic_string = $ascii_string;
eval '$ebcdic_string =~ tr/\000-\377/' . $cp_037 . '/';
```

To convert from EBCDIC 037 to ASCII just reverse the order of the tr/// arguments like so:

```
my $ascii_string = $ebcdic_string;
eval '$ascii_string =~ tr/' . $cp_037 . '/\000-\377/';
```

Similarly one could take the output of the third numbers column from recipe 2 to obtain a `$cp_1047` table. The fourth numbers column of the output from recipe 2 could provide a `$cp_posix_bc` table suitable for transcoding as well.

If you wanted to see the inverse tables, you would first have to sort on the desired numbers column as in recipes 4, 5 or 6, then take the output of the first numbers column.

19.6.3 iconv

XPG operability often implies the presence of an *iconv* utility available from the shell or from the C library. Consult your system's documentation for information on iconv.

On OS/390 or z/OS see the iconv(1) manpage. One way to invoke the `iconv` shell utility from within perl would be to:

```
# OS/390 or z/OS example
$ascii_data = `echo '$ebcdic_data'| iconv -f IBM-1047 -t ISO8859-1`
```

or the inverse map:

```
# OS/390 or z/OS example
$ebcdic_data = `echo '$ascii_data'| iconv -f ISO8859-1 -t IBM-1047`
```

For other Perl-based conversion options see the `Convert::*` modules on CPAN.

19.6.4 C RTL

The OS/390 and z/OS C run-time libraries provide `_atoe()` and `_etoa()` functions.

19.7 OPERATOR DIFFERENCES

The `..` range operator treats certain character ranges with care on EBCDIC platforms. For example the following array will have twenty six elements on either an EBCDIC platform or an ASCII platform:

```
@alphabet = ('A'..'Z');   #  $#alphabet == 25
```

The bitwise operators such as & ^ | may return different results when operating on string or character data in a Perl program running on an EBCDIC platform than when run on an ASCII platform. Here is an example adapted from the one in Section 48.1 [perlop NAME], page 798:

```
# EBCDIC-based examples
print "j p \n" ^ " a h";                      # prints "JAPH\n"
print "JA" | "  ph\n";                        # prints "japh\n"
print "JAPH\nJunk" & "\277\277\277\277\277";  # prints "japh\n";
print ’p N$’ ^ " E<H\n";                      # prints "Perl\n";
```

An interesting property of the 32 C0 control characters in the ASCII table is that they can "literally" be constructed as control characters in Perl, e.g. `(chr(0)` eq `\c@)>` `(chr(1)` eq `\cA)>`, and so on. Perl on EBCDIC platforms has been ported to take `\c@` to `chr(0)` and `\cA` to `chr(1)`, etc. as well, but the characters that result depend on which code page you are using. The table below uses the standard acronyms for the controls. The POSIX-BC and 1047 sets are identical throughout this range and differ from the 0037 set at only one spot (21 decimal). Note that the line terminator character may be generated by `\cJ` on ASCII platforms but by `\cU` on 1047 or POSIX-BC platforms and cannot be generated as a `"\c.letter."` control character on 0037 platforms. Note also that `\c\` cannot be the final element in a string or regex, as it will absorb the terminator. But `\c\`*X* is a `FILE SEPARATOR` concatenated with *X* for all *X*. The outlier `\c?` on ASCII, which yields a non-C0 control `DEL`, yields the outlier control `APC` on EBCDIC, the one that isn't in the block of contiguous controls. Note that a subtlety of this is that `\c?` on ASCII platforms is an ASCII character, while it isn't equivalent to any ASCII character in EBCDIC platforms.

chr	ord	8859-1	0037	1047 && POSIX-BC	
\c@	0	<NUL>	<NUL>	<NUL>	
\cA	1	<SOH>	<SOH>	<SOH>	
\cB	2	<STX>	<STX>	<STX>	
\cC	3	<ETX>	<ETX>	<ETX>	
\cD	4	<EOT>	<ST>	<ST>	
\cE	5	<ENQ>	<HT>	<HT>	
\cF	6	<ACK>	<SSA>	<SSA>	
\cG	7	<BEL>	<DEL>	<DEL>	
\cH	8	<BS>	<EPA>	<EPA>	
\cI	9	<HT>	<RI>	<RI>	
\cJ	10	<LF>	<SS2>	<SS2>	
\cK	11	<VT>	<VT>	<VT>	
\cL	12	<FF>	<FF>	<FF>	
\cM	13	<CR>	<CR>	<CR>	
\cN	14	<SO>	<SO>	<SO>	
\cO	15	<SI>	<SI>	<SI>	
\cP	16	<DLE>	<DLE>	<DLE>	
\cQ	17	<DC1>	<DC1>	<DC1>	
\cR	18	<DC2>	<DC2>	<DC2>	
\cS	19	<DC3>	<DC3>	<DC3>	
\cT	20	<DC4>	<OSC>	<OSC>	
\cU	21	<NAK>	<NEL>	<LF>	**
\cV	22	<SYN>	<BS>	<BS>	

\cW	23	<ETB>	<ESA>	<ESA>
\cX	24	<CAN>	<CAN>	<CAN>
\cY	25	<EOM>	<EOM>	<EOM>
\cZ	26	<SUB>	<PU2>	<PU2>
\c[	27	<ESC>	<SS3>	<SS3>
\c\X	28	<FS>X	<FS>X	<FS>X
\c]	29	<GS>	<GS>	<GS>
\c^	30	<RS>	<RS>	<RS>
\c_	31	<US>	<US>	<US>
\c?	*	<DEL>	<APC>	<APC>

* Note: \c? maps to ordinal 127 (DEL) on ASCII platforms, but since ordinal 127 is a not a control character on EBCDIC machines, \c? instead maps on them to APC, which is 255 in 0037 and 1047, and 95 in POSIX-BC.

19.8 FUNCTION DIFFERENCES

chr()

chr() must be given an EBCDIC code number argument to yield a desired character return value on an EBCDIC platform. For example:

```
$CAPITAL_LETTER_A = chr(193);
```

The largest code point that is representable in UTF-EBCDIC is U+7FFF_FFFF. If you do chr() on a larger value, a runtime error (similar to division by 0) will happen.

ord()

ord() will return EBCDIC code number values on an EBCDIC platform. For example:

```
$the_number_193 = ord("A");
```

pack()

The "c" and "C" templates for pack() are dependent upon character set encoding. Examples of usage on EBCDIC include:

```
$foo = pack("CCCC",193,194,195,196);
# $foo eq "ABCD"
$foo = pack("C4",193,194,195,196);
# same thing

$foo = pack("ccxxcc",193,194,195,196);
# $foo eq "AB\0\0CD"
```

The "U" template has been ported to mean "Unicode" on all platforms so that

```
pack("U", 65) eq ’A’
```

is true on all platforms. If you want native code points for the low 256, use the "W" template. This means that the equivalences

```
pack("W", ord($character)) eq $character
unpack("W", $character) == ord $character
```

will hold.

The largest code point that is representable in UTF-EBCDIC is U+7FFF_FFFF. If you try to pack a larger value into a character, a runtime error (similar to division by 0) will happen.

`print()`

One must be careful with scalars and strings that are passed to print that contain ASCII encodings. One common place for this to occur is in the output of the MIME type header for CGI script writing. For example, many Perl programming guides recommend something similar to:

```
print "Content-type:\ttext/html\015\012\015\012";
# this may be wrong on EBCDIC
```

You can instead write

```
print "Content-type:\ttext/html\r\n\r\n"; # OK for DGW et al
```

and have it work portably.

That is because the translation from EBCDIC to ASCII is done by the web server in this case. Consult your web server's documentation for further details.

`printf()`

The formats that can convert characters to numbers and vice versa will be different from their ASCII counterparts when executed on an EBCDIC platform. Examples include:

```
printf("%c%c%c",193,194,195);  # prints ABC
```

`sort()`

EBCDIC sort results may differ from ASCII sort results especially for mixed case strings. This is discussed in more detail Section 19.11 [below], page 293.

`sprintf()`

See the discussion of `[printf()], page 289` above. An example of the use of sprintf would be:

```
$CAPITAL_LETTER_A = sprintf("%c",193);
```

`unpack()`

See the discussion of `[pack()], page 288` above.

Note that it is possible to write portable code for these by specifying things in Unicode numbers, and using a conversion function:

```
printf("%c",utf8::unicode_to_native(65));  # prints A on all
                                           # platforms
print utf8::native_to_unicode(ord("A"));   # Likewise, prints 65
```

See Section 83.2.4 [perluniintro Unicode and EBCDIC], page 1355 and Section 19.6 [CONVERSIONS], page 285 for other options.

19.9 REGULAR EXPRESSION DIFFERENCES

You can write your regular expressions just like someone on an ASCII platform would do. But keep in mind that using octal or hex notation to specify a particular code point will give you the character that the EBCDIC code page natively maps to it. (This is also true of all double-quoted strings.) If you want to write portably, just use the `\N{U+...}` notation everywhere where you would have used `\x{...}`, and don't use octal notation at all.

Starting in Perl v5.22, this applies to ranges in bracketed character classes. If you say, for example, `qr/[\N{U+20}-\N{U+7F}]/`, it means the characters `\N{U+20}`, `\N{U+21}`, ..., `\N{U+7F}`. This range is all the printable characters that the ASCII character set contains.

Prior to v5.22, you couldn't specify any ranges portably, except (starting in Perl v5.5.3) all subsets of the `[A-Z]` and `[a-z]` ranges are specially coded to not pick up gap characters. For example, characters such as "ô" (`o WITH CIRCUMFLEX`) that lie between "I" and "J" would not be matched by the regular expression range `/[H-K]/`. But if either of the range end points is explicitly numeric (and neither is specified by `\N{U+...}`), the gap characters are matched:

```
/[\x89-\x91]/
```

will match `\x8e`, even though `\x89` is "i" and `\x91` is "j", and `\x8e` is a gap character, from the alphabetic viewpoint.

Another construct to be wary of is the inappropriate use of hex (unless you use `\N{U+...}`) or octal constants in regular expressions. Consider the following set of subs:

```
sub is_c0 {
    my $char = substr(shift,0,1);
    $char =~ /[\000-\037]/;
}

sub is_print_ascii {
    my $char = substr(shift,0,1);
    $char =~ /[\040-\176]/;
}

sub is_delete {
    my $char = substr(shift,0,1);
    $char eq "\177";
}

sub is_c1 {
    my $char = substr(shift,0,1);
    $char =~ /[\200-\237]/;
}

sub is_latin_1 {    # But not ASCII; not C1
    my $char = substr(shift,0,1);
    $char =~ /[\240-\377]/;
}
```

These are valid only on ASCII platforms. Starting in Perl v5.22, simply changing the octal constants to equivalent \N{U+...} values makes them portable:

```
sub is_c0 {
    my $char = substr(shift,0,1);
    $char =~ /[\N{U+00}-\N{U+1F}]/;
}

sub is_print_ascii {
    my $char = substr(shift,0,1);
    $char =~ /[\N{U+20}-\N{U+7E}]/;
}

sub is_delete {
    my $char = substr(shift,0,1);
    $char eq "\N{U+7F}";
}

sub is_c1 {
    my $char = substr(shift,0,1);
    $char =~ /[\N{U+80}-\N{U+9F}]/;
}

sub is_latin_1 {    # But not ASCII; not C1
    my $char = substr(shift,0,1);
    $char =~ /[\N{U+A0}-\N{U+FF}]/;
}
```

And here are some alternative portable ways to write them:

```
sub Is_c0 {
    my $char = substr(shift,0,1);
    return $char =~ /[[:cntrl:]]/a && ! Is_delete($char);

    # Alternatively:
    # return $char =~ /[[:cntrl:]]/
    #        && $char =~ /[[:ascii:]]/
    #        && ! Is_delete($char);
}

sub Is_print_ascii {
    my $char = substr(shift,0,1);

    return $char =~ /[[:print:]]/a;

    # Alternatively:
    # return $char =~ /[[:print:]]/ && $char =~ /[[:ascii:]]/;

    # Or
```

```
        # return $char
        #      =~ /[ !"\#\$%&’()*+,\-.\/0-9:;<=>?\@A-Z[\\\]^_‘a-z{|}~]/;
    }

    sub Is_delete {
        my $char = substr(shift,0,1);
        return utf8::native_to_unicode(ord $char) == 0x7F;
    }

    sub Is_c1 {
        use feature ’unicode_strings’;
        my $char = substr(shift,0,1);
        return $char =~ /[[:cntrl:]]/ && $char !~ /[[:ascii:]]/;
    }

    sub Is_latin_1 {    # But not ASCII; not C1
        use feature ’unicode_strings’;
        my $char = substr(shift,0,1);
        return ord($char) < 256
               && $char !~ /[[:ascii:]]/
               && $char !~ /[[:cntrl:]]/;
    }
```

Another way to write `Is_latin_1()` would be to use the characters in the range explicitly:

```
    sub Is_latin_1 {
        my $char = substr(shift,0,1);
        $char =~ /[ ¡¢£¤¥¦§¨©ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏ]
                  [ÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþÿ]/x;
    }
```

Although that form may run into trouble in network transit (due to the presence of 8 bit characters) or on non ISO-Latin character sets. But it does allow `Is_c1` to be rewritten so it works on Perls that don’t have `’unicode_strings’` (earlier than v5.14):

```
    sub Is_latin_1 {    # But not ASCII; not C1
        my $char = substr(shift,0,1);
        return ord($char) < 256
               && $char !~ /[[:ascii:]]/
               && ! Is_latin1($char);
    }
```

19.10 SOCKETS

Most socket programming assumes ASCII character encodings in network byte order. Exceptions can include CGI script writing under a host web server where the server may take care of translation for you. Most host web servers convert EBCDIC data to ISO-8859-1 or Unicode on output.

19.11 SORTING

One big difference between ASCII-based character sets and EBCDIC ones are the relative positions of the characters when sorted in native order. Of most concern are the upper- and lowercase letters, the digits, and the underscore ("_"). On ASCII platforms the native sort order has the digits come before the uppercase letters which come before the underscore which comes before the lowercase letters. On EBCDIC, the underscore comes first, then the lowercase letters, then the uppercase ones, and the digits last. If sorted on an ASCII-based platform, the two-letter abbreviation for a physician comes before the two letter abbreviation for drive; that is:

```
@sorted = sort(qw(Dr. dr.));  # @sorted holds ('Dr.','dr.') on ASCII,
                                 # but ('dr.','Dr.') on EBCDIC
```

The property of lowercase before uppercase letters in EBCDIC is even carried to the Latin 1 EBCDIC pages such as 0037 and 1047. An example would be that "Ë" (E WITH DIAERESIS, 203) comes before "ë" (e WITH DIAERESIS, 235) on an ASCII platform, but the latter (83) comes before the former (115) on an EBCDIC platform. (Astute readers will note that the uppercase version of "ß" SMALL LETTER SHARP S is simply "SS" and that the upper case versions of "ÿ" (small y WITH DIAERESIS) and "μ" (MICRO SIGN) are not in the 0..255 range but are in Unicode, in a Unicode enabled Perl).

The sort order will cause differences between results obtained on ASCII platforms versus EBCDIC platforms. What follows are some suggestions on how to deal with these differences.

19.11.1 Ignore ASCII vs. EBCDIC sort differences.

This is the least computationally expensive strategy. It may require some user education.

19.11.2 Use a sort helper function

This is completely general, but the most computationally expensive strategy. Choose one or the other character set and transform to that for every sort comparision. Here's a complete example that transforms to ASCII sort order:

```
sub native_to_uni($) {
   my $string = shift;

   # Saves time on an ASCII platform
   return $string if ord 'A' ==  65;

   my $output = "";
   for my $i (0 .. length($string) - 1) {
       $output
           .= chr(utf8::native_to_unicode(ord(substr($string, $i, 1))));
   }

   # Preserve utf8ness of input onto the output, even if it didn't need
   # to be utf8
   utf8::upgrade($output) if utf8::is_utf8($string);
   return $output;
}

sub ascii_order {   # Sort helper
   return native_to_uni($a) cmp native_to_uni($b);
}

sort ascii_order @list;
```

19.11.3 MONO CASE then sort data (for non-digits, non-underscore)

If you don't care about where digits and underscore sort to, you can do something like this

```
sub case_insensitive_order {   # Sort helper
   return lc($a) cmp lc($b)
}

sort case_insensitive_order @list;
```

If performance is an issue, and you don't care if the output is in the same case as the input, Use `tr///` to transform to the case most employed within the data. If the data are primarily UPPERCASE non-Latin1, then apply `tr/[a-z]/[A-Z]/`, and then `sort()`. If the data are primarily lowercase non Latin1 then apply `tr/[A-Z]/[a-z]/` before sorting. If the data are primarily UPPERCASE and include Latin-1 characters then apply:

```
tr/[a-z]/[A-Z]/;
tr/[àáâãäåæçèéêëìíîïðñòóôõöøùúûüýþ]/[ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞ/;
s/ß/SS/g;
```

then `sort()`. If you have a choice, it's better to lowercase things to avoid the problems of the two Latin-1 characters whose uppercase is outside Latin-1: `"ÿ"` (small `y WITH DIAERESIS`) and `"μ"` (`MICRO SIGN`). If you do need to upppercase, you can; with a Unicode-enabled Perl, do:

```
tr/ÿ/\x{178}/;
tr/μ/\x{39C}/;
```

19.11.4 Perform sorting on one type of platform only.

This strategy can employ a network connection. As such it would be computationally expensive.

19.12 TRANSFORMATION FORMATS

There are a variety of ways of transforming data with an intra character set mapping that serve a variety of purposes. Sorting was discussed in the previous section and a few of the other more popular mapping techniques are discussed next.

19.12.1 URL decoding and encoding

Note that some URLs have hexadecimal ASCII code points in them in an attempt to overcome character or protocol limitation issues. For example the tilde character is not on every keyboard hence a URL of the form:

```
http://www.pvhp.com/~pvhp/
```

may also be expressed as either of:

```
http://www.pvhp.com/%7Epvhp/

http://www.pvhp.com/%7epvhp/
```

where 7E is the hexadecimal ASCII code point for "~". Here is an example of decoding such a URL in any EBCDIC code page:

```
$url = ’http://www.pvhp.com/%7Epvhp/’;
$url =~ s/%([0-9a-fA-F]{2})/
          pack("c",utf8::unicode_to_native(hex($1)))/xge;
```

Conversely, here is a partial solution for the task of encoding such a URL in any EBCDIC code page:

```
$url = ’http://www.pvhp.com/~pvhp/’;
# The following regular expression does not address the
# mappings for: (’.’ => ’%2E’, ’/’ => ’%2F’, ’:’ => ’%3A’)
$url =~ s/([\t "#%&\(\),;<=>\?\@\[\\\]^‘{|}~])/
           sprintf("%%%02X",utf8::native_to_unicode(ord($1)))/xge;
```

where a more complete solution would split the URL into components and apply a full s/// substitution only to the appropriate parts.

19.12.2 uu encoding and decoding

The u template to pack() or unpack() will render EBCDIC data in EBCDIC characters equivalent to their ASCII counterparts. For example, the following will print "Yes indeed\n" on either an ASCII or EBCDIC computer:

```
$all_byte_chrs = ’’;
for (0..255) { $all_byte_chrs .= chr($_); }
$uuencode_byte_chrs = pack(’u’, $all_byte_chrs);
($uu = <<’ENDOFHEREDOC’) =~ s/^\s*//gm;
M‘‘$"‘P0%!@<("0H+#‘T.#Q‘1$A,4%187&!D:&QP=’A\@(2(C)"4F)R@I*BLL
M+2XO,#$R,S0U-C<X.3H[/#T^/T!!0D-$149’2$E*2TQ-3D]045)35%565UA9
M6EM<75Y?8&%B8V1E9F=H:6IK;&UN;W!Q<G-T=79W>’EZ>WQ]?G^‘@8*#A(6&
MAXB)BHN,C8Z/D)&2DY25EI>8F9J;G)V>GZ"AHJ.DI::GJ*FJJZRMKJ^PL;*S
MM+6VM[BYNKN\O;Z_P,’"P\3%QL?(R<K+S,W.S]#1TM/4U=;7V-G:V]S=WM_@
?X>+CY.7FY^CIZNOL[>[O\/’R\_3U]O?X^?K[_/W^_P‘‘
ENDOFHEREDOC
if ($uuencode_byte_chrs eq $uu) {
    print "Yes ";
}
$uudecode_byte_chrs = unpack(’u’, $uuencode_byte_chrs);
if ($uudecode_byte_chrs eq $all_byte_chrs) {
    print "indeed\n";
}
```

Here is a very spartan uudecoder that will work on EBCDIC:

```
#!/usr/local/bin/perl
```

```
$_ = <> until ($mode,$file) = /^begin\s*(\d*)\s*(\S*)/;
open(OUT, "> $file") if $file ne "";
while(<>) {
    last if /^end/;
    next if /[a-z]/;
    next unless int((((utf8::native_to_unicode(ord()) - 32 ) & 077)
                                                               + 2) / 3)
                == int(length() / 4);
    print OUT unpack("u", $_);
}
close(OUT);
chmod oct($mode), $file;
```

19.12.3 Quoted-Printable encoding and decoding

On ASCII-encoded platforms it is possible to strip characters outside of the printable set using:

```
# This QP encoder works on ASCII only
$qp_string =~ s/([=\x00-\x1F\x80-\xFF])/
                sprintf("=%02X",ord($1))/xge;
```

Starting in Perl v5.22, this is trivially changeable to work portably on both ASCII and EBCDIC platforms.

```
# This QP encoder works on both ASCII and EBCDIC
$qp_string =~ s/([=\N{U+00}-\N{U+1F}\N{U+80}-\N{U+FF}])/
                sprintf("=%02X",ord($1))/xge;
```

For earlier Perls, a QP encoder that works on both ASCII and EBCDIC platforms would look somewhat like the following:

```
$delete = utf8::unicode_to_native(ord("\x7F"));
$qp_string =~
  s/([^[:print:]$delete])/
     sprintf("=%02X",utf8::native_to_unicode(ord($1)))/xage;
```

(although in production code the substitutions might be done in the EBCDIC branch with the function call and separately in the ASCII branch without the expense of the identity map; in Perl v5.22, the identity map is optimized out so there is no expense, but the alternative above is simpler and is also available in v5.22).

Such QP strings can be decoded with:

```
# This QP decoder is limited to ASCII only
$string =~ s/=([[:xdigit:][[:xdigit:])/chr hex $1/ge;
$string =~ s/=[\n\r]+$//;
```

Whereas a QP decoder that works on both ASCII and EBCDIC platforms would look somewhat like the following:

```
$string =~ s/=([[:xdigit:][:xdigit:]])/
                                chr utf8::native_to_unicode(hex $1)/xge;
$string =~ s/=[\n\r]+$//;
```

19.12.4 Caesarean ciphers

The practice of shifting an alphabet one or more characters for encipherment dates back thousands of years and was explicitly detailed by Gaius Julius Caesar in his **Gallic Wars** text. A single alphabet shift is sometimes referred to as a rotation and the shift amount is given as a number $n after the string 'rot' or "rot$n". Rot0 and rot26 would designate identity maps on the 26-letter English version of the Latin alphabet. Rot13 has the interesting property that alternate subsequent invocations are identity maps (thus rot13 is its own non-trivial inverse in the group of 26 alphabet rotations). Hence the following is a rot13 encoder and decoder that will work on ASCII and EBCDIC platforms:

```
#!/usr/local/bin/perl

while(<>){
    tr/n-za-mN-ZA-M/a-zA-Z/;
    print;
}
```

In one-liner form:

```
perl -ne 'tr/n-za-mN-ZA-M/a-zA-Z/;print'
```

19.13 Hashing order and checksums

Perl deliberately randomizes hash order for security purposes on both ASCII and EBCDIC platforms.

EBCDIC checksums will differ for the same file translated into ASCII and vice versa.

19.14 I18N AND L10N

Internationalization (I18N) and localization (L10N) are supported at least in principle even on EBCDIC platforms. The details are system-dependent and discussed under the Section 19.16 [OS ISSUES], page 297 section below.

19.15 MULTI-OCTET CHARACTER SETS

Perl works with UTF-EBCDIC, a multi-byte encoding. In Perls earlier than v5.22, there may be various bugs in this regard.

Legacy multi byte EBCDIC code pages XXX.

19.16 OS ISSUES

There may be a few system-dependent issues of concern to EBCDIC Perl programmers.

19.16.1 OS/400

PASE

The PASE environment is a runtime environment for OS/400 that can run executables built for PowerPC AIX in OS/400; see `perlos400`. PASE is ASCII-based, not EBCDIC-based as the ILE.

IFS access

XXX.

19.16.2 OS/390, z/OS

Perl runs under Unix Systems Services or USS.

`sigaction`

`SA_SIGINFO` can have segmentation faults.

`chcp`

chcp is supported as a shell utility for displaying and changing one's code page. See also chcp(1).

dataset access

For sequential data set access try:

```
my @ds_records = ‘cat //DSNAME‘;
```

or:

```
my @ds_records = ‘cat //’HLQ.DSNAME’‘;
```

See also the OS390::Stdio module on CPAN.

`iconv`

iconv is supported as both a shell utility and a C RTL routine. See also the iconv(1) and iconv(3) manual pages.

locales

Locales are supported. There may be glitches when a locale is another EBCDIC code page which has some of the Section 19.3.4.1 [code-page variant characters], page 268 in other positions.

There aren't currently any real UTF-8 locales, even though some locale names contain the string `"UTF-8"`.

See Section 38.1 [perllocale NAME], page 701 for information on locales. The L10N files are in `/usr/nls/locale`. `$Config{d_setlocale}` is `’define’` on OS/390 or z/OS.

19.16.3 POSIX-BC?

XXX.

19.17 BUGS

- The `cmp` (and hence `sort`) operators do not necessarily give the correct results when both operands are UTF-EBCDIC encoded strings and there is a mixture of ASCII and/or control characters, along with other characters.
- Ranges containing `\N{...}` in the `tr///` (and `y///`) transliteration operators are treated differently than the equivalent ranges in regular expression pattersn. They should, but don't, cause the values in the ranges to all be treated as Unicode code points, and not native ones. (Section 58.2.7 [perlre Version 8 Regular Expressions], page 1024 gives details as to how it should work.)
- Not all shells will allow multiple `-e` string arguments to perl to be concatenated together properly as recipes in this document 0, 2, 4, 5, and 6 might seem to imply.
- There are some bugs in the `pack`/`unpack` `"U0"` template

- There are a significant number of test failures in the CPAN modules shipped with Perl v5.22. These are only in modules not primarily maintained by Perl 5 porters. Some of these are failures in the tests only: they don't realize that it is proper to get different results on EBCDIC platforms. And some of the failures are real bugs. If you compile and do a `make test` on Perl, all tests on the `/cpan` directory are skipped.

  In particular, the extensions `Unicode-Collate` and `Unicode-Normalize` are not supported under EBCDIC; likewise for the (now deprecated) `encoding` pragma.

  `Encode` partially works.
- In earlier versions, when byte and character data were concatenated, the new string was sometimes created by decoding the byte strings as *ISO 8859-1 (Latin-1)*, even if the old Unicode string used EBCDIC.

19.18 SEE ALSO

Section 38.1 [perllocale NAME], page 701, Section 25.1 [perlfunc NAME], page 351, Section 81.1 [perlunicode NAME], page 1317, `utf8`.

19.19 REFERENCES

`http://anubis.dkuug.dk/i18n/charmaps`

`http://www.unicode.org/`

`http://www.unicode.org/unicode/reports/tr16/`

`http://www.wps.com/projects/codes/` **ASCII: American Standard Code for Information Infiltration** Tom Jennings, September 1999.

The Unicode Standard, Version 3.0 The Unicode Consortium, Lisa Moore ed., ISBN 0-201-61633-5, Addison Wesley Developers Press, February 2000.

CDRA: IBM - Character Data Representation Architecture - Reference and Registry, IBM SC09-2190-00, December 1996.

"Demystifying Character Sets", Andrea Vine, Multilingual Computing & Technology, **#26 Vol. 10 Issue 4**, August/September 1999; ISSN 1523-0309; Multilingual Computing Inc. Sandpoint ID, USA.

Codes, Ciphers, and Other Cryptic and Clandestine Communication Fred B. Wrixon, ISBN 1-57912-040-7, Black Dog & Leventhal Publishers, 1998.

`http://www.bobbemer.com/P-BIT.HTM` **IBM - EBCDIC and the P-bit; The biggest Computer Goof Ever** Robert Bemer.

19.20 HISTORY

15 April 2001: added UTF-8 and UTF-EBCDIC to main table, pvhp.

19.21 AUTHOR

Peter Prymmer pvhp@best.com wrote this in 1999 and 2000 with CCSID 0819 and 0037 help from Chris Leach and André Pirard A.Pirard@ulg.ac.be as well as POSIX-BC help from Thomas Dorner Thomas.Dorner@start.de. Thanks also to Vickie Cooper, Philip Newton, William Raffloer, and Joe Smith. Trademarks, registered trademarks, service marks and registered service marks used in this document are the property of their respective owners.

Now maintained by Perl5 Porters.

20 perlembed

20.1 NAME

perlembed - how to embed perl in your C program

20.2 DESCRIPTION

20.2.1 PREAMBLE

Do you want to:

Use C from Perl?
Read `perlxstut`, `perlxs`, `h2xs`, Section 28.1 [perlguts NAME], page 512, and `perlapi`.

Use a Unix program from Perl?
Read about back-quotes and about `system` and `exec` in Section 25.1 [perlfunc NAME], page 351.

Use Perl from Perl?
Read about ⟨undefined⟩ [perlfunc do], page ⟨undefined⟩ and [perlfunc eval], page 377 and [perlfunc require], page 437 and ⟨undefined⟩ [perlfunc use], page ⟨undefined⟩.

Use C from C?
Rethink your design.

Use Perl from C?
Read on...

20.2.2 ROADMAP

- Compiling your C program
- Adding a Perl interpreter to your C program
- Calling a Perl subroutine from your C program
- Evaluating a Perl statement from your C program
- Performing Perl pattern matches and substitutions from your C program
- Fiddling with the Perl stack from your C program
- Maintaining a persistent interpreter
- Maintaining multiple interpreter instances
- Using Perl modules, which themselves use C libraries, from your C program
- Embedding Perl under Win32

20.2.3 Compiling your C program

If you have trouble compiling the scripts in this documentation, you're not alone. The cardinal rule: COMPILE THE PROGRAMS IN EXACTLY THE SAME WAY THAT YOUR PERL WAS COMPILED. (Sorry for yelling.)

Also, every C program that uses Perl must link in the *perl library*. What's that, you ask? Perl is itself written in C; the perl library is the collection of compiled C programs that were used to create your perl executable (*/usr/bin/perl* or equivalent). (Corollary: you can't use Perl from your C program unless Perl has been compiled on your machine, or installed properly–that's why you shouldn't blithely copy Perl executables from machine to machine without also copying the *lib* directory.)

When you use Perl from C, your C program will–usually–allocate, "run", and deallocate a *PerlInterpreter* object, which is defined by the perl library.

If your copy of Perl is recent enough to contain this documentation (version 5.002 or later), then the perl library (and *EXTERN.h* and *perl.h*, which you'll also need) will reside in a directory that looks like this:

```
/usr/local/lib/perl5/your_architecture_here/CORE
```

or perhaps just

```
/usr/local/lib/perl5/CORE
```

or maybe something like

```
/usr/opt/perl5/CORE
```

Execute this statement for a hint about where to find CORE:

```
perl -MConfig -e 'print $Config{archlib}'
```

Here's how you'd compile the example in the next section, Section 20.2.4 [Adding a Perl interpreter to your C program], page 303, on my Linux box:

```
% gcc -O2 -Dbool=char -DHAS_BOOL -I/usr/local/include
-I/usr/local/lib/perl5/i586-linux/5.003/CORE
-L/usr/local/lib/perl5/i586-linux/5.003/CORE
-o interp interp.c -lperl -lm
```

(That's all one line.) On my DEC Alpha running old 5.003_05, the incantation is a bit different:

```
% cc -O2 -Olimit 2900 -DSTANDARD_C -I/usr/local/include
-I/usr/local/lib/perl5/alpha-dec_osf/5.00305/CORE
-L/usr/local/lib/perl5/alpha-dec_osf/5.00305/CORE -L/usr/local/lib
-D__LANGUAGE_C__ -D_NO_PROTO -o interp interp.c -lperl -lm
```

How can you figure out what to add? Assuming your Perl is post-5.001, execute a `perl -V` command and pay special attention to the "cc" and "ccflags" information.

You'll have to choose the appropriate compiler (*cc*, *gcc*, et al.) for your machine: `perl -MConfig -e 'print $Config{cc}'` will tell you what to use.

You'll also have to choose the appropriate library directory (*/usr/local/lib/...*) for your machine. If your compiler complains that certain functions are undefined, or that it can't locate *-lperl*, then you need to change the path following the `-L`. If it complains that it can't find *EXTERN.h* and *perl.h*, you need to change the path following the `-I`.

You may have to add extra libraries as well. Which ones? Perhaps those printed by

```
perl -MConfig -e 'print $Config{libs}'
```

Provided your perl binary was properly configured and installed the **ExtUtils::Embed** module will determine all of this information for you:

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

If the **ExtUtils::Embed** module isn't part of your Perl distribution, you can retrieve it from http://www.perl.com/perl/CPAN/modules/by-module/ExtUtils/ (If this documentation came from your Perl distribution, then you're running 5.004 or better and you already have it.)

The **ExtUtils::Embed** kit on CPAN also contains all source code for the examples in this document, tests, additional examples and other information you may find useful.

20.2.4 Adding a Perl interpreter to your C program

In a sense, perl (the C program) is a good example of embedding Perl (the language), so I'll demonstrate embedding with *miniperlmain.c*, included in the source distribution. Here's a bastardized, non-portable version of *miniperlmain.c* containing the essentials of embedding:

```
#include <EXTERN.h>               /* from the Perl distribution     */
#include <perl.h>                 /* from the Perl distribution     */

static PerlInterpreter *my_perl;  /***    The Perl interpreter    ***/

int main(int argc, char **argv, char **env)
{
       PERL_SYS_INIT3(&argc,&argv,&env);
       my_perl = perl_alloc();
       perl_construct(my_perl);
       PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
       perl_parse(my_perl, NULL, argc, argv, (char **)NULL);
       perl_run(my_perl);
       perl_destruct(my_perl);
       perl_free(my_perl);
       PERL_SYS_TERM();
}
```

Notice that we don't use the `env` pointer. Normally handed to `perl_parse` as its final argument, `env` here is replaced by `NULL`, which means that the current environment will be used.

The macros PERL_SYS_INIT3() and PERL_SYS_TERM() provide system-specific tune up of the C runtime environment necessary to run Perl interpreters; they should only be called once regardless of how many interpreters you create or destroy. Call PERL_SYS_INIT3() before you create your first interpreter, and PERL_SYS_TERM() after you free your last interpreter.

Since PERL_SYS_INIT3() may change `env`, it may be more appropriate to provide `env` as an argument to perl_parse().

Also notice that no matter what arguments you pass to perl_parse(), PERL_SYS_INIT3() must be invoked on the C main() argc, argv and env and only once.

Now compile this program (I'll call it *interp.c*) into an executable:

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

After a successful compilation, you'll be able to use *interp* just like perl itself:

```
% interp
print "Pretty Good Perl \n";
print "10890 - 9801 is ", 10890 - 9801;
<CTRL-D>
Pretty Good Perl
10890 - 9801 is 1089
```

or

```
% interp -e 'printf("%x", 3735928559)'
deadbeef
```

You can also read and execute Perl statements from a file while in the midst of your C program, by placing the filename in *argv[1]* before calling *perl_run*.

20.2.5 Calling a Perl subroutine from your C program

To call individual Perl subroutines, you can use any of the **call_*** functions documented in Section 7.1 [perlcall NAME], page 28. In this example we'll use `call_argv`.

That's shown below, in a program I'll call *showtime.c*.

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

int main(int argc, char **argv, char **env)
{
    char *args[] = { NULL };
    PERL_SYS_INIT3(&argc,&argv,&env);
    my_perl = perl_alloc();
    perl_construct(my_perl);

    perl_parse(my_perl, NULL, argc, argv, NULL);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;

    /*** skipping perl_run() ***/

    call_argv("showtime", G_DISCARD | G_NOARGS, args);

    perl_destruct(my_perl);
    perl_free(my_perl);
    PERL_SYS_TERM();
}
```

where *showtime* is a Perl subroutine that takes no arguments (that's the *G_NOARGS*) and for which I'll ignore the return value (that's the *G_DISCARD*). Those flags, and others, are discussed in Section 7.1 [perlcall NAME], page 28.

I'll define the *showtime* subroutine in a file called *showtime.pl*:

```
print "I shan't be printed.";
```

```
sub showtime {
    print time;
}
```

Simple enough. Now compile and run:

```
% cc -o showtime showtime.c \
    `perl -MExtUtils::Embed -e ccopts -e ldopts`
% showtime showtime.pl
818284590
```

yielding the number of seconds that elapsed between January 1, 1970 (the beginning of the Unix epoch), and the moment I began writing this sentence.

In this particular case we don't have to call *perl_run*, as we set the PL_exit_flag PERL_EXIT_DESTRUCT_END which executes END blocks in perl_destruct.

If you want to pass arguments to the Perl subroutine, you can add strings to the `NULL`-terminated `args` list passed to *call_argv*. For other data types, or to examine return values, you'll need to manipulate the Perl stack. That's demonstrated in Section 20.2.8 [Fiddling with the Perl stack from your C program], page 311.

20.2.6 Evaluating a Perl statement from your C program

Perl provides two API functions to evaluate pieces of Perl code. These are Section "eval_sv" in `perlapi` and Section "eval_pv" in `perlapi`.

Arguably, these are the only routines you'll ever need to execute snippets of Perl code from within your C program. Your code can be as long as you wish; it can contain multiple statements; it can employ ⟨undefined⟩ [perlfunc use], page ⟨undefined⟩, [perlfunc require], page 437, and ⟨undefined⟩ [perlfunc do], page ⟨undefined⟩ to include external Perl files.

eval_pv lets us evaluate individual Perl strings, and then extract variables for coercion into C types. The following program, *string.c*, executes three Perl strings, extracting an `int` from the first, a `float` from the second, and a `char *` from the third.

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

main (int argc, char **argv, char **env)
{
    char *embedding[] = { "", "-e", "0" };

    PERL_SYS_INIT3(&argc,&argv,&env);
    my_perl = perl_alloc();
    perl_construct( my_perl );

    perl_parse(my_perl, NULL, 3, embedding, NULL);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
    perl_run(my_perl);

    /** Treat $a as an integer **/
    eval_pv("$a = 3; $a **= 2", TRUE);
    printf("a = %d\n", SvIV(get_sv("a", 0)));

    /** Treat $a as a float **/
    eval_pv("$a = 3.14; $a **= 2", TRUE);
    printf("a = %f\n", SvNV(get_sv("a", 0)));

    /** Treat $a as a string **/
    eval_pv(
      "$a = 'rekcaH lreP rehtonA tsuJ'; $a = reverse($a);", TRUE);
    printf("a = %s\n", SvPV_nolen(get_sv("a", 0)));

    perl_destruct(my_perl);
    perl_free(my_perl);
    PERL_SYS_TERM();
}
```

All of those strange functions with *sv* in their names help convert Perl scalars to C types. They're described in Section 28.1 [perlguts NAME], page 512 and `perlapi`.

If you compile and run *string.c*, you'll see the results of using *SvIV()* to create an `int`, *SvNV()* to create a `float`, and *SvPV()* to create a string:

```
a = 9
a = 9.859600
a = Just Another Perl Hacker
```

In the example above, we've created a global variable to temporarily store the computed value of our eval'ed expression. It is also possible and in most cases a better strategy to fetch the return value from *eval_pv()* instead. Example:

```
...
SV *val = eval_pv("reverse 'rekcaH lreP rehtonA tsuJ'", TRUE);
printf("%s\n", SvPV_nolen(val));
...
```

This way, we avoid namespace pollution by not creating global variables and we've simplified our code as well.

20.2.7 Performing Perl pattern matches and substitutions from your C program

The *eval_sv()* function lets us evaluate strings of Perl code, so we can define some functions that use it to "specialize" in matches and substitutions: *match()*, *substitute()*, and *matches()*.

```
I32 match(SV *string, char *pattern);
```

Given a string and a pattern (e.g., `m/clasp/` or `/\b\w*\b/`, which in your C program might appear as "/\\b\\w*\\b/"), match() returns 1 if the string matches the pattern and 0 otherwise.

```
int substitute(SV **string, char *pattern);
```

Given a pointer to an SV and an =~ operation (e.g., s/bob/robert/g or tr[A-Z][a-z]), substitute() modifies the string within the SV as according to the operation, returning the number of substitutions made.

```
SSize_t matches(SV *string, char *pattern, AV **matches);
```

Given an SV, a pattern, and a pointer to an empty AV, matches() evaluates $string =~ $pattern in a list context, and fills in *matches* with the array elements, returning the number of matches found.

Here's a sample program, *match.c*, that uses all three (long lines have been wrapped here):

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

/** my_eval_sv(code, error_check)
** kinda like eval_sv(),
** but we pop the return value off the stack
**/
SV* my_eval_sv(SV *sv, I32 croak_on_error)
{
    dSP;
    SV* retval;


    PUSHMARK(SP);
    eval_sv(sv, G_SCALAR);

    SPAGAIN;
    retval = POPs;
    PUTBACK;

    if (croak_on_error && SvTRUE(ERRSV))
       croak(SvPVx_nolen(ERRSV));

    return retval;
}

/** match(string, pattern)
**
** Used for matches in a scalar context.
**
** Returns 1 if the match was successful; 0 otherwise.
**/

I32 match(SV *string, char *pattern)
{
    SV *command = newSV(0), *retval;

    sv_setpvf(command, "my $string = '%s'; $string =~ %s",
             SvPV_nolen(string), pattern);

    retval = my_eval_sv(command, TRUE);
    SvREFCNT_dec(command);

    return SvIV(retval);
}

/** substitute(string, pattern)
**
** Used for =~ operations that
** modify their left-hand side (s/// and tr///)
**
** Returns the number of successful matches, and
** modifies the input string if there were any.
**/

I32 substitute(SV **string, char *pattern)
{
    SV *command = newSV(0), *retval;

    sv_setpvf(command, "$string = '%s'; ($string =~ %s)",
             SvPV_nolen(*string), pattern);

    retval = my_eval_sv(command, TRUE);
    SvREFCNT_dec(command);

    *string = get_sv("string", 0);
    return SvIV(retval);
}

/** matches(string, pattern, matches)
**
** Used for matches in a list context.
**
** Returns the number of matches,
** and fills in **matches with the matching substrings
**/

SSize_t matches(SV *string, char *pattern, AV **match_list)
{
    SV *command = newSV(0);
    SSize_t num_matches;
    sv_setpvf(command, "my $string = ’%s’; @array = ($string =˜ %s)",
             SvPV_nolen(string), pattern);

    my_eval_sv(command, TRUE);
    SvREFCNT_dec(command);

    *match_list = get_av("array", 0);
    num_matches = av_top_index(*match_list) + 1;

    return num_matches;
}

main (int argc, char **argv, char **env)
{
    char *embedding[] = { "", "-e", "0" };
    AV *match_list;
    I32 num_matches, i;
    SV *text;

    PERL_SYS_INIT3(&argc,&argv,&env);
    my_perl = perl_alloc();
    perl_construct(my_perl);
    perl_parse(my_perl, NULL, 3, embedding, NULL);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;

    text = newSV(0);
    sv_setpv(text, "When he is at a convenience store and the "
       "bill comes to some amount like 76 cents, Maynard is "
       "aware that there is something he *should* do, something "
       "that will enable him to get back a quarter, but he has "
       "no idea *what*.  He fumbles through his red squeezey "
       "changepurse and gives the boy three extra pennies with "
       "his dollar, hoping that he might luck into the correct "
       "amount.  The boy gives him back two of his own pennies "
       "and then the big shiny quarter that is his prize. "
       "-RICHH");

    if (match(text, "m/quarter/")) /** Does text contain ’quarter’? **/
       printf("match: Text contains the word ’quarter’.\n\n");
    else
       printf("match: Text doesn’t contain the word ’quarter’.\n\n");

    if (match(text, "m/eighth/")) /** Does text contain ’eighth’? **/
       printf("match: Text contains the word ’eighth’.\n\n");
    else
       printf("match: Text doesn’t contain the word ’eighth’.\n\n");
    /** Match all occurrences of /wi../ **/
    num_matches = matches(text, "m/(wi..)/g", &match_list);
    printf("matches: m/(wi..)/g found %d matches...\n", num_matches);

    for (i = 0; i < num_matches; i++)
        printf("match: %s\n",
                 SvPV_nolen(*av_fetch(match_list, i, FALSE)));
    printf("\n");

    /** Remove all vowels from text **/
    num_matches = substitute(&text, "s/[aeiou]//gi");
    if (num_matches) {
       printf("substitute: s/[aeiou]//gi...%lu substitutions made.\n",
              (unsigned long)num_matches);
       printf("Now text is: %s\n\n", SvPV_nolen(text));
    }

    /** Attempt a substitution **/
    if (!substitute(&text, "s/Perl/C/")) {
       printf("substitute: s/Perl/C...No substitution made.\n\n");
    }

    SvREFCNT_dec(text);
    PL_perl_destruct_level = 1;
    perl_destruct(my_perl);
    perl_free(my_perl);
    PERL_SYS_TERM();
}
```

which produces the output (again, long lines have been wrapped here)

```
match: Text contains the word ’quarter’.

match: Text doesn’t contain the word ’eighth’.

matches: m/(wi..)/g found 2 matches...
match: will
match: with

substitute: s/[aeiou]//gi...139 substitutions made.
Now text is: Whn h s t  cnvnnc str nd th bll cms t sm mnt lk 76 cnts,
Mynrd s wr tht thr s smthng h *shld* d, smthng tht wll nbl hm t gt
bck qrtr, bt h hs n d *wht*.  H fmbls thrgh hs rd sqzy chngprs nd
gvs th by thr xtr pnns wth hs dllr, hpng tht h mght lck nt th crrct
mnt.  Th by gvs hm bck tw f hs wn pnns nd thn th bg shny qrtr tht s
hs prz. -RCHH

substitute: s/Perl/C...No substitution made.
```

20.2.8 Fiddling with the Perl stack from your C program

When trying to explain stacks, most computer science textbooks mumble something about spring-loaded columns of cafeteria plates: the last thing you pushed on the stack is the first thing you pop off. That'll do for our purposes: your C program will push some arguments onto "the Perl stack", shut its eyes while some magic happens, and then pop the results–the return value of your Perl subroutine–off the stack.

First you'll need to know how to convert between C types and Perl types, with newSViv() and sv_setnv() and newAV() and all their friends. They're described in Section 28.1 [perlguts NAME], page 512 and `perlapi`.

Then you'll need to know how to manipulate the Perl stack. That's described in Section 7.1 [perlcall NAME], page 28.

Once you've understood those, embedding Perl in C is easy.

Because C has no builtin function for integer exponentiation, let's make Perl's ** operator available to it (this is less useful than it sounds, because Perl implements ** with C's *pow()* function). First I'll create a stub exponentiation function in *power.pl*:

```
sub expo {
    my ($a, $b) = @_;
    return $a ** $b;
}
```

Now I'll create a C program, *power.c*, with a function *PerlPower()* that contains all the perlguts necessary to push the two arguments into *expo()* and to pop the return value out. Take a deep breath...

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

static void
PerlPower(int a, int b)
{
  dSP;                            /* initialize stack pointer      */
  ENTER;                          /* everything created after here */
  SAVETMPS;                       /* ...is a temporary variable.   */
  PUSHMARK(SP);                   /* remember the stack pointer    */
  XPUSHs(sv_2mortal(newSViv(a))); /* push the base onto the stack  */
  XPUSHs(sv_2mortal(newSViv(b))); /* push the exponent onto stack  */
  PUTBACK;                      /* make local stack pointer global */
  call_pv("expo", G_SCALAR);      /* call the function             */
  SPAGAIN;                        /* refresh stack pointer         */
                                /* pop the return value from stack */
  printf ("%d to the %dth power is %d.\n", a, b, POPi);
  PUTBACK;
  FREETMPS;                       /* free that return value        */
  LEAVE;                       /* ...and the XPUSHed "mortal" args.*/
}
```

```
int main (int argc, char **argv, char **env)
{
  char *my_argv[] = { "", "power.pl" };

  PERL_SYS_INIT3(&argc,&argv,&env);
  my_perl = perl_alloc();
  perl_construct( my_perl );

  perl_parse(my_perl, NULL, 2, my_argv, (char **)NULL);
  PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
  perl_run(my_perl);

  PerlPower(3, 4);                      /*** Compute 3 ** 4 ***/

  perl_destruct(my_perl);
  perl_free(my_perl);
  PERL_SYS_TERM();
}
```

Compile and run:

```
% cc -o power power.c `perl -MExtUtils::Embed -e ccopts -e ldopts`

% power
3 to the 4th power is 81.
```

20.2.9 Maintaining a persistent interpreter

When developing interactive and/or potentially long-running applications, it's a good idea to maintain a persistent interpreter rather than allocating and constructing a new interpreter multiple times. The major reason is speed: since Perl will only be loaded into memory once.

However, you have to be more cautious with namespace and variable scoping when using a persistent interpreter. In previous examples we've been using global variables in the default package `main`. We knew exactly what code would be run, and assumed we could avoid variable collisions and outrageous symbol table growth.

Let's say your application is a server that will occasionally run Perl code from some arbitrary file. Your server has no way of knowing what code it's going to run. Very dangerous.

If the file is pulled in by `perl_parse()`, compiled into a newly constructed interpreter, and subsequently cleaned out with `perl_destruct()` afterwards, you're shielded from most namespace troubles.

One way to avoid namespace collisions in this scenario is to translate the filename into a guaranteed-unique package name, and then compile the code into that package using [perlfunc eval], page 377. In the example below, each file will only be compiled once. Or, the application might choose to clean out the symbol table associated with the file after it's no longer needed. Using Section "call_argv" in `perlapi`, We'll call the subroutine Embed::Persistent::eval_file which lives in the file persistent.pl and pass the filename and boolean cleanup/cache flag as arguments.

Note that the process will continue to grow for each file that it uses. In addition, there might be AUTOLOADed subroutines and other conditions that cause Perl's symbol table to grow. You might want to add some logic that keeps track of the process size, or restarts itself after a certain number of requests, to ensure that memory consumption is minimized. You'll also want to scope your variables with ⟨undefined⟩ [perlfunc my], page ⟨undefined⟩ whenever possible.

```
package Embed::Persistent;
#persistent.pl

use strict;
our %Cache;
use Symbol qw(delete_package);

sub valid_package_name {
    my($string) = @_;
    $string =~ s/([^A-Za-z0-9\/])/sprintf("_%2x",unpack("C",$1))/eg;
    # second pass only for words starting with a digit
    $string =~ s|/(\d)|sprintf("/_%2x",unpack("C",$1))|eg;

    # Dress it up as a real package name
    $string =~ s|/|::|g;
    return "Embed" . $string;
}

sub eval_file {
    my($filename, $delete) = @_;
    my $package = valid_package_name($filename);
    my $mtime = -M $filename;
    if(defined $Cache{$package}{mtime}
       &&
       $Cache{$package}{mtime} <= $mtime)
    {
       # we have compiled this subroutine already,
       # it has not been updated on disk, nothing left to do
       print STDERR "already compiled $package->handler\n";
    }
    else {
       local *FH;
       open FH, $filename or die "open '$filename' $!";
       local($/) = undef;
       my $sub = <FH>;
       close FH;

       #wrap the code into a subroutine inside our unique package
```

```
        my $eval = qq{package $package; sub handler { $sub; }};
        {
            # hide our variables within this block
            my($filename,$mtime,$package,$sub);
            eval $eval;
        }
        die $@ if $@;

        #cache it unless we're cleaning out each time
        $Cache{$package}{mtime} = $mtime unless $delete;
    }

    eval {$package->handler;};
    die $@ if $@;

    delete_package($package) if $delete;

    #take a look if you want
    #print Devel::Symdump->rnew($package)->as_string, $/;
}

1;

__END__

/* persistent.c */
#include <EXTERN.h>
#include <perl.h>

/* 1 = clean out filename's symbol table after each request,
   0 = don't
*/
#ifndef DO_CLEAN
#define DO_CLEAN 0
#endif

#define BUFFER_SIZE 1024

static PerlInterpreter *my_perl = NULL;

int
main(int argc, char **argv, char **env)
{
    char *embedding[] = { "", "persistent.pl" };
    char *args[] = { "", DO_CLEAN, NULL };
    char filename[BUFFER_SIZE];
    int exitstatus = 0;
    PERL_SYS_INIT3(&argc,&argv,&env);
    if((my_perl = perl_alloc()) == NULL) {
       fprintf(stderr, "no memory!");
       exit(1);
    }
    perl_construct(my_perl);

    PL_origalen = 1; /* don't let $0 assignment update the
                        proctitle or embedding[0] */
    exitstatus = perl_parse(my_perl, NULL, 2, embedding, NULL);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
    if(!exitstatus) {
       exitstatus = perl_run(my_perl);

       while(printf("Enter file name: ") &&
             fgets(filename, BUFFER_SIZE, stdin)) {

           filename[strlen(filename)-1] = '\0'; /* strip \n */
           /* call the subroutine,
                    passing it the filename as an argument */
           args[0] = filename;
           call_argv("Embed::Persistent::eval_file",
                          G_DISCARD | G_EVAL, args);

           /* check $@ */
           if(SvTRUE(ERRSV))
               fprintf(stderr, "eval error: %s\n", SvPV_nolen(ERRSV));
       }
    }

    PL_perl_destruct_level = 0;
    perl_destruct(my_perl);
    perl_free(my_perl);
    PERL_SYS_TERM();
    exit(exitstatus);
}
```

Now compile:

```
% cc -o persistent persistent.c \
       `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

Here's an example script file:

```
#test.pl
my $string = "hello";
foo($string);

sub foo {
    print "foo says: @_\n";
}
```

Now run:

```
% persistent
Enter file name: test.pl
foo says: hello
Enter file name: test.pl
already compiled Embed::test_2epl->handler
foo says: hello
Enter file name: ^C
```

20.2.10 Execution of END blocks

Traditionally END blocks have been executed at the end of the perl_run. This causes problems for applications that never call perl_run. Since perl 5.7.2 you can specify `PL_exit_flags |= PERL_EXIT_DESTRUCT_END` to get the new behaviour. This also enables the running of END blocks if the perl_parse fails and `perl_destruct` will return the exit value.

20.2.11 $0 assignments

When a perl script assigns a value to $0 then the perl runtime will try to make this value show up as the program name reported by "ps" by updating the memory pointed to by the argv passed to perl_parse() and also calling API functions like setproctitle() where available. This behaviour might not be appropriate when embedding perl and can be disabled by assigning the value `1` to the variable `PL_origalen` before perl_parse() is called.

The `persistent.c` example above is for instance likely to segfault when $0 is assigned to if the `PL_origalen = 1;` assignment is removed. This because perl will try to write to the read only memory of the `embedding[]` strings.

20.2.12 Maintaining multiple interpreter instances

Some rare applications will need to create more than one interpreter during a session. Such an application might sporadically decide to release any resources associated with the interpreter.

The program must take care to ensure that this takes place *before* the next interpreter is constructed. By default, when perl is not built with any special options, the global variable `PL_perl_destruct_level` is set to `0`, since extra cleaning isn't usually needed when a program only ever creates a single interpreter in its entire lifetime.

Setting `PL_perl_destruct_level` to `1` makes everything squeaky clean:

```
while(1) {
    ...
    /* reset global variables here with PL_perl_destruct_level = 1 */
    PL_perl_destruct_level = 1;
    perl_construct(my_perl);
    ...
    /* clean and reset _everything_ during perl_destruct */
    PL_perl_destruct_level = 1;
    perl_destruct(my_perl);
    perl_free(my_perl);
    ...
    /* let's go do it again! */
}
```

When *perl_destruct()* is called, the interpreter's syntax parse tree and symbol tables are cleaned up, and global variables are reset. The second assignment to `PL_perl_destruct_level` is needed because perl_construct resets it to `0`.

Now suppose we have more than one interpreter instance running at the same time. This is feasible, but only if you used the Configure option `-Dusemultiplicity` or the options `-Dusethreads -Duseithreads` when building perl. By default, enabling one of these Configure options sets the per-interpreter global variable `PL_perl_destruct_level` to `1`, so that thorough cleaning is automatic and interpreter variables are initialized correctly. Even if you don't intend to run two or more interpreters at the same time, but to run them sequentially, like in the above example, it is recommended to build perl with the `-Dusemultiplicity` option otherwise some interpreter variables may not be initialized correctly between consecutive runs and your application may crash.

See also Section "Thread-aware system interfaces" in `perlxs`.

Using `-Dusethreads -Duseithreads` rather than `-Dusemultiplicity` is more appropriate if you intend to run multiple interpreters concurrently in different threads, because it enables support for linking in the thread libraries of your system with the interpreter.

Let's give it a try:

```
#include <EXTERN.h>
#include <perl.h>

/* we're going to embed two interpreters */

#define SAY_HELLO "-e", "print qq(Hi, I'm $^X\n)"

int main(int argc, char **argv, char **env)
{
    PerlInterpreter *one_perl, *two_perl;
    char *one_args[] = { "one_perl", SAY_HELLO };
    char *two_args[] = { "two_perl", SAY_HELLO };

    PERL_SYS_INIT3(&argc,&argv,&env);
    one_perl = perl_alloc();
    two_perl = perl_alloc();

    PERL_SET_CONTEXT(one_perl);
    perl_construct(one_perl);
    PERL_SET_CONTEXT(two_perl);
    perl_construct(two_perl);

    PERL_SET_CONTEXT(one_perl);
    perl_parse(one_perl, NULL, 3, one_args, (char **)NULL);
    PERL_SET_CONTEXT(two_perl);
    perl_parse(two_perl, NULL, 3, two_args, (char **)NULL);

    PERL_SET_CONTEXT(one_perl);
    perl_run(one_perl);
    PERL_SET_CONTEXT(two_perl);
    perl_run(two_perl);

    PERL_SET_CONTEXT(one_perl);
    perl_destruct(one_perl);
    PERL_SET_CONTEXT(two_perl);
    perl_destruct(two_perl);

    PERL_SET_CONTEXT(one_perl);
    perl_free(one_perl);
    PERL_SET_CONTEXT(two_perl);
    perl_free(two_perl);
    PERL_SYS_TERM();
}
```

Note the calls to PERL_SET_CONTEXT(). These are necessary to initialize the global state that tracks which interpreter is the "current" one on the particular process or thread that may be running it. It should always be used if you have more than one interpreter and are making perl API calls on both interpreters in an interleaved fashion.

PERL_SET_CONTEXT(interp) should also be called whenever `interp` is used by a thread that did not create it (using either perl_alloc(), or the more esoteric perl_clone()).

Compile as usual:

```
% cc -o multiplicity multiplicity.c \
  `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

Run it, Run it:

```
% multiplicity
Hi, I'm one_perl
Hi, I'm two_perl
```

20.2.13 Using Perl modules, which themselves use C libraries, from your C program

If you've played with the examples above and tried to embed a script that *use()*s a Perl module (such as *Socket*) which itself uses a C or C++ library, this probably happened:

```
Can't load module Socket, dynamic loading not available in this perl.
  (You may need to build a new perl executable which either supports
  dynamic loading or has the Socket module statically linked into it.)
```

What's wrong?

Your interpreter doesn't know how to communicate with these extensions on its own. A little glue will help. Up until now you've been calling *perl_parse()*, handing it NULL for the second argument:

```
perl_parse(my_perl, NULL, argc, my_argv, NULL);
```

That's where the glue code can be inserted to create the initial contact between Perl and linked C/C++ routines. Let's take a look some pieces of *perlmain.c* to see how Perl does this:

```
static void xs_init (pTHX);

EXTERN_C void boot_DynaLoader (pTHX_ CV* cv);
EXTERN_C void boot_Socket (pTHX_ CV* cv);


EXTERN_C void
xs_init(pTHX)
{
       char *file = __FILE__;
       /* DynaLoader is a special case */
       newXS("DynaLoader::boot_DynaLoader", boot_DynaLoader, file);
       newXS("Socket::bootstrap", boot_Socket, file);
}
```

Simply put: for each extension linked with your Perl executable (determined during its initial configuration on your computer or when adding a new extension), a Perl subroutine is created to incorporate the extension's routines. Normally, that subroutine is named *Module::bootstrap()* and is invoked when you say *use Module*. In turn, this hooks into an XSUB, *boot_Module*, which creates a Perl counterpart for each of the extension's XSUBs. Don't worry about this part; leave that to the *xsubpp* and extension authors. If your extension is dynamically loaded, DynaLoader creates *Module::bootstrap()* for you on the fly. In fact, if you have a working DynaLoader then there is rarely any need to link in any other extensions statically.

Once you have this code, slap it into the second argument of *perl_parse()*:

```
perl_parse(my_perl, xs_init, argc, my_argv, NULL);
```

Then compile:

```
% cc -o interp interp.c ‘perl -MExtUtils::Embed -e ccopts -e ldopts‘

% interp
  use Socket;
  use SomeDynamicallyLoadedModule;

  print "Now I can use extensions!\n"'
```

ExtUtils::Embed can also automate writing the *xs_init* glue code.

```
% perl -MExtUtils::Embed -e xsinit -- -o perlxsi.c
% cc -c perlxsi.c ‘perl -MExtUtils::Embed -e ccopts‘
% cc -c interp.c  ‘perl -MExtUtils::Embed -e ccopts‘
% cc -o interp perlxsi.o interp.o ‘perl -MExtUtils::Embed -e ldopts‘
```

Consult `perlxs`, Section 28.1 [perlguts NAME], page 512, and `perlapi` for more details.

20.2.14 Using embedded Perl with POSIX locales

(See Section 38.1 [perllocale NAME], page 701 for information about these.) When a Perl interpreter normally starts up, it tells the system it wants to use the system's default locale. This is often, but not necessarily, the "C" or "POSIX" locale. Absent a `"use locale"` within the perl code, this mostly has no effect (but see [perllocale Not within the scope of "use locale"], page 703). Also, there is not a problem if the locale you want to use in your embedded Perl is the same as the system default. However, this doesn't work if you have set up and want to use a locale that isn't the system default one. Starting in Perl v5.20, you can tell the embedded Perl interpreter that the locale is already properly set up, and to skip doing its own normal initialization. It skips if the environment variable `PERL_SKIP_LOCALE_INIT` is set (even if set to 0 or `""`). A Perl that has this capability will define the C pre-processor symbol `HAS_SKIP_LOCALE_INIT`. This allows code that has to work with multiple Perl versions to do some sort of work-around when confronted with an earlier Perl.

20.3 Hiding Perl_

If you completely hide the short forms of the Perl public API, add -DPERL_NO_SHORT_NAMES to the compilation flags. This means that for example instead of writing

```
warn("%d bottles of beer on the wall", bottlecount);
```

you will have to write the explicit full form

```
Perl_warn(aTHX_ "%d bottles of beer on the wall", bottlecount);
```

(See Section 28.9.1 [perlguts Background and PERL_IMPLICIT_CONTEXT], page 547 for the explanation of the `aTHX_`.) Hiding the short forms is very useful for avoiding all sorts of nasty (C preprocessor or otherwise) conflicts with other software packages (Perl defines about 2400 APIs with these short names, take or leave few hundred, so there certainly is room for conflict.)

20.4 MORAL

You can sometimes *write faster code* in C, but you can always *write code faster* in Perl. Because you can use each from the other, combine them as you wish.

20.5 AUTHOR

Jon Orwant `<orwant@media.mit.edu>` and Doug MacEachern `<dougm@covalent.net>`, with small contributions from Tim Bunce, Tom Christiansen, Guy Decoux, Hallvard Furuseth, Dov Grobgeld, and Ilya Zakharevich.

Doug MacEachern has an article on embedding in Volume 1, Issue 4 of The Perl Journal (http://www.tpj.com/). Doug is also the developer of the most widely-used Perl embedding: the mod_perl system (perl.apache.org), which embeds Perl in the Apache web server. Oracle, Binary Evolution, ActiveState, and Ben Sugars's nsapi_perl have used this model for Oracle, Netscape and Internet Information Server Perl plugins.

20.6 COPYRIGHT

21 perlexperiment

21.1 NAME

perlexperiment - A listing of experimental features in Perl

21.2 DESCRIPTION

This document lists the current and past experimental features in the perl core. Although all of these are documented with their appropriate topics, this succinct listing gives you an overview and basic facts about their status.

So far we've merely tried to find and list the experimental features and infer their inception, versions, etc. There's a lot of speculation here.

21.2.1 Current experiments

`our` can now have an experimental optional attribute `unique`

Introduced in Perl 5.8.0

Deprecated in Perl 5.10.0

The ticket for this feature is [perl #119313] (`https://rt.perl.org/rt3/Ticket/Display.html?id=119313`).

Smart match (`~~`)

Introduced in Perl 5.10.0

Modified in Perl 5.10.1, 5.12.0

Using this feature triggers warnings in the category `experimental::smartmatch`.

The ticket for this feature is [perl #119317] (`https://rt.perl.org/rt3/Ticket/Display.html?id=119317`).

Lexical `$_`

Introduced in Perl 5.10.0

Using this feature triggers warnings in the category `experimental::lexical_topic`.

The ticket for this feature is [perl #119315] (`https://rt.perl.org/rt3/Ticket/Display.html?id=119315`).

Pluggable keywords

The ticket for this feature is [perl #119455] (`https://rt.perl.org/rt3/Ticket/Display.html?id=119455`).

See Section "PL_keyword_plugin" in `perlapi` for the mechanism.

Introduced in: Perl 5.11.2

Array and hash container functions accept references

Introduced in Perl 5.14.0

The ticket for this feature is [perl #119437] (`https://rt.perl.org/rt3/Ticket/Display.html?id=119437`).

Lexical subroutines

Introduced in: Perl 5.18

See also: Section 73.3.6 [perlsub Lexical Subroutines], page 1232

Using this feature triggers warnings in the category `experimental::lexical_subs`.

The ticket for this feature is [perl #120085] (`https://rt.perl.org/rt3/Ticket/Display.html?id=120085`).

Regular Expression Set Operations

Introduced in: Perl 5.18

The ticket for this feature is [perl #119451] (`https://rt.perl.org/rt3/Ticket/Display.html?id=119451`).

See also: Section 61.2.3.9 [perlrecharclass Extended Bracketed Character Classes], page 1073

Using this feature triggers warnings in the category `experimental::regex_sets`.

Subroutine signatures

Introduced in Perl 5.20.0

Using this feature triggers warnings in the category `experimental::signatures`.

The ticket for this feature is [perl #121481] (`https://rt.perl.org/Ticket/Display.html?id=121481`).

Postfix dereference syntax

Introduced in Perl 5.20.0

Using this feature triggers warnings in the category `experimental::postderef`.

The ticket for this feature is [perl #120162] (`https://rt.perl.org:443/rt3/Ticket/Display.html?id=120162`).

Aliasing via reference

Introduced in Perl 5.22.0

Using this feature triggers warnings in the category `experimental::refaliasing`.

The ticket for this feature is [perl #122947] (`https://rt.perl.org/rt3/Ticket/Display.html?id=122947`).

See also: Section 62.6 [perlref Assigning to References], page 1089

The "const" attribute

Introduced in Perl 5.22.0

Using this feature triggers warnings in the category `experimental::const_attr`.

The ticket for this feature is [perl #123630] (`https://rt.perl.org/rt3/Ticket/Display.html?id=123630`).

See also: Section 73.3.11 [perlsub Constant Functions], page 1241

use re 'strict';

Introduced in Perl 5.22.0

Using this feature triggers warnings in the category `experimental::re_strict`.

See Section "'strict' mode" in `re`

String- and number-specific bitwise operators

Introduced in: Perl 5.22.0

See also: Section 48.2.36 [perlop Bitwise String Operators], page 848

Using this feature triggers warnings in the category `experimental::bitwise`.

The ticket for this feature is [perl #123707] (`https://rt.perl.org/rt3/Ticket/Display.html?id=123707`).

The <:win32> IO pseudolayer

The ticket for this feature is [perl #119453] (`https://rt.perl.org/rt3/Ticket/Display.html?id=119453`).

See also Section 69.1 [perlrun NAME], page 1176

There is an `installhtml` target in the Makefile.

The ticket for this feature is [perl #116487] (`https://rt.perl.org/rt3/Ticket/Display.html?id=116487`).

Unicode in Perl on EBCDIC

21.2.2 Accepted features

These features were so wildly successful and played so well with others that we decided to remove their experimental status and admit them as full, stable features in the world of Perl, lavishing all the benefits and luxuries thereof. They are also awarded +5 Stability and +3 Charisma.

64-bit support

Introduced in Perl 5.005

die accepts a reference

Introduced in Perl 5.005

DB module

Introduced in Perl 5.6.0

See also Section 15.1 [perldebug NAME], page 120, Section 14.1 [perldebtut NAME], page 107

Weak references

Introduced in Perl 5.6.0

Internal file glob

Introduced in Perl 5.6.0

fork() emulation

Introduced in Perl 5.6.1

See also Section 23.1 [perlfork NAME], page 337

-Dusemultiplicity -Duseithreads

Introduced in Perl 5.6.0

Accepted in Perl 5.8.0

Support for long doubles

Introduced in Perl 5.6.0

Accepted in Perl 5.8.1

The \N regex character class

The \N character class, not to be confused with the named character sequence \N{NAME}, denotes any non-newline character in a regular expression.

Introduced in Perl 5.12

Exact version of acceptance unclear, but no later than Perl 5.18.

(?{code}) and (??{ code })

Introduced in Perl 5.6.0

Accepted in Perl 5.20.0

See also Section 58.1 [perlre NAME], page 989

Linux abstract Unix domain sockets

Introduced in Perl 5.9.2

Accepted before Perl 5.20.0. The Socket library is now primarily maintained on CPAN, rather than in the perl core.

See also Socket

Lvalue subroutines

Introduced in Perl 5.6.0

Accepted in Perl 5.20.0

See also Section 73.1 [perlsub NAME], page 1216

Backtracking control verbs

(*ACCEPT)

Introduced in: Perl 5.10

Accepted in Perl 5.20.0

The <:pop> IO pseudolayer

See also Section 69.1 [perlrun NAME], page 1176

Accepted in Perl 5.20.0

\s in regexp matches vertical tab

Accepted in Perl 5.22.0

21.2.3 Removed features

These features are no longer considered experimental and their functionality has disappeared. It's your own fault if you wrote production programs using these features after we explicitly told you not to (see Section 55.1 [perlpolicy NAME], page 943).

5.005-style threading

Introduced in Perl 5.005

Removed in Perl 5.10

perlcc

Introduced in Perl 5.005

Moved from Perl 5.9.0 to CPAN

The pseudo-hash data type

Introduced in Perl 5.6.0

Removed in Perl 5.9.0

GetOpt::Long Options can now take multiple values at once (experimental)

`Getopt::Long` upgraded to version 2.35

Removed in Perl 5.8.8

Assertions

The `-A` command line switch

Introduced in Perl 5.9.0

Removed in Perl 5.9.5

Test::Harness::Straps

Moved from Perl 5.10.1 to CPAN

`legacy`

The experimental `legacy` pragma was swallowed by the `feature` pragma.

Introduced in: 5.11.2

Removed in: 5.11.3

21.3 SEE ALSO

For a complete list of features check `feature`.

21.4 AUTHORS

brian d foy `<brian.d.foy@gmail.com>`

Sébastien Aperghis-Tramoni `<saper@cpan.org>`

21.5 COPYRIGHT

21.6 LICENSE

22 perlfilter

22.1 NAME

perlfilter - Source Filters

22.2 DESCRIPTION

This article is about a little-known feature of Perl called *source filters.* Source filters alter the program text of a module before Perl sees it, much as a C preprocessor alters the source text of a C program before the compiler sees it. This article tells you more about what source filters are, how they work, and how to write your own.

The original purpose of source filters was to let you encrypt your program source to prevent casual piracy. This isn't all they can do, as you'll soon learn. But first, the basics.

22.3 CONCEPTS

Before the Perl interpreter can execute a Perl script, it must first read it from a file into memory for parsing and compilation. If that script itself includes other scripts with a `use` or `require` statement, then each of those scripts will have to be read from their respective files as well.

Now think of each logical connection between the Perl parser and an individual file as a *source stream.* A source stream is created when the Perl parser opens a file, it continues to exist as the source code is read into memory, and it is destroyed when Perl is finished parsing the file. If the parser encounters a `require` or `use` statement in a source stream, a new and distinct stream is created just for that file.

The diagram below represents a single source stream, with the flow of source from a Perl script file on the left into the Perl parser on the right. This is how Perl normally operates.

```
file -------> parser
```

There are two important points to remember:

1. Although there can be any number of source streams in existence at any given time, only one will be active.
2. Every source stream is associated with only one file.

A source filter is a special kind of Perl module that intercepts and modifies a source stream before it reaches the parser. A source filter changes our diagram like this:

```
file ----> filter ----> parser
```

If that doesn't make much sense, consider the analogy of a command pipeline. Say you have a shell script stored in the compressed file *trial.gz.* The simple pipeline command below runs the script without needing to create a temporary file to hold the uncompressed file.

```
gunzip -c trial.gz | sh
```

In this case, the data flow from the pipeline can be represented as follows:

```
trial.gz ----> gunzip ----> sh
```

With source filters, you can store the text of your script compressed and use a source filter to uncompress it for Perl's parser:

```
 compressed           gunzip
Perl program ---> source filter ---> parser
```

22.4 USING FILTERS

So how do you use a source filter in a Perl script? Above, I said that a source filter is just a special kind of module. Like all Perl modules, a source filter is invoked with a use statement.

Say you want to pass your Perl source through the C preprocessor before execution. As it happens, the source filters distribution comes with a C preprocessor filter module called Filter::cpp.

Below is an example program, `cpp_test`, which makes use of this filter. Line numbers have been added to allow specific lines to be referenced easily.

```
1: use Filter::cpp;
2: #define TRUE 1
3: $a = TRUE;
4: print "a = $a\n";
```

When you execute this script, Perl creates a source stream for the file. Before the parser processes any of the lines from the file, the source stream looks like this:

```
cpp_test ---------> parser
```

Line 1, `use Filter::cpp`, includes and installs the `cpp` filter module. All source filters work this way. The use statement is compiled and executed at compile time, before any more of the file is read, and it attaches the cpp filter to the source stream behind the scenes. Now the data flow looks like this:

```
cpp_test ----> cpp filter ----> parser
```

As the parser reads the second and subsequent lines from the source stream, it feeds those lines through the `cpp` source filter before processing them. The `cpp` filter simply passes each line through the real C preprocessor. The output from the C preprocessor is then inserted back into the source stream by the filter.

```
               .-> cpp --.
               |         |
               |         |
               |       <-'
cpp_test ----> cpp filter ----> parser
```

The parser then sees the following code:

```
use Filter::cpp;
$a = 1;
print "a = $a\n";
```

Let's consider what happens when the filtered code includes another module with use:

```
1: use Filter::cpp;
2: #define TRUE 1
3: use Fred;
4: $a = TRUE;
5: print "a = $a\n";
```

The cpp filter does not apply to the text of the Fred module, only to the text of the file that used it (cpp_test). Although the use statement on line 3 will pass through the cpp filter, the module that gets included (Fred) will not. The source streams look like this after line 3 has been parsed and before line 4 is parsed:

```
cpp_test ---> cpp filter ---> parser (INACTIVE)

Fred.pm ----> parser
```

As you can see, a new stream has been created for reading the source from Fred.pm. This stream will remain active until all of Fred.pm has been parsed. The source stream for cpp_test will still exist, but is inactive. Once the parser has finished reading Fred.pm, the source stream associated with it will be destroyed. The source stream for cpp_test then becomes active again and the parser reads line 4 and subsequent lines from cpp_test.

You can use more than one source filter on a single file. Similarly, you can reuse the same filter in as many files as you like.

For example, if you have a uuencoded and compressed source file, it is possible to stack a uudecode filter and an uncompression filter like this:

```
use Filter::uudecode; use Filter::uncompress;
M’XL(".H<US4’’V9I;F%L’)Q;>7/;1I;_>_I3=&E=%:F*I"T?22Q/
M6]9*<IQCO*XFT"O[PL%%’Y+IG?WN^ZYN-$’J.[.JE$,20/?K=_[>
...
```

Once the first line has been processed, the flow will look like this:

```
file ---> uudecode ---> uncompress ---> parser
           filter         filter
```

Data flows through filters in the same order they appear in the source file. The uudecode filter appeared before the uncompress filter, so the source file will be uudecoded before it’s uncompressed.

22.5 WRITING A SOURCE FILTER

There are three ways to write your own source filter. You can write it in C, use an external program as a filter, or write the filter in Perl. I won’t cover the first two in any great detail, so I’ll get them out of the way first. Writing the filter in Perl is most convenient, so I’ll devote the most space to it.

22.6 WRITING A SOURCE FILTER IN C

The first of the three available techniques is to write the filter completely in C. The external module you create interfaces directly with the source filter hooks provided by Perl.

The advantage of this technique is that you have complete control over the implementation of your filter. The big disadvantage is the increased complexity required to write the filter - not only do you need to understand the source filter hooks, but you also need a reasonable knowledge of Perl guts. One of the few times it is worth going to this trouble is when writing a source scrambler. The decrypt filter (which unscrambles the source before Perl parses it) included with the source filter distribution is an example of a C source filter (see Decryption Filters, below).

Decryption Filters

All decryption filters work on the principle of "security through obscurity." Regardless of how well you write a decryption filter and how strong your encryption algorithm is, anyone determined enough can retrieve the original source code. The reason is quite simple - once the decryption filter has decrypted the source back to its original form, fragments of it will be stored in the computer's memory as Perl parses it. The source might only be in memory for a short period of time, but anyone possessing a debugger, skill, and lots of patience can eventually reconstruct your program.

That said, there are a number of steps that can be taken to make life difficult for the potential cracker. The most important: Write your decryption filter in C and statically link the decryption module into the Perl binary. For further tips to make life difficult for the potential cracker, see the file *decrypt.pm* in the source filters distribution.

22.7 CREATING A SOURCE FILTER AS A SEPARATE EXECUTABLE

An alternative to writing the filter in C is to create a separate executable in the language of your choice. The separate executable reads from standard input, does whatever processing is necessary, and writes the filtered data to standard output. `Filter::cpp` is an example of a source filter implemented as a separate executable - the executable is the C preprocessor bundled with your C compiler.

The source filter distribution includes two modules that simplify this task: `Filter::exec` and `Filter::sh`. Both allow you to run any external executable. Both use a coprocess to control the flow of data into and out of the external executable. (For details on coprocesses, see Stephens, W.R., "Advanced Programming in the UNIX Environment." Addison-Wesley, ISBN 0-210-56317-7, pages 441-445.) The difference between them is that `Filter::exec` spawns the external command directly, while `Filter::sh` spawns a shell to execute the external command. (Unix uses the Bourne shell; NT uses the cmd shell.) Spawning a shell allows you to make use of the shell metacharacters and redirection facilities.

Here is an example script that uses `Filter::sh`:

```
use Filter::sh 'tr XYZ PQR';
$a = 1;
print "XYZ a = $a\n";
```

The output you'll get when the script is executed:

```
PQR a = 1
```

Writing a source filter as a separate executable works fine, but a small performance penalty is incurred. For example, if you execute the small example above, a separate subprocess will be created to run the Unix `tr` command. Each use of the filter requires its own subprocess. If creating subprocesses is expensive on your system, you might want to consider one of the other options for creating source filters.

22.8 WRITING A SOURCE FILTER IN PERL

The easiest and most portable option available for creating your own source filter is to write it completely in Perl. To distinguish this from the previous two techniques, I'll call it a Perl source filter.

To help understand how to write a Perl source filter we need an example to study. Here is a complete source filter that performs rot13 decoding. (Rot13 is a very simple encryption scheme used in Usenet postings to hide the contents of offensive posts. It moves every letter forward thirteen places, so that A becomes N, B becomes O, and Z becomes M.)

```
package Rot13;

use Filter::Util::Call;

sub import {
   my ($type) = @_;
   my ($ref) = [];
   filter_add(bless $ref);
}

sub filter {
   my ($self) = @_;
   my ($status);

   tr/n-za-mN-ZA-M/a-zA-Z/
      if ($status = filter_read()) > 0;
   $status;
}

1;
```

All Perl source filters are implemented as Perl classes and have the same basic structure as the example above.

First, we include the `Filter::Util::Call` module, which exports a number of functions into your filter's namespace. The filter shown above uses two of these functions, `filter_add()` and `filter_read()`.

Next, we create the filter object and associate it with the source stream by defining the `import` function. If you know Perl well enough, you know that `import` is called automatically every time a module is included with a use statement. This makes `import` the ideal place to both create and install a filter object.

In the example filter, the object (`$ref`) is blessed just like any other Perl object. Our example uses an anonymous array, but this isn't a requirement. Because this example doesn't need to store any context information, we could have used a scalar or hash reference just as well. The next section demonstrates context data.

The association between the filter object and the source stream is made with the `filter_add()` function. This takes a filter object as a parameter (`$ref` in this case) and installs it in the source stream.

Finally, there is the code that actually does the filtering. For this type of Perl source filter, all the filtering is done in a method called `filter()`. (It is also possible to write a Perl source filter using a closure. See the `Filter::Util::Call` manual page for more details.) It's called every time the Perl parser needs another line of source to process. The `filter()` method, in turn, reads lines from the source stream using the `filter_read()` function.

If a line was available from the source stream, `filter_read()` returns a status value greater than zero and appends the line to `$_`. A status value of zero indicates end-of-file, less than zero means an error. The filter function itself is expected to return its status in the same way, and put the filtered line it wants written to the source stream in `$_`. The use of `$_` accounts for the brevity of most Perl source filters.

In order to make use of the rot13 filter we need some way of encoding the source file in rot13 format. The script below, `mkrot13`, does just that.

```
die "usage mkrot13 filename\n" unless @ARGV;
my $in = $ARGV[0];
my $out = "$in.tmp";
open(IN, "<$in") or die "Cannot open file $in: $!\n";
open(OUT, ">$out") or die "Cannot open file $out: $!\n";

print OUT "use Rot13;\n";
while (<IN>) {
   tr/a-zA-Z/n-za-mN-ZA-M/;
   print OUT;
}

close IN;
close OUT;
unlink $in;
rename $out, $in;
```

If we encrypt this with `mkrot13`:

```
print " hello fred \n";
```

the result will be this:

```
use Rot13;
cevag "uryyb serq\a";
```

Running it produces this output:

```
hello fred
```

22.9 USING CONTEXT: THE DEBUG FILTER

The rot13 example was a trivial example. Here's another demonstration that shows off a few more features.

Say you wanted to include a lot of debugging code in your Perl script during development, but you didn't want it available in the released product. Source filters offer a solution. In order to keep the example simple, let's say you wanted the debugging output to be controlled by an environment variable, DEBUG. Debugging code is enabled if the variable exists, otherwise it is disabled.

Two special marker lines will bracket debugging code, like this:

```
## DEBUG_BEGIN
if ($year > 1999) {
   warn "Debug: millennium bug in year $year\n";
}
## DEBUG_END
```

The filter ensures that Perl parses the code between the <DEBUG_BEGIN> and DEBUG_END markers only when the DEBUG environment variable exists. That means that when DEBUG does exist, the code above should be passed through the filter unchanged. The marker lines can also be passed through as-is, because the Perl parser will see them as comment lines. When DEBUG isn't set, we need a way to disable the debug code. A simple way to achieve that is to convert the lines between the two markers into comments:

```
## DEBUG_BEGIN
#if ($year > 1999) {
#     warn "Debug: millennium bug in year $year\n";
#}
## DEBUG_END
```

Here is the complete Debug filter:

```
package Debug;

use strict;
use warnings;
use Filter::Util::Call;

use constant TRUE => 1;
use constant FALSE => 0;

sub import {
   my ($type) = @_;
   my (%context) = (
     Enabled => defined $ENV{DEBUG},
     InTraceBlock => FALSE,
     Filename => (caller)[1],
     LineNo => 0,
     LastBegin => 0,
   );
   filter_add(bless \%context);
}

sub Die {
   my ($self) = shift;
   my ($message) = shift;
   my ($line_no) = shift || $self->{LastBegin};
   die "$message at $self->{Filename} line $line_no.\n"
}

sub filter {
   my ($self) = @_;
   my ($status);
   $status = filter_read();
   ++ $self->{LineNo};

   # deal with EOF/error first
   if ($status <= 0) {
       $self->Die("DEBUG_BEGIN has no DEBUG_END")
           if $self->{InTraceBlock};
       return $status;
   }

   if ($self->{InTraceBlock}) {
      if (/^\s*##\s*DEBUG_BEGIN/ ) {
          $self->Die("Nested DEBUG_BEGIN", $self->{LineNo})
      } elsif (/^\s*##\s*DEBUG_END/) {
          $self->{InTraceBlock} = FALSE;
      }

      # comment out the debug lines when the filter is disabled
      s/^/#/ if ! $self->{Enabled};
   } elsif ( /^\s*##\s*DEBUG_BEGIN/ ) {
      $self->{InTraceBlock} = TRUE;
      $self->{LastBegin} = $self->{LineNo};
   } elsif ( /^\s*##\s*DEBUG_END/ ) {
      $self->Die("DEBUG_END has no DEBUG_BEGIN", $self->{LineNo});
   }
   return $status;
}

1;
```

The big difference between this filter and the previous example is the use of context data in the filter object. The filter object is based on a hash reference, and is used to keep various pieces of context information between calls to the filter function. All but two of the hash fields are used for error reporting. The first of those two, Enabled, is used by the filter to determine whether the debugging code should be given to the Perl parser. The second, InTraceBlock, is true when the filter has encountered a `DEBUG_BEGIN` line, but has not yet encountered the following `DEBUG_END` line.

If you ignore all the error checking that most of the code does, the essence of the filter is as follows:

```
sub filter {
   my ($self) = @_;
```

```
    my ($status);
    $status = filter_read();

    # deal with EOF/error first
    return $status if $status <= 0;
    if ($self->{InTraceBlock}) {
       if (/^\s*##\s*DEBUG_END/) {
          $self->{InTraceBlock} = FALSE
       }

       # comment out debug lines when the filter is disabled
       s/^/#/ if ! $self->{Enabled};
    } elsif ( /^\s*##\s*DEBUG_BEGIN/ ) {
       $self->{InTraceBlock} = TRUE;
    }
    return $status;
 }
```

Be warned: just as the C-preprocessor doesn't know C, the Debug filter doesn't know Perl. It can be fooled quite easily:

```
print <<EOM;
##DEBUG_BEGIN
EOM
```

Such things aside, you can see that a lot can be achieved with a modest amount of code.

22.10 CONCLUSION

You now have better understanding of what a source filter is, and you might even have a possible use for them. If you feel like playing with source filters but need a bit of inspiration, here are some extra features you could add to the Debug filter.

First, an easy one. Rather than having debugging code that is all-or-nothing, it would be much more useful to be able to control which specific blocks of debugging code get included. Try extending the syntax for debug blocks to allow each to be identified. The contents of the DEBUG environment variable can then be used to control which blocks get included.

Once you can identify individual blocks, try allowing them to be nested. That isn't difficult either.

Here is an interesting idea that doesn't involve the Debug filter. Currently Perl subroutines have fairly limited support for formal parameter lists. You can specify the number of parameters and their type, but you still have to manually take them out of the @_ array yourself. Write a source filter that allows you to have a named parameter list. Such a filter would turn this:

```
sub MySub ($first, $second, @rest) { ... }
```

into this:

```
sub MySub($$@) {
   my ($first) = shift;
   my ($second) = shift;
```

```
        my (@rest) = @_;
        ...
    }
```

Finally, if you feel like a real challenge, have a go at writing a full-blown Perl macro preprocessor as a source filter. Borrow the useful features from the C preprocessor and any other macro processors you know. The tricky bit will be choosing how much knowledge of Perl's syntax you want your filter to have.

22.11 LIMITATIONS

Source filters only work on the string level, thus are highly limited in its ability to change source code on the fly. It cannot detect comments, quoted strings, heredocs, it is no replacement for a real parser. The only stable usage for source filters are encryption, compression, or the byteloader, to translate binary code back to source code.

See for example the limitations in Switch, which uses source filters, and thus is does not work inside a string eval, the presence of regexes with embedded newlines that are specified with raw /.../ delimiters and don't have a modifier //x are indistinguishable from code chunks beginning with the division operator /. As a workaround you must use m/.../ or m?...? for such patterns. Also, the presence of regexes specified with raw ?...? delimiters may cause mysterious errors. The workaround is to use m?...? instead. See http://search.cpan.org/perldoc?Switch#LIMITATIONS

Currently internal buffer lengths are limited to 32-bit only.

22.12 THINGS TO LOOK OUT FOR

Some Filters Clobber the `DATA` Handle

Some source filters use the `DATA` handle to read the calling program. When using these source filters you cannot rely on this handle, nor expect any particular kind of behavior when operating on it. Filters based on Filter::Util::Call (and therefore Filter::Simple) do not alter the `DATA` filehandle.

22.13 REQUIREMENTS

The Source Filters distribution is available on CPAN, in

```
CPAN/modules/by-module/Filter
```

Starting from Perl 5.8 Filter::Util::Call (the core part of the Source Filters distribution) is part of the standard Perl distribution. Also included is a friendlier interface called Filter::Simple, by Damian Conway.

22.14 AUTHOR

Paul Marquess <Paul.Marquess@btinternet.com>

22.15 Copyrights

This article originally appeared in The Perl Journal #11, and is copyright 1998 The Perl Journal. It appears courtesy of Jon Orwant and The Perl Journal. This document may be distributed under the same terms as Perl itself.

23 perlfork

23.1 NAME

perlfork - Perl's fork() emulation

23.2 SYNOPSIS

```
NOTE:  As of the 5.8.0 release, fork() emulation has considerably
matured.  However, there are still a few known bugs and differences
from real fork() that might affect you.  See the "BUGS" and
"CAVEATS AND LIMITATIONS" sections below.
```

Perl provides a fork() keyword that corresponds to the Unix system call of the same name. On most Unix-like platforms where the fork() system call is available, Perl's fork() simply calls it.

On some platforms such as Windows where the fork() system call is not available, Perl can be built to emulate fork() at the interpreter level. While the emulation is designed to be as compatible as possible with the real fork() at the level of the Perl program, there are certain important differences that stem from the fact that all the pseudo child "processes" created this way live in the same real process as far as the operating system is concerned.

This document provides a general overview of the capabilities and limitations of the fork() emulation. Note that the issues discussed here are not applicable to platforms where a real fork() is available and Perl has been configured to use it.

23.3 DESCRIPTION

The fork() emulation is implemented at the level of the Perl interpreter. What this means in general is that running fork() will actually clone the running interpreter and all its state, and run the cloned interpreter in a separate thread, beginning execution in the new thread just after the point where the fork() was called in the parent. We will refer to the thread that implements this child "process" as the pseudo-process.

To the Perl program that called fork(), all this is designed to be transparent. The parent returns from the fork() with a pseudo-process ID that can be subsequently used in any process-manipulation functions; the child returns from the fork() with a value of `0` to signify that it is the child pseudo-process.

23.3.1 Behavior of other Perl features in forked pseudo-processes

Most Perl features behave in a natural way within pseudo-processes.

$$ or $PROCESS_ID

This special variable is correctly set to the pseudo-process ID. It can be used to identify pseudo-processes within a particular session. Note that this value is subject to recycling if any pseudo-processes are launched after others have been wait()-ed on.

%ENV

Each pseudo-process maintains its own virtual environment. Modifications to %ENV affect the virtual environment, and are only visible within that pseudo-process, and in any processes (or pseudo-processes) launched from it.

chdir() and all other builtins that accept filenames

Each pseudo-process maintains its own virtual idea of the current directory. Modifications to the current directory using chdir() are only visible within that pseudo-process, and in any processes (or pseudo-processes) launched from it. All file and directory accesses from the pseudo-process will correctly map the virtual working directory to the real working directory appropriately.

wait() and waitpid()

wait() and waitpid() can be passed a pseudo-process ID returned by fork(). These calls will properly wait for the termination of the pseudo-process and return its status.

kill()

`kill('KILL', ...)` can be used to terminate a pseudo-process by passing it the ID returned by fork(). The outcome of kill on a pseudo-process is unpredictable and it should not be used except under dire circumstances, because the operating system may not guarantee integrity of the process resources when a running thread is terminated. The process which implements the pseudo-processes can be blocked and the Perl interpreter hangs. Note that using `kill('KILL', ...)` on a pseudo-process() may typically cause memory leaks, because the thread that implements the pseudo-process does not get a chance to clean up its resources.

`kill('TERM', ...)` can also be used on pseudo-processes, but the signal will not be delivered while the pseudo-process is blocked by a system call, e.g. waiting for a socket to connect, or trying to read from a socket with no data available. Starting in Perl 5.14 the parent process will not wait for children to exit once they have been signalled with `kill('TERM', ...)` to avoid deadlock during process exit. You will have to explicitly call waitpid() to make sure the child has time to clean-up itself, but you are then also responsible that the child is not blocking on I/O either.

exec()

Calling exec() within a pseudo-process actually spawns the requested executable in a separate process and waits for it to complete before exiting with the same exit status as that process. This means that the process ID reported within the running executable will be different from what the earlier Perl fork() might have returned. Similarly, any process manipulation functions applied to the ID returned by fork() will affect the waiting pseudo-process that called exec(), not the real process it is waiting for after the exec().

When exec() is called inside a pseudo-process then DESTROY methods and END blocks will still be called after the external process returns.

exit()

exit() always exits just the executing pseudo-process, after automatically wait()-ing for any outstanding child pseudo-processes. Note that this means that the process as a whole will not exit unless all running pseudo-processes have exited. See below for some limitations with open filehandles.

Open handles to files, directories and network sockets

All open handles are dup()-ed in pseudo-processes, so that closing any handles in one process does not affect the others. See below for some limitations.

23.3.2 Resource limits

In the eyes of the operating system, pseudo-processes created via the fork() emulation are simply threads in the same process. This means that any process-level limits imposed by the operating system apply to all pseudo-processes taken together. This includes any limits imposed by the operating system on the number of open file, directory and socket handles, limits on disk space usage, limits on memory size, limits on CPU utilization etc.

23.3.3 Killing the parent process

If the parent process is killed (either using Perl's kill() builtin, or using some external means) all the pseudo-processes are killed as well, and the whole process exits.

23.3.4 Lifetime of the parent process and pseudo-processes

During the normal course of events, the parent process and every pseudo-process started by it will wait for their respective pseudo-children to complete before they exit. This means that the parent and every pseudo-child created by it that is also a pseudo-parent will only exit after their pseudo-children have exited.

Starting with Perl 5.14 a parent will not wait() automatically for any child that has been signalled with `kill('TERM', ...)` to avoid a deadlock in case the child is blocking on I/O and never receives the signal.

23.4 CAVEATS AND LIMITATIONS

BEGIN blocks

The fork() emulation will not work entirely correctly when called from within a BEGIN block. The forked copy will run the contents of the BEGIN block, but will not continue parsing the source stream after the BEGIN block. For example, consider the following code:

```
BEGIN {
    fork and exit;          # fork child and exit the parent
    print "inner\n";
}
print "outer\n";
```

This will print:

```
inner
```

rather than the expected:

```
inner
outer
```

This limitation arises from fundamental technical difficulties in cloning and restarting the stacks used by the Perl parser in the middle of a parse.

Open filehandles

Any filehandles open at the time of the fork() will be dup()-ed. Thus, the files can be closed independently in the parent and child, but beware that the dup()-ed handles will still share the same seek pointer. Changing the seek position in the parent will change it in the child and vice-versa. One can avoid this by opening files that need distinct seek pointers separately in the child.

On some operating systems, notably Solaris and Unixware, calling `exit()` from a child process will flush and close open filehandles in the parent, thereby corrupting the filehandles. On these systems, calling `_exit()` is suggested instead. `_exit()` is available in Perl through the `POSIX` module. Please consult your system's manpages for more information on this.

Open directory handles

Perl will completely read from all open directory handles until they reach the end of the stream. It will then seekdir() back to the original location and all future readdir() requests will be fulfilled from the cache buffer. That means that neither the directory handle held by the parent process nor the one held by the child process will see any changes made to the directory after the fork() call.

Note that rewinddir() has a similar limitation on Windows and will not force readdir() to read the directory again either. Only a newly opened directory handle will reflect changes to the directory.

Forking pipe open() not yet implemented

The `open(FOO, "|-")` and `open(BAR, "-|")` constructs are not yet implemented. This limitation can be easily worked around in new code by creating a pipe explicitly. The following example shows how to write to a forked child:

```
# simulate open(FOO, "|-")
sub pipe_to_fork ($) {
    my $parent = shift;
    pipe my $child, $parent or die;
    my $pid = fork();
    die "fork() failed: $!" unless defined $pid;
    if ($pid) {
        close $child;
    }
    else {
        close $parent;
        open(STDIN, "<&=" . fileno($child)) or die;
    }
    $pid;
}

if (pipe_to_fork('FOO')) {
    # parent
    print FOO "pipe_to_fork\n";
    close FOO;
}
else {
    # child
    while (<STDIN>) { print; }
    exit(0);
}
```

And this one reads from the child:

```
    # simulate open(FOO, "-|")
    sub pipe_from_fork ($) {
        my $parent = shift;
        pipe $parent, my $child or die;
        my $pid = fork();
        die "fork() failed: $!" unless defined $pid;
        if ($pid) {
            close $child;
        }
        else {
            close $parent;
            open(STDOUT, ">&=" . fileno($child)) or die;
        }
        $pid;
    }

    if (pipe_from_fork('BAR')) {
        # parent
        while (<BAR>) { print; }
        close BAR;
    }
    else {
        # child
        print "pipe_from_fork\n";
        exit(0);
    }
```

Forking pipe open() constructs will be supported in future.

Global state maintained by XSUBs

External subroutines (XSUBs) that maintain their own global state may not work correctly. Such XSUBs will either need to maintain locks to protect simultaneous access to global data from different pseudo-processes, or maintain all their state on the Perl symbol table, which is copied naturally when fork() is called. A callback mechanism that provides extensions an opportunity to clone their state will be provided in the near future.

Interpreter embedded in larger application

The fork() emulation may not behave as expected when it is executed in an application which embeds a Perl interpreter and calls Perl APIs that can evaluate bits of Perl code. This stems from the fact that the emulation only has knowledge about the Perl interpreter's own data structures and knows nothing about the containing application's state. For example, any state carried on the application's own call stack is out of reach.

Thread-safety of extensions

Since the fork() emulation runs code in multiple threads, extensions calling into non-thread-safe libraries may not work reliably when calling fork(). As Perl's threading support gradually becomes more widely adopted even on platforms with a native fork(), such extensions are expected to be fixed for thread-safety.

23.5 PORTABILITY CAVEATS

In portable Perl code, `kill(9, $child)` must not be used on forked processes. Killing a forked process is unsafe and has unpredictable results. See [kill()], page 338, above.

23.6 BUGS

- Having pseudo-process IDs be negative integers breaks down for the integer `-1` because the wait() and waitpid() functions treat this number as being special. The tacit assumption in the current implementation is that the system never allocates a thread ID of `1` for user threads. A better representation for pseudo-process IDs will be implemented in future.
- In certain cases, the OS-level handles created by the pipe(), socket(), and accept() operators are apparently not duplicated accurately in pseudo-processes. This only happens in some situations, but where it does happen, it may result in deadlocks between the read and write ends of pipe handles, or inability to send or receive data across socket handles.
- This document may be incomplete in some respects.

23.7 AUTHOR

Support for concurrent interpreters and the fork() emulation was implemented by ActiveState, with funding from Microsoft Corporation.

This document is authored and maintained by Gurusamy Sarathy <gsar@activestate.com>.

23.8 SEE ALSO

[perlfunc fork], page 386, Section 36.1 [perlipc NAME], page 667

24 perlform

24.1 NAME

perlform - Perl formats

24.2 DESCRIPTION

Perl has a mechanism to help you generate simple reports and charts. To facilitate this, Perl helps you code up your output page close to how it will look when it's printed. It can keep track of things like how many lines are on a page, what page you're on, when to print page headers, etc. Keywords are borrowed from FORTRAN: format() to declare and write() to execute; see their entries in Section 25.1 [perlfunc NAME], page 351. Fortunately, the layout is much more legible, more like BASIC's PRINT USING statement. Think of it as a poor man's nroff(1).

Formats, like packages and subroutines, are declared rather than executed, so they may occur at any point in your program. (Usually it's best to keep them all together though.) They have their own namespace apart from all the other "types" in Perl. This means that if you have a function named "Foo", it is not the same thing as having a format named "Foo". However, the default name for the format associated with a given filehandle is the same as the name of the filehandle. Thus, the default format for STDOUT is named "STDOUT", and the default format for filehandle TEMP is named "TEMP". They just look the same. They aren't.

Output record formats are declared as follows:

```
format NAME =
FORMLIST
.
```

If the name is omitted, format "STDOUT" is defined. A single "." in column 1 is used to terminate a format. FORMLIST consists of a sequence of lines, each of which may be one of three types:

1. A comment, indicated by putting a '#' in the first column.
2. A "picture" line giving the format for one output line.
3. An argument line supplying values to plug into the previous picture line.

Picture lines contain output field definitions, intermingled with literal text. These lines do not undergo any kind of variable interpolation. Field definitions are made up from a set of characters, for starting and extending a field to its desired width. This is the complete set of characters for field definitions:

```
 >>
```

```
@    start of regular field
^    start of special field
<    pad character for left justification
|    pad character for centering
```

```
>    pad character for right justification
#    pad character for a right-justified numeric field
0    instead of first #: pad number with leading zeroes
.    decimal point within a numeric field
...  terminate a text field, show "..." as truncation evidence
@*   variable width field for a multi-line value
^*   variable width field for next line of a multi-line value
~    suppress line with all fields empty
~~   repeat line until all fields are exhausted
```

Each field in a picture line starts with either "@" (at) or "^" (caret), indicating what we'll call, respectively, a "regular" or "special" field. The choice of pad characters determines whether a field is textual or numeric. The tilde operators are not part of a field. Let's look at the various possibilities in detail.

24.2.1 Text Fields

The length of the field is supplied by padding out the field with multiple "<", ">", or "|" characters to specify a non-numeric field with, respectively, left justification, right justification, or centering. For a regular field, the value (up to the first newline) is taken and printed according to the selected justification, truncating excess characters. If you terminate a text field with "...", three dots will be shown if the value is truncated. A special text field may be used to do rudimentary multi-line text block filling; see Section 24.2.6 [Using Fill Mode], page 345 for details.

```
Example:
   format STDOUT =
   @<<<<<<   @||||||   @>>>>>>
   "left",   "middle", "right"
   .
Output:
   left      middle    right
```

24.2.2 Numeric Fields

Using "#" as a padding character specifies a numeric field, with right justification. An optional "." defines the position of the decimal point. With a "0" (zero) instead of the first "#", the formatted number will be padded with leading zeroes if necessary. A special numeric field is blanked out if the value is undefined. If the resulting value would exceed the width specified the field is filled with "#" as overflow evidence.

```
Example:
   format STDOUT =
   @###   @.###   @##.###  @###   @###   ^####
    42,   3.1415,  undef,    0, 10000,   undef
   .
Output:
     42   3.142     0.000     0   ####
```

24.2.3 The Field @* for Variable-Width Multi-Line Text

The field "@*" can be used for printing multi-line, nontruncated values; it should (but need not) appear by itself on a line. A final line feed is chomped off, but all other characters are emitted verbatim.

24.2.4 The Field ^* for Variable-Width One-line-at-a-time Text

Like "@*", this is a variable-width field. The value supplied must be a scalar variable. Perl puts the first line (up to the first "\n") of the text into the field, and then chops off the front of the string so that the next time the variable is referenced, more of the text can be printed. The variable will *not* be restored.

```
Example:
   $text = "line 1\nline 2\nline 3";
   format STDOUT =
   Text: ^*
         $text
   ~~    ^*
         $text
   .
Output:
   Text: line 1
         line 2
         line 3
```

24.2.5 Specifying Values

The values are specified on the following format line in the same order as the picture fields. The expressions providing the values must be separated by commas. They are all evaluated in a list context before the line is processed, so a single list expression could produce multiple list elements. The expressions may be spread out to more than one line if enclosed in braces. If so, the opening brace must be the first token on the first line. If an expression evaluates to a number with a decimal part, and if the corresponding picture specifies that the decimal part should appear in the output (that is, any picture except multiple "#" characters **without** an embedded "."), the character used for the decimal point is determined by the current LC_NUMERIC locale if `use locale` is in effect. This means that, if, for example, the run-time environment happens to specify a German locale, "," will be used instead of the default ".". See Section 38.1 [perllocale NAME], page 701 and Section 24.4 [WARNINGS], page 350 for more information.

24.2.6 Using Fill Mode

On text fields the caret enables a kind of fill mode. Instead of an arbitrary expression, the value supplied must be a scalar variable that contains a text string. Perl puts the next portion of the text into the field, and then chops off the front of the string so that the next time the variable is referenced, more of the text can be printed. (Yes, this means that the variable itself is altered during execution of the write() call, and is not restored.) The next portion of text is determined by a crude line-breaking algorithm. You may use the carriage return character (`\r`) to force a line break. You can change which characters are legal to break on by changing the variable `$:` (that's $FORMAT_LINE_BREAK_CHARACTERS if you're using the English module) to a list of the desired characters.

Normally you would use a sequence of fields in a vertical stack associated with the same scalar variable to print out a block of text. You might wish to end the final field with the text "...", which will appear in the output if the text was too long to appear in its entirety.

24.2.7 Suppressing Lines Where All Fields Are Void

Using caret fields can produce lines where all fields are blank. You can suppress such lines by putting a "~" (tilde) character anywhere in the line. The tilde will be translated to a space upon output.

24.2.8 Repeating Format Lines

If you put two contiguous tilde characters "~~" anywhere into a line, the line will be repeated until all the fields on the line are exhausted, i.e. undefined. For special (caret) text fields this will occur sooner or later, but if you use a text field of the at variety, the expression you supply had better not give the same value every time forever! (`shift(@f)` is a simple example that would work.) Don't use a regular (at) numeric field in such lines, because it will never go blank.

24.2.9 Top of Form Processing

Top-of-form processing is by default handled by a format with the same name as the current filehandle with "_TOP" concatenated to it. It's triggered at the top of each page. See [perlfunc write], page 489.

Examples:

```
# a report on the /etc/passwd file
format STDOUT_TOP =
                        Passwd File
Name                Login    Office   Uid   Gid Home
------------------------------------------------------------------
.
format STDOUT =
@<<<<<<<<<<<<<<<<<< @||||||| @<<<<<<@>>>> @>>>> @<<<<<<<<<<<<<<<<<
$name,              $login,  $office,$uid,$gid, $home
.


# a report from a bug report form
format STDOUT_TOP =
                        Bug Reports
@<<<<<<<<<<<<<<<<<<<<<<<     @|||         @>>>>>>>>>>>>>>>>>>>>>>>
$system,                      $%,         $date
------------------------------------------------------------------
.
format STDOUT =
Subject: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
         $subject
```

```
 Index: @<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
        $index,                       $description
 Priority: @<<<<<<<<<< Date: @<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
           $priority,        $date,   $description
 From: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
       $from,                         $description
 Assigned to: @<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
              $programmer,            $description
 ~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                      $description
 ~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                      $description
 ~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                      $description
 ~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                      $description
 ~                                    ^<<<<<<<<<<<<<<<<<<<<<<<...
                                      $description
 .
```

It is possible to intermix print()s with write()s on the same output channel, but you'll have to handle `$-` (`$FORMAT_LINES_LEFT`) yourself.

24.2.10 Format Variables

The current format name is stored in the variable `$~` (`$FORMAT_NAME`), and the current top of form format name is in `$^` (`$FORMAT_TOP_NAME`). The current output page number is stored in `$%` (`$FORMAT_PAGE_NUMBER`), and the number of lines on the page is in `$=` (`$FORMAT_LINES_PER_PAGE`). Whether to autoflush output on this handle is stored in `$|` (`$OUTPUT_AUTOFLUSH`). The string output before each top of page (except the first) is stored in `$^L` (`$FORMAT_FORMFEED`). These variables are set on a per-filehandle basis, so you'll need to select() into a different one to affect them:

```
select((select(OUTF),
        $~ = "My_Other_Format",
        $^ = "My_Top_Format"
       )[0]);
```

Pretty ugly, eh? It's a common idiom though, so don't be too surprised when you see it. You can at least use a temporary variable to hold the previous filehandle: (this is a much better approach in general, because not only does legibility improve, you now have an intermediary stage in the expression to single-step the debugger through):

```
$ofh = select(OUTF);
$~ = "My_Other_Format";
$^ = "My_Top_Format";
select($ofh);
```

If you use the English module, you can even read the variable names:

```
use English;
$ofh = select(OUTF);
```

```
$FORMAT_NAME     = "My_Other_Format";
$FORMAT_TOP_NAME = "My_Top_Format";
select($ofh);
```

But you still have those funny select()s. So just use the FileHandle module. Now, you can access these special variables using lowercase method names instead:

```
use FileHandle;
format_name     OUTF "My_Other_Format";
format_top_name OUTF "My_Top_Format";
```

Much better!

24.3 NOTES

Because the values line may contain arbitrary expressions (for at fields, not caret fields), you can farm out more sophisticated processing to other functions, like sprintf() or one of your own. For example:

```
format Ident =
    @<<<<<<<<<<<<<<<
    &commify($n)
.
```

To get a real at or caret into the field, do this:

```
format Ident =
I have an @ here.
        "@"
.
```

To center a whole line of text, do something like this:

```
format Ident =
@|||||||||||||||||||||||||||||||||||||||||||||||
        "Some text line"
.
```

There is no builtin way to say "float this to the right hand side of the page, however wide it is." You have to specify where it goes. The truly desperate can generate their own format on the fly, based on the current number of columns, and then eval() it:

```
$format  = "format STDOUT = \n"
         . ’^’ . ’<’ x $cols . "\n"
         . ’$entry’ . "\n"
         . "\t^" . "<" x ($cols-8) . "~~\n"
         . ’$entry’ . "\n"
         . ".\n";
print $format if $Debugging;
eval $format;
die $@ if $@;
```

Which would generate a format looking something like this:

```
format STDOUT =
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

```
$entry
        ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<~~
$entry
.
```

Here's a little program that's somewhat like fmt(1):

```
format =
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ~~
$_

.

$/ = '';
while (<>) {
    s/\s*\n\s*/ /g;
    write;
}
```

24.3.1 Footers

While $FORMAT_TOP_NAME contains the name of the current header format, there is no corresponding mechanism to automatically do the same thing for a footer. Not knowing how big a format is going to be until you evaluate it is one of the major problems. It's on the TODO list.

Here's one strategy: If you have a fixed-size footer, you can get footers by checking $FORMAT_LINES_LEFT before each write() and print the footer yourself if necessary.

Here's another strategy: Open a pipe to yourself, using `open(MYSELF, "|-")` (see ⟨undefined⟩ [perlfunc open], page ⟨undefined⟩) and always write() to MYSELF instead of STDOUT. Have your child process massage its STDIN to rearrange headers and footers however you like. Not very convenient, but doable.

24.3.2 Accessing Formatting Internals

For low-level access to the formatting mechanism, you may use formline() and access `$^A` (the $ACCUMULATOR variable) directly.

For example:

```
$str = formline <<'END', 1,2,3;
@<<<  @|||  @>>>
END

print "Wow, I just stored '$^A' in the accumulator!\n";
```

Or to make an swrite() subroutine, which is to write() what sprintf() is to printf(), do this:

```
use Carp;
sub swrite {
    croak "usage: swrite PICTURE ARGS" unless @_;
    my $format = shift;
    $^A = "";
```

```
        formline($format,@_);
        return $^A;
    }

    $string = swrite(<<'END', 1, 2, 3);
 Check me out
 @<<<  @|||  @>>>
 END
    print $string;
```

24.4 WARNINGS

The lone dot that ends a format can also prematurely end a mail message passing through a misconfigured Internet mailer (and based on experience, such misconfiguration is the rule, not the exception). So when sending format code through mail, you should indent it so that the format-ending dot is not on the left margin; this will prevent SMTP cutoff.

Lexical variables (declared with "my") are not visible within a format unless the format is declared within the scope of the lexical variable.

If a program's environment specifies an LC_NUMERIC locale and `use locale` is in effect when the format is declared, the locale is used to specify the decimal point character in formatted output. Formatted output cannot be controlled by `use locale` at the time when write() is called. See Section 38.1 [perllocale NAME], page 701 for further discussion of locale handling.

Within strings that are to be displayed in a fixed-length text field, each control character is substituted by a space. (But remember the special meaning of `\r` when using fill mode.) This is done to avoid misalignment when control characters "disappear" on some output media.

25 perlfunc

25.1 NAME

perlfunc - Perl builtin functions

25.2 DESCRIPTION

The functions in this section can serve as terms in an expression. They fall into two major categories: list operators and named unary operators. These differ in their precedence relationship with a following comma. (See the precedence table in Section 48.1 [perlop NAME], page 798.) List operators take more than one argument, while unary operators can never take more than one argument. Thus, a comma terminates the argument of a unary operator, but merely separates the arguments of a list operator. A unary operator generally provides scalar context to its argument, while a list operator may provide either scalar or list contexts for its arguments. If it does both, scalar arguments come first and list argument follow, and there can only ever be one such list argument. For instance, splice() has three scalar arguments followed by a list, whereas gethostbyname() has four scalar arguments.

In the syntax descriptions that follow, list operators that expect a list (and provide list context for elements of the list) are shown with LIST as an argument. Such a list may consist of any combination of scalar arguments or list values; the list values will be included in the list as if each individual element were interpolated at that point in the list, forming a longer single-dimensional list value. Commas should separate literal elements of the LIST.

Any function in the list below may be used either with or without parentheses around its arguments. (The syntax descriptions omit the parentheses.) If you use parentheses, the simple but occasionally surprising rule is this: It *looks* like a function, therefore it *is* a function, and precedence doesn't matter. Otherwise it's a list operator or unary operator, and precedence does matter. Whitespace between the function and left parenthesis doesn't count, so sometimes you need to be careful:

```
print 1+2+4;      # Prints 7.
print(1+2) + 4;   # Prints 3.
print (1+2)+4;    # Also prints 3!
print +(1+2)+4;   # Prints 7.
print ((1+2)+4);  # Prints 7.
```

If you run Perl with the **-w** switch it can warn you about this. For example, the third line above produces:

```
print (...) interpreted as function at - line 1.
Useless use of integer addition in void context at - line 1.
```

A few functions take no arguments at all, and therefore work as neither unary nor list operators. These include such functions as `time` and `endpwent`. For example, `time+86_400` always means `time() + 86_400`.

For functions that can be used in either a scalar or list context, nonabortive failure is generally indicated in scalar context by returning the undefined value, and in list context by returning the empty list.

Remember the following important rule: There is **no rule** that relates the behavior of an expression in list context to its behavior in scalar context, or vice versa. It might do two totally different things. Each operator and function decides which sort of value would be most appropriate to return in scalar context. Some operators return the length of the list that would have been returned in list context. Some operators return the first value in the list. Some operators return the last value in the list. Some operators return a count of successful operations. In general, they do what you want, unless you want consistency.

A named array in scalar context is quite different from what would at first glance appear to be a list in scalar context. You can't get a list like `(1,2,3)` into being in scalar context, because the compiler knows the context at compile time. It would generate the scalar comma operator there, not the list construction version of the comma. That means it was never a list to start with.

In general, functions in Perl that serve as wrappers for system calls ("syscalls") of the same name (like chown(2), fork(2), closedir(2), etc.) return true when they succeed and `undef` otherwise, as is usually mentioned in the descriptions below. This is different from the C interfaces, which return `-1` on failure. Exceptions to this rule include `wait`, `waitpid`, and `syscall`. System calls also set the special `$!` variable on failure. Other functions do not, except accidentally.

Extension modules can also hook into the Perl parser to define new kinds of keyword-headed expression. These may look like functions, but may also look completely different. The syntax following the keyword is defined entirely by the extension. If you are an implementor, see Section "PL_keyword_plugin" in `perlapi` for the mechanism. If you are using such a module, see the module's documentation for details of the syntax that it defines.

25.2.1 Perl Functions by Category

Here are Perl's functions (including things that look like functions, like some keywords and named operators) arranged by category. Some functions appear in more than one place.

Functions for SCALARs or strings

`chomp`, `chop`, `chr`, `crypt`, `fc`, `hex`, `index`, `lc`, `lcfirst`, `length`, `oct`, `ord`, `pack`, `q//`, `qq//`, `reverse`, `rindex`, `sprintf`, `substr`, `tr///`, `uc`, `ucfirst`, `y///`

`fc` is available only if the `"fc"` feature is enabled or if it is prefixed with `CORE::`. The `"fc"` feature is enabled automatically with a `use v5.16` (or higher) declaration in the current scope.

Regular expressions and pattern matching

`m//`, `pos`, `qr//`, `quotemeta`, `s///`, `split`, `study`

Numeric functions

`abs`, `atan2`, `cos`, `exp`, `hex`, `int`, `log`, `oct`, `rand`, `sin`, `sqrt`, `srand`

Functions for real @ARRAYs

`each`, `keys`, `pop`, `push`, `shift`, `splice`, `unshift`, `values`

Functions for list data

`grep`, `join`, `map`, `qw//`, `reverse`, `sort`, `unpack`

Functions for real %HASHes

`delete`, `each`, `exists`, `keys`, `values`

Input and output functions

`binmode`, `close`, `closedir`, `dbmclose`, `dbmopen`, `die`, `eof`, `fileno`, `flock`, `format`, `getc`, `print`, `printf`, `read`, `readdir`, `readline` `rewinddir`, `say`, `seek`, `seekdir`, `select`, `syscall`, `sysread`, `sysseek`, `syswrite`, `tell`, `telldir`, `truncate`, `warn`, `write`

`say` is available only if the `"say"` feature is enabled or if it is prefixed with `CORE::`. The `"say"` feature is enabled automatically with a `use v5.10` (or higher) declaration in the current scope.

Functions for fixed-length data or records

`pack`, `read`, `syscall`, `sysread`, `sysseek`, `syswrite`, `unpack`, `vec`

Functions for filehandles, files, or directories

`-`*`X`*, `chdir`, `chmod`, `chown`, `chroot`, `fcntl`, `glob`, `ioctl`, `link`, `lstat`, `mkdir`, `open`, `opendir`, `readlink`, `rename`, `rmdir`, `stat`, `symlink`, `sysopen`, `umask`, `unlink`, `utime`

Keywords related to the control flow of your Perl program

`break`, `caller`, `continue`, `die`, `do`, `dump`, `eval`, `evalbytes` `exit`, `__FILE__` , `goto`, `last`, `__LINE__`, `next`, `__PACKAGE__`, `redo`, `return`, `sub`, `__SUB__`, `wantarray`

`break` is available only if you enable the experimental `"switch"` feature or use the `CORE::` prefix. The `"switch"` feature also enables the `default`, `given` and `when` statements, which are documented in Section 74.2.11 [perlsyn Switch Statements], page 1258. The `"switch"` feature is enabled automatically with a `use v5.10` (or higher) declaration in the current scope. In Perl v5.14 and earlier, `continue` required the `"switch"` feature, like the other keywords.

`evalbytes` is only available with the `"evalbytes"` feature (see `feature`) or if prefixed with `CORE::`. `__SUB__` is only available with the `"current_sub"` feature or if prefixed with `CORE::`. Both the `"evalbytes"` and `"current_sub"` features are enabled automatically with a `use v5.16` (or higher) declaration in the current scope.

Keywords related to scoping

`caller`, `import`, `local`, `my`, `our`, `package`, `state`, `use`

`state` is available only if the `"state"` feature is enabled or if it is prefixed with `CORE::`. The `"state"` feature is enabled automatically with a `use v5.10` (or higher) declaration in the current scope.

Miscellaneous functions

`defined`, `formline`, `lock`, `prototype`, `reset`, `scalar`, `undef`

Functions for processes and process groups

`alarm`, `exec`, `fork`, `getpgrp`, `getppid`, `getpriority`, `kill`, `pipe`, `qx//`, `readpipe`, `setpgrp`, `setpriority`, `sleep`, `system`, `times`, `wait`, `waitpid`

Keywords related to Perl modules

`do`, `import`, `no`, `package`, `require`, `use`

Keywords related to classes and object-orientation

`bless`, `dbmclose`, `dbmopen`, `package`, `ref`, `tie`, `tied`, `untie`, `use`

Low-level socket functions

accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, send, setsockopt, shutdown, socket, socketpair

System V interprocess communication functions

msgctl, msgget, msgrcv, msgsnd, semctl, semget, semop, shmctl, shmget, shmread, shmwrite

Fetching user and group info

endgrent, endhostent, endnetent, endpwent, getgrent, getgrgid, getgrnam, getlogin, getpwent, getpwnam, getpwuid, setgrent, setpwent

Fetching network info

endprotoent, endservent, gethostbyaddr, gethostbyname, gethostent, getnetbyaddr, getnetbyname, getnetent, getprotobyname, getprotobynumber, getprotoent, getservbyname, getservbyport, getservent, sethostent, setnetent, setprotoent, setservent

Time-related functions

gmtime, localtime, time, times

Non-function keywords

and, AUTOLOAD, BEGIN, CHECK, cmp, CORE, __DATA__, default, DESTROY, else, elseif, elsif, END, __END__, eq, for, foreach, ge, given, gt, if, INIT, le, lt, ne, not, or, UNITCHECK, unless, until, when, while, x, xor

25.2.2 Portability

Perl was born in Unix and can therefore access all common Unix system calls. In non-Unix environments, the functionality of some Unix system calls may not be available or details of the available functionality may differ slightly. The Perl functions affected by this are:

-X, binmode, chmod, chown, chroot, crypt, dbmclose, dbmopen, dump, endgrent, endhostent, endnetent, endprotoent, endpwent, endservent, exec, fcntl, flock, fork, getgrent, getgrgid, gethostbyname, gethostent, getlogin, getnetbyaddr, getnetbyname, getnetent, getppid, getpgrp, getpriority, getprotobynumber, getprotoent, getpwent, getpwnam, getpwuid, getservbyport, getservent, getsockopt, glob, ioctl, kill, link, lstat, msgctl, msgget, msgrcv, msgsnd, open, pipe, readlink, rename, select, semctl, semget, semop, setgrent, sethostent, setnetent, setpgrp, setpriority, setprotoent, setpwent, setservent, setsockopt, shmctl, shmget, shmread, shmwrite, socket, socketpair, stat, symlink, syscall, sysopen, system, times, truncate, umask, unlink, utime, wait, waitpid

For more information about the portability of these functions, see Section 56.1 [perlport NAME], page 951 and other available platform-specific documentation.

25.2.3 Alphabetical Listing of Perl Functions

-X FILEHANDLE
-X EXPR

-X DIRHANDLE
-X

A file test, where X is one of the letters listed below. This unary operator takes one argument, either a filename, a filehandle, or a dirhandle, and tests the associated file to see if something is true about it. If the argument is omitted, tests `$_`, except for `-t`, which tests STDIN. Unless otherwise documented, it returns `1` for true and `''` for false. If the file doesn't exist or can't be examined, it returns `undef` and sets `$!` (errno). Despite the funny names, precedence is the same as any other named unary operator. The operator may be any of:

```
-r  File is readable by effective uid/gid.
-w  File is writable by effective uid/gid.
-x  File is executable by effective uid/gid.
-o  File is owned by effective uid.

-R  File is readable by real uid/gid.
-W  File is writable by real uid/gid.
-X  File is executable by real uid/gid.
-O  File is owned by real uid.

-e  File exists.
-z  File has zero size (is empty).
-s  File has nonzero size (returns size in bytes).

-f  File is a plain file.
-d  File is a directory.
-l  File is a symbolic link (false if symlinks aren't
    supported by the file system).
-p  File is a named pipe (FIFO), or Filehandle is a pipe.
-S  File is a socket.
-b  File is a block special file.
-c  File is a character special file.
-t  Filehandle is opened to a tty.

-u  File has setuid bit set.
-g  File has setgid bit set.
-k  File has sticky bit set.

-T  File is an ASCII or UTF-8 text file (heuristic guess).
-B  File is a "binary" file (opposite of -T).

-M  Script start time minus file modification time, in days.
-A  Same for access time.
-C  Same for inode change time (Unix, may differ for other
    platforms)
```

Example:

```
while (<>) {
    chomp;
    next unless -f $_;  # ignore specials
```

```
        #...
    }
```

Note that -s/a/b/ does not do a negated substitution. Saying -exp($foo) still works as expected, however: only single letters following a minus are interpreted as file tests.

These operators are exempt from the "looks like a function rule" described above. That is, an opening parenthesis after the operator does not affect how much of the following code constitutes the argument. Put the opening parentheses before the operator to separate it from code that follows (this applies only to operators with higher precedence than unary operators, of course):

```
    -s($file) + 1024   # probably wrong; same as -s($file + 1024)
    (-s $file) + 1024  # correct
```

The interpretation of the file permission operators -r, -R, -w, -W, -x, and -X is by default based solely on the mode of the file and the uids and gids of the user. There may be other reasons you can't actually read, write, or execute the file: for example network filesystem access controls, ACLs (access control lists), read-only filesystems, and unrecognized executable formats. Note that the use of these six specific operators to verify if some operation is possible is usually a mistake, because it may be open to race conditions.

Also note that, for the superuser on the local filesystems, the -r, -R, -w, and -W tests always return 1, and -x and -X return 1 if any execute bit is set in the mode. Scripts run by the superuser may thus need to do a stat() to determine the actual mode of the file, or temporarily set their effective uid to something else.

If you are using ACLs, there is a pragma called filetest that may produce more accurate results than the bare stat() mode bits. When under use filetest 'access' the above-mentioned filetests test whether the permission can(not) be granted using the access(2) family of system calls. Also note that the -x and -X may under this pragma return true even if there are no execute permission bits set (nor any extra execute permission ACLs). This strangeness is due to the underlying system calls' definitions. Note also that, due to the implementation of use filetest 'access', the _ special filehandle won't cache the results of the file tests when this pragma is in effect. Read the documentation for the filetest pragma for more information.

The -T and -B switches work as follows. The first block or so of the file is examined to see if it is valid UTF-8 that includes non-ASCII characters. If, so it's a -T file. Otherwise, that same portion of the file is examined for odd characters such as strange control codes or characters with the high bit set. If more than a third of the characters are strange, it's a -B file; otherwise it's a -T file. Also, any file containing a zero byte in the examined portion is considered a binary file. (If executed within the scope of a Section 38.1 [use locale], page 701 which includes LC_CTYPE, odd characters are anything that isn't a printable nor space in the current locale.) If -T or -B is used on a filehandle, the current IO buffer is examined rather than the first block. Both -T and -B return true on an empty file, or a file at EOF when testing a filehandle. Because you have to read a file to do the `-T` test, on most occasions you want to use a `-f` against the file first, as in `next unless -f $file && -T $file`.

If any of the file tests (or either the `stat` or `lstat` operator) is given the special filehandle consisting of a solitary underline, then the stat structure of the previous file test (or stat operator) is used, saving a system call. (This doesn't work with `-t`, and you need to remember that lstat() and `-l` leave values in the stat structure for the symbolic link, not the real file.) (Also, if the stat buffer was filled by an `lstat` call, `-T` and `-B` will reset it with the results of `stat _`). Example:

```
print "Can do.\n" if -r $a || -w _ || -x _;

stat($filename);
print "Readable\n" if -r _;
print "Writable\n" if -w _;
print "Executable\n" if -x _;
print "Setuid\n" if -u _;
print "Setgid\n" if -g _;
print "Sticky\n" if -k _;
print "Text\n" if -T _;
print "Binary\n" if -B _;
```

As of Perl 5.10.0, as a form of purely syntactic sugar, you can stack file test operators, in a way that `-f -w -x $file` is equivalent to `-x $file && -w _ && -f _`. (This is only fancy syntax: if you use the return value of `-f $file` as an argument to another filetest operator, no special magic will happen.)

Portability issues: [perlport -X], page 972.

To avoid confusing would-be users of your code with mysterious syntax errors, put something like this at the top of your script:

```
use 5.010;  # so filetest ops can stack
```

abs VALUE
abs

Returns the absolute value of its argument. If VALUE is omitted, uses `$_`.

accept NEWSOCKET,GENERICSOCKET

Accepts an incoming socket connect, just as accept(2) does. Returns the packed address if it succeeded, false otherwise. See the example in Section 36.6 [perlipc Sockets: Client/Server Communication], page 682.

On systems that support a close-on-exec flag on files, the flag will be set for the newly opened file descriptor, as determined by the value of $^F. See [perlvar $^F], page 1381.

alarm SECONDS
alarm

Arranges to have a SIGALRM delivered to this process after the specified number of wallclock seconds has elapsed. If SECONDS is not specified, the value stored in `$_` is used. (On some machines, unfortunately, the elapsed time may be up to one second less or more than you specified because of how seconds are counted, and process scheduling may delay the delivery of the signal even further.)

Only one timer may be counting at once. Each call disables the previous timer, and an argument of 0 may be supplied to cancel the previous timer without starting a new one. The returned value is the amount of time remaining on the previous timer.

For delays of finer granularity than one second, the Time::HiRes module (from CPAN, and starting from Perl 5.8 part of the standard distribution) provides ualarm(). You may also use Perl's four-argument version of select() leaving the first three arguments undefined, or you might be able to use the `syscall` interface to access setitimer(2) if your system supports it. See `perlfaq8` for details.

It is usually a mistake to intermix `alarm` and `sleep` calls, because `sleep` may be internally implemented on your system with `alarm`.

If you want to use `alarm` to time out a system call you need to use an `eval`/`die` pair. You can't rely on the alarm causing the system call to fail with `$!` set to `EINTR` because Perl sets up signal handlers to restart system calls on some systems. Using `eval`/`die` always works, modulo the caveats given in Section 36.3 [perlipc Signals], page 667.

```
eval {
    local $SIG{ALRM} = sub { die "alarm\n" }; # NB: \n required
    alarm $timeout;
    $nread = sysread SOCKET, $buffer, $size;
    alarm 0;
};
if ($@) {
    die unless $@ eq "alarm\n";   # propagate unexpected errors
    # timed out
}
else {
    # didn't
}
```

For more information see Section 36.1 [perlipc NAME], page 667.

Portability issues: [perlport alarm], page 972.

atan2 Y,X

Returns the arctangent of Y/X in the range -PI to PI.

For the tangent operation, you may use the `Math::Trig::tan` function, or use the familiar relation:

```
sub tan { sin($_[0]) / cos($_[0])  }
```

The return value for `atan2(0,0)` is implementation-defined; consult your atan2(3) manpage for more information.

Portability issues: [perlport atan2], page 972.

bind SOCKET,NAME

Binds a network address to a socket, just as bind(2) does. Returns true if it succeeded, false otherwise. NAME should be a packed address of the appropriate type for the socket. See the examples in Section 36.6 [perlipc Sockets: Client/Server Communication], page 682.

binmode FILEHANDLE, LAYER
binmode FILEHANDLE

Arranges for FILEHANDLE to be read or written in "binary" or "text" mode on systems where the run-time libraries distinguish between binary and text files. If FILEHANDLE is an expression, the value is taken as the name of the filehandle. Returns true on success, otherwise it returns `undef` and sets `$!` (errno).

On some systems (in general, DOS- and Windows-based systems) binmode() is necessary when you're not working with a text file. For the sake of portability it is a good idea always to use it when appropriate, and never to use it when it isn't appropriate. Also, people can set their I/O to be by default UTF8-encoded Unicode, not bytes.

In other words: regardless of platform, use binmode() on binary data, like images, for example.

If LAYER is present it is a single string, but may contain multiple directives. The directives alter the behaviour of the filehandle. When LAYER is present, using binmode on a text file makes sense.

If LAYER is omitted or specified as `:raw` the filehandle is made suitable for passing binary data. This includes turning off possible CRLF translation and marking it as bytes (as opposed to Unicode characters). Note that, despite what may be implied in "*Programming Perl*" (the Camel, 3rd edition) or elsewhere, `:raw` is *not* simply the inverse of `:crlf`. Other layers that would affect the binary nature of the stream are *also* disabled. See `PerlIO`, Section 69.1 [perlrun NAME], page 1176, and the discussion about the PERLIO environment variable.

The `:bytes`, `:crlf`, `:utf8`, and any other directives of the form `:...`, are called I/O *layers*. The `open` pragma can be used to establish default I/O layers. See `open`.

The LAYER parameter of the binmode() function is described as "DISCIPLINE" in "Programming Perl, 3rd Edition". However, since the publishing of this book, by many known as "Camel III", the consensus of the naming of this functionality has moved from "discipline" to "layer". All documentation of this version of Perl therefore refers to "layers" rather than to "disciplines". Now back to the regularly scheduled documentation...

To mark FILEHANDLE as UTF-8, use `:utf8` or `:encoding(UTF-8)`. `:utf8` just marks the data as UTF-8 without further checking, while `:encoding(UTF-8)` checks the data for actually being valid UTF-8. More details can be found in `PerlIO-encoding`.

In general, binmode() should be called after open() but before any I/O is done on the filehandle. Calling binmode() normally flushes any pending buffered output data (and perhaps pending input data) on the handle. An exception to this is the `:encoding` layer that changes the default character encoding of the handle; see ⟨undefined⟩ [open], page ⟨undefined⟩. The `:encoding` layer sometimes needs to be called in mid-stream, and it doesn't flush the stream. The `:encoding` also implicitly pushes on top of itself the `:utf8` layer because internally Perl operates on UTF8-encoded Unicode characters.

The operating system, device drivers, C libraries, and Perl run-time system all conspire to let the programmer treat a single character (`\n`) as the line terminator, irrespective of external representation. On many operating systems, the native text file representation matches the internal representation, but on some platforms the external representation of `\n` is made up of more than one character.

All variants of Unix, Mac OS (old and new), and Stream_LF files on VMS use a single character to end each line in the external representation of text (even though that single character is CARRIAGE RETURN on old, pre-Darwin flavors of Mac OS, and is LINE FEED on Unix and most VMS files). In other systems like OS/2, DOS, and the various flavors of MS-Windows, your program sees a `\n` as a simple `\cJ`, but what's stored in text files are the two characters `\cM\cJ`. That means that if you don't use binmode() on these systems, `\cM\cJ` sequences on disk will be converted to `\n` on input, and any `\n` in your program will be converted back to `\cM\cJ` on output. This is what you want for text files, but it can be disastrous for binary files.

Another consequence of using binmode() (on some systems) is that special end-of-file markers will be seen as part of the data stream. For systems from the Microsoft family this means that, if your binary data contain `\cZ`, the I/O subsystem will regard it as the end of the file, unless you use binmode().

binmode() is important not only for readline() and print() operations, but also when using read(), seek(), sysread(), syswrite() and tell() (see Section 56.1 [perlport NAME], page 951 for more details). See the `$/` and `$\` variables in Section 86.1 [perlvar NAME], page 1375 for how to manually set your input and output line-termination sequences.

Portability issues: [perlport binmode], page 972.

bless REF,CLASSNAME
bless REF

This function tells the thingy referenced by REF that it is now an object in the CLASSNAME package. If CLASSNAME is omitted, the current package is used. Because a `bless` is often the last thing in a constructor, it returns the reference for convenience. Always use the two-argument version if a derived class might inherit the function doing the blessing. See Section 46.1 [perlobj NAME], page 769 for more about the blessing (and blessings) of objects.

Consider always blessing objects in CLASSNAMEs that are mixed case. Namespaces with all lowercase names are considered reserved for Perl pragmata. Builtin types have all uppercase names. To prevent confusion, you may wish to avoid such package names as well. Make sure that CLASSNAME is a true value.

See Section 40.2.6 [perlmod Perl Modules], page 738.

break

Break out of a `given()` block.

This keyword is enabled by the `"switch"` feature; see `feature` for more information on `"switch"`. You can also access it by prefixing it with `CORE::`. Alternatively, include a `use v5.10` or later to the current scope.

caller EXPR
caller

Returns the context of the current pure perl subroutine call. In scalar context, returns the caller's package name if there *is* a caller (that is, if we're in a subroutine or `eval` or `require`) and the undefined value otherwise. caller never returns XS subs and they are skipped. The next pure perl sub will appear instead of the XS sub in caller's return values. In list context, caller returns

```
# 0         1          2
($package, $filename, $line) = caller;
```

With EXPR, it returns some extra information that the debugger uses to print a stack trace. The value of EXPR indicates how many call frames to go back before the current one.

```
#  0         1          2      3            4
($package, $filename, $line, $subroutine, $hasargs,

#  5          6          7            8       9         10
$wantarray, $evaltext, $is_require, $hints, $bitmask, $hinthash)
 = caller($i);
```

Here, $subroutine is the function that the caller called (rather than the function containing the caller). Note that $subroutine may be `(eval)` if the frame is not a subroutine call, but an `eval`. In such a case additional elements $evaltext and `$is_require` are set: `$is_require` is true if the frame is created by a `require` or `use` statement, $evaltext contains the text of the `eval EXPR` statement. In particular, for an `eval BLOCK` statement, $subroutine is `(eval)`, but $evaltext is undefined. (Note also that each `use` statement creates a `require` frame inside an `eval EXPR` frame.) $subroutine may also be `(unknown)` if this particular subroutine happens to have been deleted from the symbol table. `$hasargs` is true if a new instance of `@_` was set up for the frame. `$hints` and `$bitmask` contain pragmatic hints that the caller was compiled with. `$hints` corresponds to `$^H`, and `$bitmask` corresponds to `${^WARNING_BITS}`. The `$hints` and `$bitmask` values are subject to change between versions of Perl, and are not meant for external use.

`$hinthash` is a reference to a hash containing the value of `%^H` when the caller was compiled, or `undef` if `%^H` was empty. Do not modify the values of this hash, as they are the actual values stored in the optree.

Furthermore, when called from within the DB package in list context, and with an argument, caller returns more detailed information: it sets the list variable `@DB::args` to be the arguments with which the subroutine was invoked.

Be aware that the optimizer might have optimized call frames away before `caller` had a chance to get the information. That means that `caller(N)` might not return information about the call frame you expect it to, for `N > 1`. In particular, `@DB::args` might have information from the previous time `caller` was called.

Be aware that setting `@DB::args` is *best effort*, intended for debugging or generating backtraces, and should not be relied upon. In particular, as `@_` contains aliases to the caller's arguments, Perl does not take a copy of `@_`, so `@DB::args` will contain modifications the subroutine makes to `@_` or its contents, not the original values at call time. `@DB::args`, like `@_`, does not hold explicit references to its elements, so under certain cases its elements may have become freed and reallocated for other variables or temporary values. Finally, a side effect of the current implementation is that the effects of `shift @_` can *normally* be undone (but not `pop @_` or other splicing, *and* not if a reference to `@_` has been taken, *and* subject to the caveat about reallocated elements), so `@DB::args` is actually a hybrid of the current state and initial state of `@_`. Buyer beware.

chdir EXPR
chdir FILEHANDLE
chdir DIRHANDLE
chdir

Changes the working directory to EXPR, if possible. If EXPR is omitted, changes to the directory specified by `$ENV{HOME}`, if set; if not, changes to the directory specified by `$ENV{LOGDIR}`. (Under VMS, the variable `$ENV{SYS$LOGIN}` is also checked, and used if it is set.) If neither is set, `chdir` does nothing. It returns true on success, false otherwise. See the example under `die`.

On systems that support fchdir(2), you may pass a filehandle or directory handle as the argument. On systems that don't support fchdir(2), passing handles raises an exception.

chmod LIST

Changes the permissions of a list of files. The first element of the list must be the numeric mode, which should probably be an octal number, and which definitely should *not* be a string of octal digits: `0644` is okay, but `"0644"` is not. Returns the number of files successfully changed. See also [oct], page 406 if all you have is a string.

```
$cnt = chmod 0755, "foo", "bar";
chmod 0755, @executables;
$mode = "0644"; chmod $mode, "foo";      # !!! sets mode to
                                         # --w----r-T
$mode = "0644"; chmod oct($mode), "foo"; # this is better
$mode = 0644;   chmod $mode, "foo";      # this is best
```

On systems that support fchmod(2), you may pass filehandles among the files. On systems that don't support fchmod(2), passing filehandles raises an exception. Filehandles must be passed as globs or glob references to be recognized; barewords are considered filenames.

```
open(my $fh, "<", "foo");
my $perm = (stat $fh)[2] & 07777;
chmod($perm | 0600, $fh);
```

You can also import the symbolic `S_I*` constants from the `Fcntl` module:

```
use Fcntl qw( :mode );
chmod S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH, @executables;
# Identical to the chmod 0755 of the example above.
```

Portability issues: [perlport chmod], page 972.

chomp VARIABLE
chomp(LIST)
chomp

This safer version of [chop], page 363 removes any trailing string that corresponds to the current value of `$/` (also known as $INPUT_RECORD_SEPARATOR in the `English` module). It returns the total number of characters removed from all its arguments. It's often used to remove the newline from the end of an input record when you're worried that the final record may be missing its newline. When in paragraph mode (`$/ = ''`), it removes all trailing newlines from the string. When in slurp mode (`$/ = undef`) or fixed-length record mode (`$/` is a reference to an integer or the like; see Section 86.1 [perlvar NAME], page 1375) chomp() won't remove anything. If VARIABLE is omitted, it chomps `$_`. Example:

```
while (<>) {
    chomp;  # avoid \n on last field
    @array = split(/:/);
    # ...
}
```

If VARIABLE is a hash, it chomps the hash's values, but not its keys, resetting the `each` iterator in the process.

You can actually chomp anything that's an lvalue, including an assignment:

```
chomp($cwd = `pwd`);
chomp($answer = <STDIN>);
```

If you chomp a list, each element is chomped, and the total number of characters removed is returned.

Note that parentheses are necessary when you're chomping anything that is not a simple variable. This is because `chomp $cwd = `pwd`;` is interpreted as `(chomp $cwd) = `pwd`;`, rather than as `chomp( $cwd = `pwd` )` which you might expect. Similarly, `chomp $a, $b` is interpreted as `chomp($a), $b` rather than as `chomp($a, $b)`.

chop VARIABLE
chop(LIST)
chop

Chops off the last character of a string and returns the character chopped. It is much more efficient than `s/.$//s` because it neither scans nor copies the string. If VARIABLE is omitted, chops `$_`. If VARIABLE is a hash, it chops the hash's values, but not its keys, resetting the `each` iterator in the process.

You can actually chop anything that's an lvalue, including an assignment.

If you chop a list, each element is chopped. Only the value of the last `chop` is returned.

Note that `chop` returns the last character. To return all but the last character, use `substr($string, 0, -1)`.

See also [chomp], page 363.

chown LIST

Changes the owner (and group) of a list of files. The first two elements of the list must be the *numeric* uid and gid, in that order. A value of -1 in either position is interpreted by most systems to leave that value unchanged. Returns the number of files successfully changed.

```
    $cnt = chown $uid, $gid, 'foo', 'bar';
    chown $uid, $gid, @filenames;
```

On systems that support fchown(2), you may pass filehandles among the files. On systems that don't support fchown(2), passing filehandles raises an exception. Filehandles must be passed as globs or glob references to be recognized; barewords are considered filenames.

Here's an example that looks up nonnumeric uids in the passwd file:

```
    print "User: ";
    chomp($user = <STDIN>);
    print "Files: ";
    chomp($pattern = <STDIN>);

    ($login,$pass,$uid,$gid) = getpwnam($user)
        or die "$user not in passwd file";

    @ary = glob($pattern);  # expand filenames
    chown $uid, $gid, @ary;
```

On most systems, you are not allowed to change the ownership of the file unless you're the superuser, although you should be able to change the group to any of your secondary groups. On insecure systems, these restrictions may be relaxed, but this is not a portable assumption. On POSIX systems, you can detect this condition this way:

```
    use POSIX qw(sysconf _PC_CHOWN_RESTRICTED);
    $can_chown_giveaway = not sysconf(_PC_CHOWN_RESTRICTED);
```

Portability issues: [perlport chown], page 973.

chr NUMBER
chr

Returns the character represented by that NUMBER in the character set. For example, `chr(65)` is `"A"` in either ASCII or Unicode, and chr(0x263a) is a Unicode smiley face.

Negative values give the Unicode replacement character (chr(0xfffd)), except under the `bytes` pragma, where the low eight bits of the value (truncated to an integer) are used.

If NUMBER is omitted, uses `$_`.

For the reverse, use [ord], page 414.

Note that characters from 128 to 255 (inclusive) are by default internally not encoded as UTF-8 for backward compatibility reasons.

See Section 81.1 [perlunicode NAME], page 1317 for more about Unicode.

chroot FILENAME
chroot

This function works like the system call by the same name: it makes the named directory the new root directory for all further pathnames that begin with a `/` by your process and all its children. (It doesn't change your current working directory, which is unaffected.) For security reasons, this call is restricted to the superuser. If FILENAME is omitted, does a `chroot` to `$_`.

NOTE: It is good security practice to do `chdir("/")` (to the root directory) immediately after a `chroot()`.

Portability issues: [perlport chroot], page 973.

close FILEHANDLE
close

Closes the file or pipe associated with the filehandle, flushes the IO buffers, and closes the system file descriptor. Returns true if those operations succeed and if no error was reported by any PerlIO layer. Closes the currently selected filehandle if the argument is omitted.

You don't have to close FILEHANDLE if you are immediately going to do another `open` on it, because `open` closes it for you. (See [open], page 407.) However, an explicit `close` on an input file resets the line counter (`$.`), while the implicit close done by `open` does not.

If the filehandle came from a piped open, `close` returns false if one of the other syscalls involved fails or if its program exits with non-zero status. If the only problem was that the program exited non-zero, `$!` will be set to `0`. Closing a pipe also waits for the process executing on the pipe to exit–in case you wish to look at the output of the pipe afterwards–and implicitly puts the exit status value of that command into `$?` and `${^CHILD_ERROR_NATIVE}`.

If there are multiple threads running, `close` on a filehandle from a piped open returns true without waiting for the child process to terminate, if the filehandle is still open in another thread.

Closing the read end of a pipe before the process writing to it at the other end is done writing results in the writer receiving a SIGPIPE. If the other end can't handle that, be sure to read all the data before closing the pipe.

Example:

```
    open(OUTPUT, '|sort >foo')  # pipe to sort
        or die "Can't start sort: $!";
```

```
#...                         # print stuff to output
close OUTPUT                 # wait for sort to finish
    or warn $! ? "Error closing sort pipe: $!"
               : "Exit status $? from sort";
open(INPUT, 'foo')           # get sort's results
    or die "Can't open 'foo' for input: $!";
```

FILEHANDLE may be an expression whose value can be used as an indirect filehandle, usually the real filehandle name or an autovivified handle.

closedir DIRHANDLE

Closes a directory opened by `opendir` and returns the success of that system call.

connect SOCKET,NAME

Attempts to connect to a remote socket, just like connect(2). Returns true if it succeeded, false otherwise. NAME should be a packed address of the appropriate type for the socket. See the examples in Section 36.6 [perlipc Sockets: Client/Server Communication], page 682.

continue BLOCK
continue

When followed by a BLOCK, `continue` is actually a flow control statement rather than a function. If there is a `continue` BLOCK attached to a BLOCK (typically in a `while` or `foreach`), it is always executed just before the conditional is about to be evaluated again, just like the third part of a `for` loop in C. Thus it can be used to increment a loop variable, even when the loop has been continued via the `next` statement (which is similar to the C `continue` statement).

`last`, `next`, or `redo` may appear within a `continue` block; `last` and `redo` behave as if they had been executed within the main block. So will `next`, but since it will execute a `continue` block, it may be more entertaining.

```
while (EXPR) {
    ### redo always comes here
    do_something;
} continue {
    ### next always comes here
    do_something_else;
    # then back the top to re-check EXPR
}
### last always comes here
```

Omitting the `continue` section is equivalent to using an empty one, logically enough, so `next` goes directly back to check the condition at the top of the loop.

When there is no BLOCK, `continue` is a function that falls through the current `when` or `default` block instead of iterating a dynamically enclosing `foreach` or exiting a lexically enclosing `given`. In Perl 5.14 and earlier, this form of `continue` was only available when the `"switch"` feature was enabled. See `feature` and Section 74.2.11 [perlsyn Switch Statements], page 1258 for more information.

cos EXPR

cos

Returns the cosine of EXPR (expressed in radians). If EXPR is omitted, takes the cosine of `$_`.

For the inverse cosine operation, you may use the `Math::Trig::acos()` function, or use this relation:

```
    sub acos { atan2( sqrt(1 - $_[0] * $_[0]), $_[0] ) }
```

crypt PLAINTEXT,SALT

Creates a digest string exactly like the crypt(3) function in the C library (assuming that you actually have a version there that has not been extirpated as a potential munition).

crypt() is a one-way hash function. The PLAINTEXT and SALT are turned into a short string, called a digest, which is returned. The same PLAINTEXT and SALT will always return the same string, but there is no (known) way to get the original PLAINTEXT from the hash. Small changes in the PLAINTEXT or SALT will result in large changes in the digest.

There is no decrypt function. This function isn't all that useful for cryptography (for that, look for `Crypt` modules on your nearby CPAN mirror) and the name "crypt" is a bit of a misnomer. Instead it is primarily used to check if two pieces of text are the same without having to transmit or store the text itself. An example is checking if a correct password is given. The digest of the password is stored, not the password itself. The user types in a password that is crypt()'d with the same salt as the stored digest. If the two digests match, the password is correct.

When verifying an existing digest string you should use the digest as the salt (like `crypt($plain, $digest) eq $digest`). The SALT used to create the digest is visible as part of the digest. This ensures crypt() will hash the new string with the same salt as the digest. This allows your code to work with the standard ⟨undefined⟩ [crypt], page ⟨undefined⟩ and with more exotic implementations. In other words, assume nothing about the returned string itself nor about how many bytes of SALT may matter.

Traditionally the result is a string of 13 bytes: two first bytes of the salt, followed by 11 bytes from the set `[./0-9A-Za-z]`, and only the first eight bytes of PLAINTEXT mattered. But alternative hashing schemes (like MD5), higher level security schemes (like C2), and implementations on non-Unix platforms may produce different strings.

When choosing a new salt create a random two character string whose characters come from the set `[./0-9A-Za-z]` (like `join '', ('.', '/', 0..9, 'A'..'Z', 'a'..'z')[rand 64, rand 64]`). This set of characters is just a recommendation; the characters allowed in the salt depend solely on your system's crypt library, and Perl can't restrict what salts `crypt()` accepts.

Here's an example that makes sure that whoever runs this program knows their password:

```
    $pwd = (getpwuid($<))[1];
    system "stty -echo";
    print "Password: ";
    chomp($word = <STDIN>);
    print "\n";
    system "stty echo";

    if (crypt($word, $pwd) ne $pwd) {
        die "Sorry...\n";
    } else {
        print "ok\n";
    }
```

Of course, typing in your own password to whoever asks you for it is unwise.

The ⟨undefined⟩ [crypt], page ⟨undefined⟩ function is unsuitable for hashing large quantities of data, not least of all because you can't get the information back. Look at the `Digest` module for more robust algorithms.

If using crypt() on a Unicode string (which *potentially* has characters with codepoints above 255), Perl tries to make sense of the situation by trying to downgrade (a copy of) the string back to an eight-bit byte string before calling crypt() (on that copy). If that works, good. If not, crypt() dies with `Wide character in crypt`.

Portability issues: [perlport crypt], page 973.

dbmclose HASH

[This function has been largely superseded by the `untie` function.]

Breaks the binding between a DBM file and a hash.

Portability issues: [perlport dbmclose], page 973.

dbmopen HASH,DBNAME,MASK

[This function has been largely superseded by the [tie], page 473 function.]

This binds a dbm(3), ndbm(3), sdbm(3), gdbm(3), or Berkeley DB file to a hash. HASH is the name of the hash. (Unlike normal `open`, the first argument is *not* a filehandle, even though it looks like one). DBNAME is the name of the database (without the `.dir` or `.pag` extension if any). If the database does not exist, it is created with protection specified by MASK (as modified by the `umask`). To prevent creation of the database if it doesn't exist, you may specify a MODE of 0, and the function will return a false value if it can't find an existing database. If your system supports only the older DBM functions, you may make only one `dbmopen` call in your program. In older versions of Perl, if your system had neither DBM nor ndbm, calling `dbmopen` produced a fatal error; it now falls back to sdbm(3).

If you don't have write access to the DBM file, you can only read hash variables, not set them. If you want to test whether you can write, either use file tests or try setting a dummy hash entry inside an `eval` to trap the error.

Note that functions such as `keys` and `values` may return huge lists when used on large DBM files. You may prefer to use the `each` function to iterate over large DBM files. Example:

```
# print out history file offsets
dbmopen(%HIST,'/usr/lib/news/history',0666);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
dbmclose(%HIST);
```

See also AnyDBM_File for a more general description of the pros and cons of the various dbm approaches, as well as DB_File for a particularly rich implementation.

You can control which DBM library you use by loading that library before you call dbmopen():

```
use DB_File;
dbmopen(%NS_Hist, "$ENV{HOME}/.netscape/history.db")
    or die "Can't open netscape history file: $!";
```

Portability issues: [perlport dbmopen], page 973.

defined EXPR
defined

Returns a Boolean value telling whether EXPR has a value other than the undefined value undef. If EXPR is not present, $_ is checked.

Many operations return undef to indicate failure, end of file, system error, uninitialized variable, and other exceptional conditions. This function allows you to distinguish undef from other values. (A simple Boolean test will not distinguish among undef, zero, the empty string, and "0", which are all equally false.) Note that since undef is a valid scalar, its presence doesn't *necessarily* indicate an exceptional condition: pop returns undef when its argument is an empty array, *or* when the element to return happens to be undef.

You may also use defined(&func) to check whether subroutine &func has ever been defined. The return value is unaffected by any forward declarations of &func. A subroutine that is not defined may still be callable: its package may have an AUTOLOAD method that makes it spring into existence the first time that it is called; see Section 73.1 [perlsub NAME], page 1216.

Use of defined on aggregates (hashes and arrays) is deprecated. It used to report whether memory for that aggregate had ever been allocated. This behavior may disappear in future versions of Perl. You should instead use a simple test for size:

```
if (@an_array) { print "has array elements\n" }
if (%a_hash)   { print "has hash members\n"   }
```

When used on a hash element, it tells you whether the value is defined, not whether the key exists in the hash. Use ⟨undefined⟩ [exists], page ⟨undefined⟩ for the latter purpose.

Examples:

```
print if defined $switch{D};
print "$val\n" while defined($val = pop(@ary));
die "Can't readlink $sym: $!"
```

```
        unless defined($value = readlink $sym);
    sub foo { defined &$bar ? &$bar(@_) : die "No bar"; }
    $debugging = 0 unless defined $debugging;
```

Note: Many folks tend to overuse `defined` and are then surprised to discover that the number `0` and `""` (the zero-length string) are, in fact, defined values. For example, if you say

```
    "ab" =~ /a(.*)b/;
```

The pattern match succeeds and `$1` is defined, although it matched `"nothing"`. It didn't really fail to match anything. Rather, it matched something that happened to be zero characters long. This is all very above-board and honest. When a function returns an undefined value, it's an admission that it couldn't give you an honest answer. So you should use `defined` only when questioning the integrity of what you're trying to do. At other times, a simple comparison to `0` or `""` is what you want.

See also [undef], page 477, ⟨undefined⟩ [exists], page ⟨undefined⟩, [ref], page 435.

delete EXPR

Given an expression that specifies an element or slice of a hash, `delete` deletes the specified elements from that hash so that exists() on that element no longer returns true. Setting a hash element to the undefined value does not remove its key, but deleting it does; see ⟨undefined⟩ [exists], page ⟨undefined⟩.

In list context, returns the value or values deleted, or the last such element in scalar context. The return list's length always matches that of the argument list: deleting non-existent elements returns the undefined value in their corresponding positions.

delete() may also be used on arrays and array slices, but its behavior is less straightforward. Although exists() will return false for deleted entries, deleting array elements never changes indices of existing values; use shift() or splice() for that. However, if any deleted elements fall at the end of an array, the array's size shrinks to the position of the highest element that still tests true for exists(), or to 0 if none do. In other words, an array won't have trailing nonexistent elements after a delete.

WARNING: Calling `delete` on array values is strongly discouraged. The notion of deleting or checking the existence of Perl array elements is not conceptually coherent, and can lead to surprising behavior.

Deleting from `%ENV` modifies the environment. Deleting from a hash tied to a DBM file deletes the entry from the DBM file. Deleting from a `tied` hash or array may not necessarily return anything; it depends on the implementation of the `tied` package's DELETE method, which may do whatever it pleases.

The `delete local EXPR` construct localizes the deletion to the current block at run time. Until the block exits, elements locally deleted temporarily no longer exist. See Section 73.3.4.5 [perlsub Localized deletion of elements of composite types], page 1230.

```
    %hash = (foo => 11, bar => 22, baz => 33);
    $scalar = delete $hash{foo};         # $scalar is 11
```

```
$scalar = delete @hash{qw(foo bar)}; # $scalar is 22
@array  = delete @hash{qw(foo baz)}; # @array  is (undef,33)
```

The following (inefficiently) deletes all the values of %HASH and @ARRAY:

```
foreach $key (keys %HASH) {
    delete $HASH{$key};
}

foreach $index (0 .. $#ARRAY) {
    delete $ARRAY[$index];
}
```

And so do these:

```
delete @HASH{keys %HASH};

delete @ARRAY[0 .. $#ARRAY];
```

But both are slower than assigning the empty list or undefining %HASH or @ARRAY, which is the customary way to empty out an aggregate:

```
%HASH = ();     # completely empty %HASH
undef %HASH;    # forget %HASH ever existed

@ARRAY = ();    # completely empty @ARRAY
undef @ARRAY;   # forget @ARRAY ever existed
```

The EXPR can be arbitrarily complicated provided its final operation is an element or slice of an aggregate:

```
delete $ref->[$x][$y]{$key};
delete @{$ref->[$x][$y]}{$key1, $key2, @morekeys};

delete $ref->[$x][$y][$index];
delete @{$ref->[$x][$y]}[$index1, $index2, @moreindices];
```

die LIST

`die` raises an exception. Inside an `eval` the error message is stuffed into `$@` and the `eval` is terminated with the undefined value. If the exception is outside of all enclosing `eval`s, then the uncaught exception prints LIST to `STDERR` and exits with a non-zero value. If you need to exit the process with a specific exit code, see [exit], page 382.

Equivalent examples:

```
die "Can't cd to spool: $!\n" unless chdir '/usr/spool/news';
chdir '/usr/spool/news' or die "Can't cd to spool: $!\n"
```

If the last element of LIST does not end in a newline, the current script line number and input line number (if any) are also printed, and a newline is supplied. Note that the "input line number" (also known as "chunk") is subject to whatever notion of "line" happens to be currently in effect, and is also available as the special variable `$.`. See [perlvar $/], page 1395 and [perlvar $.], page 1395.

Hint: sometimes appending ", stopped" to your message will cause it to make better sense when the string "at foo line 123" is appended. Suppose you are running script "canasta".

```
die "/etc/games is no good";
die "/etc/games is no good, stopped";
```

produce, respectively

```
/etc/games is no good at canasta line 123.
/etc/games is no good, stopped at canasta line 123.
```

If the output is empty and $@ already contains a value (typically from a previous eval) that value is reused after appending "\t...propagated". This is useful for propagating exceptions:

```
eval { ... };
die unless $@ =~ /Expected exception/;
```

If the output is empty and $@ contains an object reference that has a PROPAGATE method, that method will be called with additional file and line number parameters. The return value replaces the value in $@; i.e., as if $@ = eval { $@->PROPAGATE(__FILE__, __LINE__) }; were called.

If $@ is empty then the string "Died" is used.

If an uncaught exception results in interpreter exit, the exit code is determined from the values of $! and $? with this pseudocode:

```
exit $! if $!;              # errno
exit $? >> 8 if $? >> 8;    # child exit status
exit 255;                   # last resort
```

The intent is to squeeze as much possible information about the likely cause into the limited space of the system exit code. However, as $! is the value of C's errno, which can be set by any system call, this means that the value of the exit code used by die can be non-predictable, so should not be relied upon, other than to be non-zero.

You can also call die with a reference argument, and if this is trapped within an eval, $@ contains that reference. This permits more elaborate exception handling using objects that maintain arbitrary state about the exception. Such a scheme is sometimes preferable to matching particular string values of $@ with regular expressions. Because $@ is a global variable and eval may be used within object implementations, be careful that analyzing the error object doesn't replace the reference in the global variable. It's easiest to make a local copy of the reference before any manipulations. Here's an example:

```
use Scalar::Util "blessed";

eval { ... ; die Some::Module::Exception->new( FOO => "bar" ) };
if (my $ev_err = $@) {
    if (blessed($ev_err)
        && $ev_err->isa("Some::Module::Exception")) {
        # handle Some::Module::Exception
    }
```

```
        else {
            # handle all other possible exceptions
        }
    }
```

Because Perl stringifies uncaught exception messages before display, you'll probably want to overload stringification operations on exception objects. See `overload` for details about that.

You can arrange for a callback to be run just before the `die` does its deed, by setting the `$SIG{__DIE__}` hook. The associated handler is called with the error text and can change the error message, if it sees fit, by calling `die` again. See [perlvar %SIG], page 1383 for details on setting `%SIG` entries, and [eval BLOCK], page 377 for some examples. Although this feature was to be run only right before your program was to exit, this is not currently so: the `$SIG{__DIE__}` hook is currently called even inside eval()ed blocks/strings! If one wants the hook to do nothing in such situations, put

```
    die @_ if $^S;
```

as the first line of the handler (see [perlvar $^S], page 1399). Because this promotes strange action at a distance, this counterintuitive behavior may be fixed in a future release.

See also exit(), warn(), and the Carp module.

do BLOCK

Not really a function. Returns the value of the last command in the sequence of commands indicated by BLOCK. When modified by the `while` or `until` loop modifier, executes the BLOCK once before testing the loop condition. (On other statements the loop modifiers test the conditional first.)

`do BLOCK` does *not* count as a loop, so the loop control statements `next`, `last`, or `redo` cannot be used to leave or restart the block. See Section 74.1 [perlsyn NAME], page 1249 for alternative strategies.

do EXPR

Uses the value of EXPR as a filename and executes the contents of the file as a Perl script.

```
    do 'stat.pl';
```

is largely like

```
    eval `cat stat.pl`;
```

except that it's more concise, runs no external processes, keeps track of the current filename for error messages, searches the `@INC` directories, and updates `%INC` if the file is found. See [perlvar @INC], page 1382 and [perlvar %INC], page 1382 for these variables. It also differs in that code evaluated with `do FILENAME` cannot see lexicals in the enclosing scope; `eval STRING` does. It's the same, however, in that it does reparse the file every time you call it, so you probably don't want to do this inside a loop.

If `do` can read the file but cannot compile it, it returns `undef` and sets an error message in `$@`. If `do` cannot read the file, it returns undef and sets `$!` to the error. Always check `$@` first, as compilation could fail in a way that also sets `$!`. If the file is successfully compiled, `do` returns the value of the last expression evaluated.

Inclusion of library modules is better done with the `use` and `require` operators, which also do automatic error checking and raise an exception if there's a problem.

You might like to use `do` to read in a program configuration file. Manual error checking can be done this way:

```
# read in config files: system first, then user
for $file ("/share/prog/defaults.rc",
           "$ENV{HOME}/.someprogrc")
{
    unless ($return = do $file) {
        warn "couldn't parse $file: $@" if $@;
        warn "couldn't do $file: $!"    unless defined $return;
        warn "couldn't run $file"       unless $return;
    }
}
```

dump LABEL
dump EXPR
dump

This function causes an immediate core dump. See also the **-u** command-line switch in Section 69.1 [perlrun NAME], page 1176, which does the same thing. Primarily this is so that you can use the **undump** program (not supplied) to turn your core dump into an executable binary after having initialized all your variables at the beginning of the program. When the new binary is executed it will begin by executing a `goto LABEL` (with all the restrictions that `goto` suffers). Think of it as a goto with an intervening core dump and reincarnation. If `LABEL` is omitted, restarts the program from the top. The `dump EXPR` form, available starting in Perl 5.18.0, allows a name to be computed at run time, being otherwise identical to `dump LABEL`.

WARNING: Any files opened at the time of the dump will *not* be open any more when the program is reincarnated, with possible resulting confusion by Perl.

This function is now largely obsolete, mostly because it's very hard to convert a core file into an executable. That's why you should now invoke it as `CORE::dump()`, if you don't want to be warned against a possible typo.

Unlike most named operators, this has the same precedence as assignment. It is also exempt from the looks-like-a-function rule, so `dump ("foo")."bar"` will cause "bar" to be part of the argument to `dump`.

Portability issues: [perlport dump], page 973.

each HASH
each ARRAY
each EXPR

When called on a hash in list context, returns a 2-element list consisting of the key and value for the next element of a hash. In Perl 5.12 and later only, it will also return the index and value for the next element of an array so that you can iterate over it; older Perls consider this a syntax error. When called in scalar context, returns only the key (not the value) in a hash, or the index in an array.

Hash entries are returned in an apparently random order. The actual random order is specific to a given hash; the exact same series of operations on two hashes may result in a different order for each hash. Any insertion into the hash may change the order, as will any deletion, with the exception that the most recent key returned by `each` or `keys` may be deleted without changing the order. So long as a given hash is unmodified you may rely on `keys`, `values` and `each` to repeatedly return the same order as each other. See Section 70.4.9 [perlsec Algorithmic Complexity Attacks], page 1205 for details on why hash order is randomized. Aside from the guarantees provided here the exact details of Perl's hash algorithm and the hash traversal order are subject to change in any release of Perl.

After `each` has returned all entries from the hash or array, the next call to `each` returns the empty list in list context and `undef` in scalar context; the next call following *that* one restarts iteration. Each hash or array has its own internal iterator, accessed by `each`, `keys`, and `values`. The iterator is implicitly reset when `each` has reached the end as just described; it can be explicitly reset by calling `keys` or `values` on the hash or array. If you add or delete a hash's elements while iterating over it, the effect on the iterator is unspecified; for example, entries may be skipped or duplicated–so don't do that. Exception: It is always safe to delete the item most recently returned by `each()`, so the following code works properly:

```
        while (($key, $value) = each %hash) {
          print $key, "\n";
          delete $hash{$key};   # This is safe
        }
```

Tied hashes may have a different ordering behaviour to perl's hash implementation.

This prints out your environment like the printenv(1) program, but in a different order:

```
    while (($key,$value) = each %ENV) {
        print "$key=$value\n";
    }
```

Starting with Perl 5.14, `each` can take a scalar EXPR, which must hold a reference to an unblessed hash or array. The argument will be dereferenced automatically. This aspect of `each` is considered highly experimental. The exact behaviour may change in a future version of Perl.

```
while (($key,$value) = each $hashref) { ... }
```

As of Perl 5.18 you can use a bare `each` in a `while` loop, which will set `$_` on every iteration.

```
while(each %ENV) {
    print "$_=$ENV{$_}\n";
}
```

To avoid confusing would-be users of your code who are running earlier versions of Perl with mysterious syntax errors, put this sort of thing at the top of your file to signal that your code will work *only* on Perls of a recent vintage:

```
use 5.012;  # so keys/values/each work on arrays
use 5.014;  # so keys/values/each work on scalars (experimental)
use 5.018;  # so each assigns to $_ in a lone while test
```

See also `keys`, `values`, and `sort`.

eof FILEHANDLE
eof ()

eof

Returns 1 if the next read on FILEHANDLE will return end of file *or* if FILEHANDLE is not open. FILEHANDLE may be an expression whose value gives the real filehandle. (Note that this function actually reads a character and then `ungetc`s it, so isn't useful in an interactive context.) Do not read from a terminal file (or call `eof(FILEHANDLE)` on it) after end-of-file is reached. File types such as terminals may lose the end-of-file condition if you do.

An `eof` without an argument uses the last file read. Using `eof()` with empty parentheses is different. It refers to the pseudo file formed from the files listed on the command line and accessed via the `<>` operator. Since `<>` isn't explicitly opened, as a normal filehandle is, an `eof()` before `<>` has been used will cause `@ARGV` to be examined to determine if input is available. Similarly, an `eof()` after `<>` has returned end-of-file will assume you are processing another `@ARGV` list, and if you haven't set `@ARGV`, will read input from `STDIN`; see Section 48.2.33 [perlop I/O Operators], page 844.

In a `while (<>)` loop, `eof` or `eof(ARGV)` can be used to detect the end of each file, whereas `eof()` will detect the end of the very last file only. Examples:

```
# reset line numbering on each input file
while (<>) {
    next if /^\s*#/;  # skip comments
    print "$.\t$_";
} continue {
    close ARGV if eof;  # Not eof()!
}

# insert dashes just before last line of last file
while (<>) {
    if (eof()) {  # check for end of last file
        print "--------------\n";
    }
    print;
    last if eof();     # needed if we're reading from a terminal
}
```

Practical hint: you almost never need to use `eof` in Perl, because the input operators typically return `undef` when they run out of data or encounter an error.

eval EXPR
eval BLOCK
eval

In the first form, often referred to as a "string eval", the return value of EXPR is parsed and executed as if it were a little Perl program. The value of the expression (which is itself determined within scalar context) is first parsed, and if there were no errors, executed as a block within the lexical context of the current Perl program. This means, that in particular, any outer lexical variables are visible to it, and any package variable settings or subroutine and format definitions remain afterwards.

Note that the value is parsed every time the `eval` executes. If EXPR is omitted, evaluates `$_`. This form is typically used to delay parsing and subsequent execution of the text of EXPR until run time.

If the `unicode_eval` feature is enabled (which is the default under a `use 5.16` or higher declaration), EXPR or `$_` is treated as a string of characters, so `use utf8` declarations have no effect, and source filters are forbidden. In the absence of the `unicode_eval` feature, the string will sometimes be treated as characters and sometimes as bytes, depending on the internal encoding, and source filters activated within the `eval` exhibit the erratic, but historical, behaviour of affecting some outer file scope that is still compiling. See also the [evalbytes], page 380 keyword, which always treats its input as a byte stream and works properly with source filters, and the `feature` pragma.

Problems can arise if the string expands a scalar containing a floating point number. That scalar can expand to letters, such as `"NaN"` or `"Infinity"`; or, within the scope of a `use locale`, the decimal point character may be something other than a dot (such as a comma). None of these are likely to parse as you are likely expecting.

In the second form, the code within the BLOCK is parsed only once–at the same time the code surrounding the `eval` itself was parsed–and executed within the context of the current Perl program. This form is typically used to trap exceptions more efficiently than the first (see below), while also providing the benefit of checking the code within BLOCK at compile time.

The final semicolon, if any, may be omitted from the value of EXPR or within the BLOCK.

In both forms, the value returned is the value of the last expression evaluated inside the mini-program; a return statement may be also used, just as with subroutines. The expression providing the return value is evaluated in void, scalar, or list context, depending on the context of the eval itself. See [wantarray], page 488 for more on how the evaluation context can be determined.

If there is a syntax error or runtime error, or a die statement is executed, eval returns undef in scalar context or an empty list in list context, and $@ is set to the error message. (Prior to 5.16, a bug caused undef to be returned in list context for syntax errors, but not for runtime errors.) If there was no error, $@ is set to the empty string. A control flow operator like last or goto can bypass the setting of $@. Beware that using eval neither silences Perl from printing warnings to STDERR, nor does it stuff the text of warning messages into $@. To do either of those, you have to use the $SIG{__WARN__} facility, or turn off warnings inside the BLOCK or EXPR using no warnings 'all'. See ⟨undefined⟩ [warn], page ⟨undefined⟩, Section 86.1 [perlvar NAME], page 1375, and warnings.

Note that, because eval traps otherwise-fatal errors, it is useful for determining whether a particular feature (such as socket or symlink) is implemented. It is also Perl's exception-trapping mechanism, where the die operator is used to raise exceptions.

If you want to trap errors when loading an XS module, some problems with the binary interface (such as Perl version skew) may be fatal even with eval unless $ENV{PERL_DL_NONLAZY} is set. See Section 69.1 [perlrun NAME], page 1176.

If the code to be executed doesn't vary, you may use the eval-BLOCK form to trap run-time errors without incurring the penalty of recompiling each time. The error, if any, is still returned in $@. Examples:

```
# make divide-by-zero nonfatal
eval { $answer = $a / $b; }; warn $@ if $@;

# same thing, but less efficient
eval '$answer = $a / $b'; warn $@ if $@;

# a compile-time error
eval { $answer = }; # WRONG

# a run-time error
eval '$answer =';   # sets $@
```

Using the eval{} form as an exception trap in libraries does have some issues. Due to the current arguably broken state of __DIE__ hooks, you may wish not to trigger any __DIE__ hooks that user code may have installed. You can use the local $SIG{__DIE__} construct for this purpose, as this example shows:

```
# a private exception trap for divide-by-zero
eval { local $SIG{'__DIE__'}; $answer = $a / $b; };
warn $@ if $@;
```

This is especially significant, given that __DIE__ hooks can call die again, which has the effect of changing their error messages:

```
# __DIE__ hooks may modify error messages
{
   local $SIG{'__DIE__'} =
          sub { (my $x = $_[0]) =~ s/foo/bar/g; die $x };
   eval { die "foo lives here" };
   print $@ if $@;                # prints "bar lives here"
}
```

Because this promotes action at a distance, this counterintuitive behavior may be fixed in a future release.

With an `eval`, you should be especially careful to remember what's being looked at when:

```
    eval $x;        # CASE 1
    eval "$x";      # CASE 2

    eval '$x';      # CASE 3
    eval { $x };    # CASE 4

    eval "\$$x++";  # CASE 5
    $$x++;          # CASE 6
```

Cases 1 and 2 above behave identically: they run the code contained in the variable $x. (Although case 2 has misleading double quotes making the reader wonder what else might be happening (nothing is).) Cases 3 and 4 likewise behave in the same way: they run the code `'$x'`, which does nothing but return the value of $x. (Case 4 is preferred for purely visual reasons, but it also has the advantage of compiling at compile-time instead of at run-time.) Case 5 is a place where normally you *would* like to use double quotes, except that in this particular situation, you can just use symbolic references instead, as in case 6.

Before Perl 5.14, the assignment to `$@` occurred before restoration of localized variables, which means that for your code to run on older versions, a temporary is required if you want to mask some but not all errors:

```
    # alter $@ on nefarious repugnancy only
    {
       my $e;
       {
         local $@; # protect existing $@
         eval { test_repugnancy() };
         # $@ =~ /nefarious/ and die $@; # Perl 5.14 and higher only
         $@ =~ /nefarious/ and $e = $@;
       }
       die $e if defined $e
    }
```

`eval BLOCK` does *not* count as a loop, so the loop control statements `next`, `last`, or `redo` cannot be used to leave or restart the block.

An `eval ''` executed within a subroutine defined in the `DB` package doesn't see the usual surrounding lexical scope, but rather the scope of the first non-DB piece of code that called it. You don't normally need to worry about this unless you are writing a Perl debugger.

evalbytes EXPR
evalbytes

This function is like [eval], page 377 with a string argument, except it always parses its argument, or `$_` if EXPR is omitted, as a string of bytes. A string containing characters whose ordinal value exceeds 255 results in an error. Source filters activated within the evaluated code apply to the code itself.

This function is only available under the `evalbytes` feature, a `use v5.16` (or higher) declaration, or with a `CORE::` prefix. See `feature` for more information.

exec LIST

exec PROGRAM LIST

The `exec` function executes a system command *and never returns*; use `system` instead of `exec` if you want it to return. It fails and returns false only if the command does not exist *and* it is executed directly instead of via your system's command shell (see below).

Since it's a common mistake to use `exec` instead of `system`, Perl warns you if `exec` is called in void context and if there is a following statement that isn't `die`, `warn`, or `exit` (if `-w` is set–but you always do that, right?). If you *really* want to follow an `exec` with some other statement, you can use one of these styles to avoid the warning:

```
exec ('foo')   or print STDERR "couldn't exec foo: $!";
{ exec ('foo') }; print STDERR "couldn't exec foo: $!";
```

If there is more than one argument in LIST, this calls execvp(3) with the arguments in LIST. If there is only one element in LIST, the argument is checked for shell metacharacters, and if there are any, the entire argument is passed to the system's command shell for parsing (this is `/bin/sh -c` on Unix platforms, but varies on other platforms). If there are no shell metacharacters in the argument, it is split into words and passed directly to `execvp`, which is more efficient. Examples:

```
exec '/bin/echo', 'Your arguments are: ', @ARGV;
exec "sort $outfile | uniq";
```

If you don't really want to execute the first argument, but want to lie to the program you are executing about its own name, you can specify the program you actually want to run as an "indirect object" (without a comma) in front of the LIST, as in `exec PROGRAM LIST`. (This always forces interpretation of the LIST as a multivalued list, even if there is only a single scalar in the list.) Example:

```
$shell = '/bin/csh';
exec $shell '-sh';    # pretend it's a login shell
```

or, more directly,

```
exec {'/bin/csh'} '-sh';  # pretend it's a login shell
```

When the arguments get executed via the system shell, results are subject to its quirks and capabilities. See [perlop 'STRING'], page 832 for details.

Using an indirect object with `exec` or `system` is also more secure. This usage (which also works fine with system()) forces interpretation of the arguments as a multivalued list, even if the list had just one argument. That way you're safe from the shell expanding wildcards or splitting up words with whitespace in them.

```
@args = ( "echo surprise" );

exec @args;               # subject to shell escapes
                            # if @args == 1
exec { $args[0] } @args;  # safe even with one-arg list
```

The first version, the one without the indirect object, ran the *echo* program, passing it `"surprise"` an argument. The second version didn't; it tried to run a program named "*echo surprise*", didn't find it, and set `$?` to a non-zero value indicating failure.

On Windows, only the `exec PROGRAM LIST` indirect object syntax will reliably avoid using the shell; `exec LIST`, even with more than one element, will fall back to the shell if the first spawn fails.

Perl attempts to flush all files opened for output before the exec, but this may not be supported on some platforms (see Section 56.1 [perlport NAME], page 951). To be safe, you may need to set `$|` ($AUTOFLUSH in English) or call the `autoflush()` method of `IO::Handle` on any open handles to avoid lost output.

Note that `exec` will not call your `END` blocks, nor will it invoke `DESTROY` methods on your objects.

Portability issues: [perlport exec], page 973.

exists EXPR

Given an expression that specifies an element of a hash, returns true if the specified element in the hash has ever been initialized, even if the corresponding value is undefined.

```
print "Exists\n"    if exists $hash{$key};
print "Defined\n"   if defined $hash{$key};
print "True\n"      if $hash{$key};
```

exists may also be called on array elements, but its behavior is much less obvious and is strongly tied to the use of ⟨undefined⟩ [delete], page ⟨undefined⟩ on arrays.

WARNING: Calling `exists` on array values is strongly discouraged. The notion of deleting or checking the existence of Perl array elements is not conceptually coherent, and can lead to surprising behavior.

```
print "Exists\n"    if exists $array[$index];
print "Defined\n"   if defined $array[$index];
print "True\n"      if $array[$index];
```

A hash or array element can be true only if it's defined and defined only if it exists, but the reverse doesn't necessarily hold true.

Given an expression that specifies the name of a subroutine, returns true if the specified subroutine has ever been declared, even if it is undefined. Mentioning a subroutine name for exists or defined does not count as declaring it. Note that a subroutine that does not exist may still be callable: its package may have an `AUTOLOAD` method that makes it spring into existence the first time that it is called; see Section 73.1 [perlsub NAME], page 1216.

```
print "Exists\n"  if exists &subroutine;
print "Defined\n" if defined &subroutine;
```

Note that the EXPR can be arbitrarily complicated as long as the final operation is a hash or array key lookup or subroutine name:

```
if (exists $ref->{A}->{B}->{$key})  { }
if (exists $hash{A}{B}{$key})       { }

if (exists $ref->{A}->{B}->[$ix])   { }
if (exists $hash{A}{B}[$ix])        { }

if (exists &{$ref->{A}{B}{$key}})   { }
```

Although the most deeply nested array or hash element will not spring into existence just because its existence was tested, any intervening ones will. Thus `$ref->{"A"}` and `$ref->{"A"}->{"B"}` will spring into existence due to the existence test for the $key element above. This happens anywhere the arrow operator is used, including even here:

```
undef $ref;
if (exists $ref->{"Some key"})    { }
print $ref;  # prints HASH(0x80d3d5c)
```

This surprising autovivification in what does not at first–or even second–glance appear to be an lvalue context may be fixed in a future release.

Use of a subroutine call, rather than a subroutine name, as an argument to exists() is an error.

```
exists &sub;    # OK
exists &sub();  # Error
```

exit EXPR
exit

Evaluates EXPR and exits immediately with that value. Example:

```
$ans = <STDIN>;
exit 0 if $ans =~ /^[Xx]/;
```

See also `die`. If EXPR is omitted, exits with `0` status. The only universally recognized values for EXPR are `0` for success and `1` for error; other values are subject to interpretation depending on the environment in which the Perl program is running. For example, exiting 69 (EX_UNAVAILABLE) from a *sendmail* incoming-mail filter will cause the mailer to return the item undelivered, but that's not true everywhere.

Don't use `exit` to abort a subroutine if there's any chance that someone might want to trap whatever error happened. Use `die` instead, which can be trapped by an `eval`.

The exit() function does not always exit immediately. It calls any defined `END` routines first, but these `END` routines may not themselves abort the exit. Likewise any object destructors that need to be called are called before the real exit. `END` routines and destructors can change the exit status by modifying `$?`. If this is a problem, you can call `POSIX::_exit($status)` to avoid END and destructor processing. See Section 40.1 [perlmod NAME], page 732 for details.

Portability issues: [perlport exit], page 973.

exp EXPR
exp

Returns *e* (the natural logarithm base) to the power of EXPR. If EXPR is omitted, gives `exp($_)`.

fc EXPR

fc

Returns the casefolded version of EXPR. This is the internal function implementing the `\F` escape in double-quoted strings.

Casefolding is the process of mapping strings to a form where case differences are erased; comparing two strings in their casefolded form is effectively a way of asking if two strings are equal, regardless of case.

Roughly, if you ever found yourself writing this

```
lc($this) eq lc($that)    # Wrong!
    # or
uc($this) eq uc($that)    # Also wrong!
    # or
$this =~ /^\Q$that\E\z/i  # Right!
```

Now you can write

```
fc($this) eq fc($that)
```

And get the correct results.

Perl only implements the full form of casefolding, but you can access the simple folds using Section “casefold()” in `Unicode-UCD` and Section “prop_invmap()” in `Unicode-UCD`. For further information on casefolding, refer to the Unicode Standard, specifically sections 3.13 `Default Case Operations`, 4.2 `Case-Normative`, and 5.18 `Case Mappings`, available at `http://www.unicode.org/versions/latest/`, as well as the Case Charts available at `http://www.unicode.org/charts/case/`.

If EXPR is omitted, uses `$_`.

This function behaves the same way under various pragma, such as within `"use feature 'unicode_strings"`, as [lc], page 399 does, with the single exception of `fc` of LATIN CAPITAL LETTER SHARP S (U+1E9E) within the scope of `use locale`. The foldcase of this character would normally be `"ss"`, but as explained in the [lc], page 399 section, case changes that cross the 255/256 boundary are problematic under locales, and are hence prohibited. Therefore, this function under locale returns instead the string `"\x{17F}\x{17F}"`, which is the LATIN SMALL LETTER LONG S. Since that character itself folds to `"s"`, the string of two of them together should be equivalent to a single U+1E9E when foldcased.

While the Unicode Standard defines two additional forms of casefolding, one for Turkic languages and one that never maps one character into multiple characters, these are not provided by the Perl core; However, the CPAN module `Unicode::Casing` may be used to provide an implementation.

This keyword is available only when the `"fc"` feature is enabled, or when prefixed with `CORE::`; See `feature`. Alternately, include a `use v5.16` or later to the current scope.

fcntl FILEHANDLE,FUNCTION,SCALAR

Implements the fcntl(2) function. You'll probably have to say

```
use Fcntl;
```

first to get the correct constant definitions. Argument processing and value returned work just like `ioctl` below. For example:

```
use Fcntl;
fcntl($filehandle, F_GETFL, $packed_return_buffer)
    or die "can't fcntl F_GETFL: $!";
```

You don't have to check for `defined` on the return from `fcntl`. Like `ioctl`, it maps a `0` return from the system call into `"0 but true"` in Perl. This string is true in boolean context and `0` in numeric context. It is also exempt from the normal **`-w`** warnings on improper numeric conversions.

Note that `fcntl` raises an exception if used on a machine that doesn't implement fcntl(2). See the Fcntl module or your fcntl(2) manpage to learn what functions are available on your system.

Here's an example of setting a filehandle named `REMOTE` to be non-blocking at the system level. You'll have to negotiate `$|` on your own, though.

```
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);

$flags = fcntl(REMOTE, F_GETFL, 0)
            or die "Can't get flags for the socket: $!\n";

$flags = fcntl(REMOTE, F_SETFL, $flags | O_NONBLOCK)
            or die "Can't set flags for the socket: $!\n";
```

Portability issues: [perlport fcntl], page 974.

__FILE__

A special token that returns the name of the file in which it occurs.

fileno FILEHANDLE

Returns the file descriptor for a filehandle, or undefined if the filehandle is not open. If there is no real file descriptor at the OS level, as can happen with filehandles connected to memory objects via `open` with a reference for the third argument, -1 is returned.

This is mainly useful for constructing bitmaps for `select` and low-level POSIX tty-handling operations. If FILEHANDLE is an expression, the value is taken as an indirect filehandle, generally its name.

You can use this to find out whether two handles refer to the same underlying descriptor:

```
if (fileno(THIS) != -1 && fileno(THIS) == fileno(THAT)) {
    print "THIS and THAT are dups\n";
} elsif (fileno(THIS) != -1 && fileno(THAT) != -1) {
    print "THIS and THAT have different " .
        "underlying file descriptors\n";
} else {
    print "At least one of THIS and THAT does " .
        "not have a real file descriptor\n";
}
```

The behavior of `fileno` on a directory handle depends on the operating system. On a system with dirfd(3) or similar, `fileno` on a directory handle returns the underlying file descriptor associated with the handle; on systems with no such support, it returns the undefined value, and sets `$!` (errno).

flock FILEHANDLE,OPERATION

Calls flock(2), or an emulation of it, on FILEHANDLE. Returns true for success, false on failure. Produces a fatal error if used on a machine that doesn't implement flock(2), fcntl(2) locking, or lockf(3). `flock` is Perl's portable file-locking interface, although it locks entire files only, not records.

Two potentially non-obvious but traditional `flock` semantics are that it waits indefinitely until the lock is granted, and that its locks are **merely advisory**. Such discretionary locks are more flexible, but offer fewer guarantees. This means that programs that do not also use `flock` may modify files locked with `flock`. See Section 56.1 [perlport NAME], page 951, your port's specific documentation, and your system-specific local manpages for details. It's best to assume traditional behavior if you're writing portable programs. (But if you're not, you should as always feel perfectly free to write for your own system's idiosyncrasies (sometimes called "features"). Slavish adherence to portability concerns shouldn't get in the way of your getting your job done.)

OPERATION is one of LOCK_SH, LOCK_EX, or LOCK_UN, possibly combined with LOCK_NB. These constants are traditionally valued 1, 2, 8 and 4, but you can use the symbolic names if you import them from the `Fcntl` module, either individually, or as a group using the `:flock` tag. LOCK_SH requests a shared lock, LOCK_EX requests an exclusive lock, and LOCK_UN releases a previously requested lock. If LOCK_NB is bitwise-or'ed with LOCK_SH or LOCK_EX, then `flock` returns immediately rather than blocking waiting for the lock; check the return status to see if you got it.

To avoid the possibility of miscoordination, Perl now flushes FILEHANDLE before locking or unlocking it.

Note that the emulation built with lockf(3) doesn't provide shared locks, and it requires that FILEHANDLE be open with write intent. These are the semantics that lockf(3) implements. Most if not all systems implement lockf(3) in terms of fcntl(2) locking, though, so the differing semantics shouldn't bite too many people.

Note that the fcntl(2) emulation of flock(3) requires that FILEHANDLE be open with read intent to use LOCK_SH and requires that it be open with write intent to use LOCK_EX.

Note also that some versions of `flock` cannot lock things over the network; you would need to use the more system-specific `fcntl` for that. If you like you can force Perl to ignore your system's flock(2) function, and so provide its own fcntl(2)-based emulation, by passing the switch `-Ud_flock` to the `Configure` program when you configure and build a new Perl.

Here's a mailbox appender for BSD systems.

```
    # import LOCK_* and SEEK_END constants
    use Fcntl qw(:flock SEEK_END);

    sub lock {
        my ($fh) = @_;
        flock($fh, LOCK_EX) or die "Cannot lock mailbox - $!\n";

        # and, in case someone appended while we were waiting...
        seek($fh, 0, SEEK_END) or die "Cannot seek - $!\n";
    }

    sub unlock {
        my ($fh) = @_;
        flock($fh, LOCK_UN) or die "Cannot unlock mailbox - $!\n";
    }

    open(my $mbox, ">>", "/usr/spool/mail/$ENV{'USER'}")
        or die "Can't open mailbox: $!";

    lock($mbox);
    print $mbox $msg,"\n\n";
    unlock($mbox);
```

On systems that support a real flock(2), locks are inherited across fork() calls, whereas those that must resort to the more capricious fcntl(2) function lose their locks, making it seriously harder to write servers.

See also `DB_File` for other flock() examples.

Portability issues: [perlport flock], page 974.

fork

Does a fork(2) system call to create a new process running the same program at the same point. It returns the child pid to the parent process, `0` to the child process, or `undef` if the fork is unsuccessful. File descriptors (and sometimes locks on those descriptors) are shared, while everything else is copied. On most systems supporting fork(), great care has gone into making it extremely efficient (for example, using copy-on-write technology on data pages), making it the dominant paradigm for multitasking over the last few decades.

Perl attempts to flush all files opened for output before forking the child process, but this may not be supported on some platforms (see Section 56.1 [perlport NAME], page 951). To be safe, you may need to set `$|` ($AUTOFLUSH in English) or call the `autoflush()` method of `IO::Handle` on any open handles to avoid duplicate output.

If you `fork` without ever waiting on your children, you will accumulate zombies. On some systems, you can avoid this by setting `$SIG{CHLD}` to `"IGNORE"`. See also Section 36.1 [perlipc NAME], page 667 for more examples of forking and reaping moribund children.

Note that if your forked child inherits system file descriptors like STDIN and STDOUT that are actually connected by a pipe or socket, even if you exit, then the remote server (such as, say, a CGI script or a backgrounded job launched from a remote shell) won't think you're done. You should reopen those to `/dev/null` if it's any issue.

On some platforms such as Windows, where the fork() system call is not available, Perl can be built to emulate fork() in the Perl interpreter. The emulation is designed, at the level of the Perl program, to be as compatible as possible with the `"Unix"` fork(). However it has limitations that have to be considered in code intended to be portable. See Section 23.1 [perlfork NAME], page 337 for more details.

Portability issues: [perlport fork], page 974.

format

Declare a picture format for use by the `write` function. For example:

```
format Something =
    Test: @<<<<<<<< @||||| @>>>>>
          $str,     $%,    '$' . int($num)
.

$str = "widget";
$num = $cost/$quantity;
$~ = 'Something';
write;
```

See Section 24.1 [perlform NAME], page 343 for many details and examples.

formline PICTURE,LIST

This is an internal function used by `format`s, though you may call it, too. It formats (see Section 24.1 [perlform NAME], page 343) a list of values according to the contents of PICTURE, placing the output into the format output accumulator, `$^A` (or `$ACCUMULATOR` in English). Eventually, when a `write` is done, the contents of `$^A` are written to some filehandle. You could also read `$^A` and then set `$^A` back to `""`. Note that a format typically does one `formline` per line of form, but the `formline` function itself doesn't care how many newlines are embedded in the PICTURE. This means that the `~` and `~~` tokens treat the entire PICTURE as a single line. You may therefore need to use multiple formlines to implement a single record format, just like the `format` compiler.

Be careful if you put double quotes around the picture, because an `@` character may be taken to mean the beginning of an array name. `formline` always returns true. See Section 24.1 [perlform NAME], page 343 for other examples.

If you are trying to use this instead of `write` to capture the output, you may find it easier to open a filehandle to a scalar (`open $fh, ">", \$output`) and write to that instead.

getc FILEHANDLE
getc

Returns the next character from the input file attached to FILEHANDLE, or the undefined value at end of file or if there was an error (in the latter case `$!` is set). If FILEHANDLE is omitted, reads from STDIN. This is not particularly efficient. However, it cannot be used by itself to fetch single characters without waiting for the user to hit enter. For that, try something more like:

```
if ($BSD_STYLE) {
    system "stty cbreak </dev/tty >/dev/tty 2>&1";
}
else {
    system "stty", ’-icanon’, ’eol’, "\001";
}

$key = getc(STDIN);

if ($BSD_STYLE) {
    system "stty -cbreak </dev/tty >/dev/tty 2>&1";
}
else {
    system ’stty’, ’icanon’, ’eol’, ’^@’; # ASCII NUL
}
print "\n";
```

Determination of whether $BSD_STYLE should be set is left as an exercise to the reader.

The `POSIX::getattr` function can do this more portably on systems purporting POSIX compliance. See also the `Term::ReadKey` module from your nearest CPAN (`http://www.cpan.org`) site.

getlogin

This implements the C library function of the same name, which on most systems returns the current login from `/etc/utmp`, if any. If it returns the empty string, use `getpwuid`.

```
$login = getlogin || getpwuid($<) || "Kilroy";
```

Do not consider `getlogin` for authentication: it is not as secure as `getpwuid`.

Portability issues: [perlport getlogin], page 974.

getpeername SOCKET

Returns the packed sockaddr address of the other end of the SOCKET connection.

```
use Socket;
$hersockaddr    = getpeername(SOCK);
($port, $iaddr) = sockaddr_in($hersockaddr);
$herhostname    = gethostbyaddr($iaddr, AF_INET);
$herstraddr     = inet_ntoa($iaddr);
```

getpgrp PID

Returns the current process group for the specified PID. Use a PID of `0` to get the current process group for the current process. Will raise an exception if used on a machine that doesn't implement getpgrp(2). If PID is omitted, returns the process group of the current process. Note that the POSIX version of `getpgrp` does not accept a PID argument, so only `PID==0` is truly portable.

Portability issues: [perlport getpgrp], page 974.

getppid

Returns the process id of the parent process.

Note for Linux users: Between v5.8.1 and v5.16.0 Perl would work around non-POSIX thread semantics the minority of Linux systems (and Debian GNU/kFreeBSD systems) that used LinuxThreads, this emulation has since been removed. See the documentation for [$$], page 1377 for details.

Portability issues: [perlport getppid], page 974.

getpriority WHICH,WHO

Returns the current priority for a process, a process group, or a user. (See getpriority(2).) Will raise a fatal exception if used on a machine that doesn't implement getpriority(2).

Portability issues: [perlport getpriority], page 974.

getpwnam NAME
getgrnam NAME
gethostbyname NAME
getnetbyname NAME
getprotobyname NAME
getpwuid UID
getgrgid GID
getservbyname NAME,PROTO
gethostbyaddr ADDR,ADDRTYPE
getnetbyaddr ADDR,ADDRTYPE
getprotobynumber NUMBER
getservbyport PORT,PROTO
getpwent

getgrent

gethostent

getnetent

getprotoent
getservent

setpwent

setgrent

sethostent STAYOPEN
setnetent STAYOPEN
setprotoent STAYOPEN
setservent STAYOPEN
endpwent

endgrent

endhostent
endnetent

endprotoent
endservent

These routines are the same as their counterparts in the system C library. In list context, the return values from the various get routines are as follows:

```
# 0         1          2           3         4
( $name,    $passwd,   $gid,       $members  ) = getgr*
( $name,    $aliases,  $addrtype,  $net      ) = getnet*
( $name,    $aliases,  $port,      $proto    ) = getserv*
( $name,    $aliases,  $proto                ) = getproto*
( $name,    $aliases,  $addrtype,  $length,  @addrs ) = gethost*
( $name,    $passwd,   $uid,       $gid,     $quota,
$comment,   $gcos,     $dir,       $shell,   $expire ) = getpw*
# 5         6          7           8         9
```

(If the entry doesn't exist, the return value is a single meaningless true value.)

The exact meaning of the $gcos field varies but it usually contains the real name of the user (as opposed to the login name) and other information pertaining to the user. Beware, however, that in many system users are able to change this information and therefore it cannot be trusted and therefore the $gcos is tainted (see Section 70.1 [perlsec NAME], page 1198). The $passwd and $shell, user's encrypted password and login shell, are also tainted, for the same reason.

In scalar context, you get the name, unless the function was a lookup by name, in which case you get the other thing, whatever it is. (If the entry doesn't exist you get the undefined value.) For example:

```
    $uid   = getpwnam($name);
    $name  = getpwuid($num);
    $name  = getpwent();
    $gid   = getgrnam($name);
    $name  = getgrgid($num);
    $name  = getgrent();
    #etc.
```

In *getpw*()* the fields $quota, $comment, and $expire are special in that they are unsupported on many systems. If the $quota is unsupported, it is an empty scalar. If it is supported, it usually encodes the disk quota. If the $comment field is unsupported, it is an empty scalar. If it is supported it usually encodes some administrative comment about the user. In some systems the $quota field may be $change or $age, fields that have to do with password aging. In some systems the $comment field may be $class. The $expire field, if present, encodes the expiration period of the account or the password. For the availability and the exact meaning of these fields in your system, please consult getpwnam(3) and your system's `pwd.h` file. You can also find out from within Perl what your $quota and $comment fields mean and whether you have the $expire field by using the `Config` module and the values `d_pwquota`, `d_pwage`, `d_pwchange`, `d_pwcomment`, and `d_pwexpire`. Shadow password files are supported only if your vendor has implemented them in the intuitive fashion that calling the regular C library routines gets the shadow versions if you're running under privilege or if there exists the shadow(3) functions as found in System V (this includes Solaris and Linux). Those systems that implement a proprietary shadow password facility are unlikely to be supported.

The $members value returned by *getgr*()* is a space-separated list of the login names of the members of the group.

For the *gethost*()* functions, if the `h_errno` variable is supported in C, it will be returned to you via `$?` if the function call fails. The `@addrs` value returned by a successful call is a list of raw addresses returned by the corresponding library call. In the Internet domain, each address is four bytes long; you can unpack it by saying something like:

```
    ($a,$b,$c,$d) = unpack('W4',$addr[0]);
```

The Socket library makes this slightly easier:

```
    use Socket;
    $iaddr = inet_aton("127.1"); # or whatever address
    $name  = gethostbyaddr($iaddr, AF_INET);

    # or going the other way
    $straddr = inet_ntoa($iaddr);
```

In the opposite way, to resolve a hostname to the IP address you can write this:

```
    use Socket;
    $packed_ip = gethostbyname("www.perl.org");
    if (defined $packed_ip) {
        $ip_address = inet_ntoa($packed_ip);
    }
```

Make sure `gethostbyname()` is called in SCALAR context and that its return value is checked for definedness.

The `getprotobynumber` function, even though it only takes one argument, has the precedence of a list operator, so beware:

```
    getprotobynumber $number eq 'icmp'   # WRONG
    getprotobynumber($number eq 'icmp')  # actually means this
    getprotobynumber($number) eq 'icmp'  # better this way
```

If you get tired of remembering which element of the return list contains which return value, by-name interfaces are provided in standard modules:

File::stat, Net::hostent, Net::netent, Net::protoent, Net::servent, Time::gmtime, Time::localtime, and User::grent. These override the normal built-ins, supplying versions that return objects with the appropriate names for each field. For example:

```
use File::stat;
use User::pwent;
$is_his = (stat($filename)->uid == pwent($whoever)->uid);
```

Even though it looks as though they're the same method calls (uid), they aren't, because a File::stat object is different from a User::pwent object.

Portability issues: [perlport getpwnam], page 974 to [perlport endservent], page 976.

getsockname SOCKET

Returns the packed sockaddr address of this end of the SOCKET connection, in case you don't know the address because you have several different IPs that the connection might have come in on.

```
use Socket;
$mysockaddr = getsockname(SOCK);
($port, $myaddr) = sockaddr_in($mysockaddr);
printf "Connect to %s [%s]\n",
   scalar gethostbyaddr($myaddr, AF_INET),
   inet_ntoa($myaddr);
```

getsockopt SOCKET,LEVEL,OPTNAME

Queries the option named OPTNAME associated with SOCKET at a given LEVEL. Options may exist at multiple protocol levels depending on the socket type, but at least the uppermost socket level SOL_SOCKET (defined in the Socket module) will exist. To query options at another level the protocol number of the appropriate protocol controlling the option should be supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, LEVEL should be set to the protocol number of TCP, which you can get using getprotobyname.

The function returns a packed string representing the requested socket option, or undef on error, with the reason for the error placed in $!. Just what is in the packed string depends on LEVEL and OPTNAME; consult getsockopt(2) for details. A common case is that the option is an integer, in which case the result is a packed integer, which you can decode using unpack with the i (or I) format.

Here's an example to test whether Nagle's algorithm is enabled on a socket:

```
use Socket qw(:all);

defined(my $tcp = getprotobyname("tcp"))
    or die "Could not determine the protocol number for tcp";
# my $tcp = IPPROTO_TCP; # Alternative
my $packed = getsockopt($socket, $tcp, TCP_NODELAY)
    or die "getsockopt TCP_NODELAY: $!";
my $nodelay = unpack("I", $packed);
```

```
print "Nagle's algorithm is turned ",
       $nodelay ? "off\n" : "on\n";
```

Portability issues: ⟨undefined⟩ [perlport getsockopt], page ⟨undefined⟩.

glob EXPR
glob

In list context, returns a (possibly empty) list of filename expansions on the value of EXPR such as the standard Unix shell `/bin/csh` would do. In scalar context, glob iterates through such filename expansions, returning undef when the list is exhausted. This is the internal function implementing the `<*.c>` operator, but you can use it directly. If EXPR is omitted, `$_` is used. The `<*.c>` operator is discussed in more detail in Section 48.2.33 [perlop I/O Operators], page 844.

Note that `glob` splits its arguments on whitespace and treats each segment as separate pattern. As such, `glob("*.c *.h")` matches all files with a `.c` or `.h` extension. The expression `glob(".* *")` matches all files in the current working directory. If you want to glob filenames that might contain whitespace, you'll have to use extra quotes around the spacey filename to protect it. For example, to glob filenames that have an `e` followed by a space followed by an `f`, use either of:

```
@spacies = <"*e f*">;
@spacies = glob '"*e f*"';
@spacies = glob q("*e f*");
```

If you had to get a variable through, you could do this:

```
@spacies = glob "'*${var}e f*'";
@spacies = glob qq("*${var}e f*");
```

If non-empty braces are the only wildcard characters used in the `glob`, no filenames are matched, but potentially many strings are returned. For example, this produces nine strings, one for each pairing of fruits and colors:

```
@many =  glob "{apple,tomato,cherry}={green,yellow,red}";
```

This operator is implemented using the standard `File::Glob` extension. See `File-Glob` for details, including `bsd_glob` which does not treat whitespace as a pattern separator.

Portability issues: [perlport glob], page 976.

gmtime EXPR
gmtime

Works just like [localtime], page 401 but the returned values are localized for the standard Greenwich time zone.

Note: When called in list context, $isdst, the last value returned by gmtime, is always `0`. There is no Daylight Saving Time in GMT.

Portability issues: [perlport gmtime], page 976.

goto LABEL
goto EXPR
goto &NAME

The goto LABEL form finds the statement labeled with LABEL and resumes execution there. It can't be used to get out of a block or subroutine given to sort. It can be used to go almost anywhere else within the dynamic scope, including out of subroutines, but it's usually better to use some other construct such as last or die. The author of Perl has never felt the need to use this form of goto (in Perl, that is; C is another matter). (The difference is that C does not offer named loops combined with loop control. Perl does, and this replaces most structured uses of goto in other languages.)

The goto EXPR form expects to evaluate EXPR to a code reference or a label name. If it evaluates to a code reference, it will be handled like goto &NAME, below. This is especially useful for implementing tail recursion via goto __SUB__.

If the expression evaluates to a label name, its scope will be resolved dynamically. This allows for computed gotos per FORTRAN, but isn't necessarily recommended if you're optimizing for maintainability:

```
goto ("FOO", "BAR", "GLARCH")[$i];
```

As shown in this example, goto EXPR is exempt from the "looks like a function" rule. A pair of parentheses following it does not (necessarily) delimit its argument. goto("NE")."XT" is equivalent to goto NEXT. Also, unlike most named operators, this has the same precedence as assignment.

Use of goto LABEL or goto EXPR to jump into a construct is deprecated and will issue a warning. Even then, it may not be used to go into any construct that requires initialization, such as a subroutine or a foreach loop. It also can't be used to go into a construct that is optimized away.

The goto &NAME form is quite different from the other forms of goto. In fact, it isn't a goto in the normal sense at all, and doesn't have the stigma associated with other gotos. Instead, it exits the current subroutine (losing any changes set by local()) and immediately calls in its place the named subroutine using the current value of @_. This is used by AUTOLOAD subroutines that wish to load another subroutine and then pretend that the other subroutine had been called in the first place (except that any modifications to @_ in the current subroutine are propagated to the other subroutine.) After the goto, not even caller will be able to tell that this routine was called first.

NAME needn't be the name of a subroutine; it can be a scalar variable containing a code reference or a block that evaluates to a code reference.

grep BLOCK LIST
grep EXPR,LIST

This is similar in spirit to, but not the same as, grep(1) and its relatives. In particular, it is not limited to using regular expressions.

Evaluates the BLOCK or EXPR for each element of LIST (locally setting $_ to each element) and returns the list value consisting of those elements for which the expression evaluated to true. In scalar context, returns the number of times the expression was true.

```
@foo = grep(!/^#/, @bar);    # weed out comments
```

or equivalently,

```
@foo = grep {!/^#/} @bar;    # weed out comments
```

Note that `$_` is an alias to the list value, so it can be used to modify the elements of the LIST. While this is useful and supported, it can cause bizarre results if the elements of LIST are not variables. Similarly, grep returns aliases into the original list, much as a for loop's index variable aliases the list elements. That is, modifying an element of a list returned by grep (for example, in a `foreach`, `map` or another `grep`) actually modifies the element in the original list. This is usually something to be avoided when writing clear code.

If `$_` is lexical in the scope where the `grep` appears (because it has been declared with the deprecated `my $_` construct) then, in addition to being locally aliased to the list elements, `$_` keeps being lexical inside the block; i.e., it can't be seen from the outside, avoiding any potential side-effects.

See also ⟨undefined⟩ [map], page ⟨undefined⟩ for a list composed of the results of the BLOCK or EXPR.

hex EXPR
hex

Interprets EXPR as a hex string and returns the corresponding value. (To convert strings that might start with either `0`, `0x`, or `0b`, see [oct], page 406.) If EXPR is omitted, uses `$_`.

```
print hex '0xAf'; # prints '175'
print hex 'aF';   # same
```

Hex strings may only represent integers. Strings that would cause integer overflow trigger a warning. Leading whitespace is not stripped, unlike oct(). To present something as hex, look into [printf], page 429, ⟨undefined⟩ [sprintf], page ⟨undefined⟩, and ⟨undefined⟩ [unpack], page ⟨undefined⟩.

import LIST

There is no builtin `import` function. It is just an ordinary method (subroutine) defined (or inherited) by modules that wish to export names to another module. The `use` function calls the `import` method for the package used. See also ⟨undefined⟩ [use], page ⟨undefined⟩, Section 40.1 [perlmod NAME], page 732, and `Exporter`.

index STR,SUBSTR,POSITION
index STR,SUBSTR

The index function searches for one string within another, but without the wildcard-like behavior of a full regular-expression pattern match. It returns the position of the first occurrence of SUBSTR in STR at or after POSITION. If POSITION is omitted, starts searching from the beginning of the string. POSITION before the beginning of the string or after its end is treated as if it were the beginning or the end, respectively. POSITION and the return value are based at zero. If the substring is not found, `index` returns -1.

int EXPR

int

Returns the integer portion of EXPR. If EXPR is omitted, uses `$_`. You should not use this function for rounding: one because it truncates towards `0`, and two because machine representations of floating-point numbers can sometimes produce counterintuitive results. For example, `int(-6.725/0.025)` produces -268 rather than the correct -269; that's because it's really more like -268.99999999999994315658 instead. Usually, the `sprintf`, `printf`, or the `POSIX::floor` and `POSIX::ceil` functions will serve you better than will int().

ioctl FILEHANDLE,FUNCTION,SCALAR

Implements the ioctl(2) function. You'll probably first have to say

```
require "sys/ioctl.ph";  # probably in
                         # $Config{archlib}/sys/ioctl.ph
```

to get the correct function definitions. If `sys/ioctl.ph` doesn't exist or doesn't have the correct definitions you'll have to roll your own, based on your C header files such as `<sys/ioctl.h>`. (There is a Perl script called **h2ph** that comes with the Perl kit that may help you in this, but it's nontrivial.) SCALAR will be read and/or written depending on the FUNCTION; a C pointer to the string value of SCALAR will be passed as the third argument of the actual `ioctl` call. (If SCALAR has no string value but does have a numeric value, that value will be passed rather than a pointer to the string value. To guarantee this to be true, add a `0` to the scalar before using it.) The `pack` and `unpack` functions may be needed to manipulate the values of structures used by `ioctl`.

The return value of `ioctl` (and `fcntl`) is as follows:

```
if OS returns:      then Perl returns:
    -1               undefined value
     0              string "0 but true"
anything else           that number
```

Thus Perl returns true on success and false on failure, yet you can still easily determine the actual value returned by the operating system:

```
$retval = ioctl(...) || -1;
printf "System returned %d\n", $retval;
```

The special string `"0 but true"` is exempt from **-w** complaints about improper numeric conversions.

Portability issues: ⟨undefined⟩ [perlport ioctl], page ⟨undefined⟩.

join EXPR,LIST

Joins the separate strings of LIST into a single string with fields separated by the value of EXPR, and returns that new string. Example:

```
$rec = join(':', $login,$passwd,$uid,$gid,$gcos,$home,$shell);
```

Beware that unlike `split`, `join` doesn't take a pattern as its first argument. Compare [split], page 453.

keys HASH
keys ARRAY
keys EXPR

Called in list context, returns a list consisting of all the keys of the named hash, or in Perl 5.12 or later only, the indices of an array. Perl releases prior to 5.12 will produce a syntax error if you try to use an array argument. In scalar context, returns the number of keys or indices.

Hash entries are returned in an apparently random order. The actual random order is specific to a given hash; the exact same series of operations on two hashes may result in a different order for each hash. Any insertion into the hash may change the order, as will any deletion, with the exception that the most recent key returned by `each` or `keys` may be deleted without changing the order. So long as a given hash is unmodified you may rely on `keys`, `values` and `each` to repeatedly return the same order as each other. See Section 70.4.9 [perlsec Algorithmic Complexity Attacks], page 1205 for details on why hash order is randomized. Aside from the guarantees provided here the exact details of Perl's hash algorithm and the hash traversal order are subject to change in any release of Perl. Tied hashes may behave differently to Perl's hashes with respect to changes in order on insertion and deletion of items.

As a side effect, calling keys() resets the internal iterator of the HASH or ARRAY (see ⟨undefined⟩ [each], page ⟨undefined⟩). In particular, calling keys() in void context resets the iterator with no other overhead.

Here is yet another way to print your environment:

```
@keys = keys %ENV;
@values = values %ENV;
while (@keys) {
    print pop(@keys), '=', pop(@values), "\n";
}
```

or how about sorted by key:

```
foreach $key (sort(keys %ENV)) {
    print $key, '=', $ENV{$key}, "\n";
}
```

The returned values are copies of the original keys in the hash, so modifying them will not affect the original hash. Compare ⟨undefined⟩ [values], page ⟨undefined⟩.

To sort a hash by value, you'll need to use a `sort` function. Here's a descending numeric sort of a hash by its values:

```
foreach $key (sort { $hash{$b} <=> $hash{$a} } keys %hash) {
    printf "%4d %s\n", $hash{$key}, $key;
}
```

Used as an lvalue, `keys` allows you to increase the number of hash buckets allocated for the given hash. This can gain you a measure of efficiency if you know the hash is going to get big. (This is similar to pre-extending an array by assigning a larger number to $#array.) If you say

```
keys %hash = 200;
```

then `%hash` will have at least 200 buckets allocated for it–256 of them, in fact, since it rounds up to the next power of two. These buckets will be retained even if you do `%hash = ()`, use `undef %hash` if you want to free the storage while `%hash` is still in scope. You can't shrink the number of buckets allocated for the hash using `keys` in this way (but you needn't worry about doing this by accident, as trying has no effect). `keys @array` in an lvalue context is a syntax error.

Starting with Perl 5.14, `keys` can take a scalar EXPR, which must contain a reference to an unblessed hash or array. The argument will be dereferenced automatically. This aspect of `keys` is considered highly experimental. The exact behaviour may change in a future version of Perl.

```
for (keys $hashref) { ... }
for (keys $obj->get_arrayref) { ... }
```

To avoid confusing would-be users of your code who are running earlier versions of Perl with mysterious syntax errors, put this sort of thing at the top of your file to signal that your code will work *only* on Perls of a recent vintage:

```
use 5.012;  # so keys/values/each work on arrays
use 5.014;  # so keys/values/each work on scalars (experimental)
```

See also `each`, `values`, and `sort`.

kill SIGNAL, LIST
kill SIGNAL

Sends a signal to a list of processes. Returns the number of arguments that were successfully used to signal (which is not necessarily the same as the number of processes actually killed, e.g. where a process group is killed).

```
$cnt = kill 'HUP', $child1, $child2;
kill 'KILL', @goners;
```

SIGNAL may be either a signal name (a string) or a signal number. A signal name may start with a `SIG` prefix, thus `FOO` and `SIGFOO` refer to the same signal. The string form of SIGNAL is recommended for portability because the same signal may have different numbers in different operating systems.

A list of signal names supported by the current platform can be found in `$Config{sig_name}`, which is provided by the `Config` module. See `Config` for more details.

A negative signal name is the same as a negative signal number, killing process groups instead of processes. For example, `kill '-KILL', $pgrp` and `kill -9, $pgrp` will send `SIGKILL` to the entire process group specified. That means you usually want to use positive not negative signals.

If SIGNAL is either the number 0 or the string `ZERO` (or `SIGZERO`), no signal is sent to the process, but `kill` checks whether it's *possible* to send a signal to it (that means, to be brief, that the process is owned by the same user, or we are the super-user). This is useful to check that a child process is still alive (even if only as a zombie) and hasn't changed its UID. See Section 56.1 [perlport NAME], page 951 for notes on the portability of this construct.

The behavior of kill when a *PROCESS* number is zero or negative depends on the operating system. For example, on POSIX-conforming systems, zero will signal the current process group, -1 will signal all processes, and any other negative PROCESS number will act as a negative signal number and kill the entire process group specified.

If both the SIGNAL and the PROCESS are negative, the results are undefined. A warning may be produced in a future version.

See Section 36.3 [perlipc Signals], page 667 for more details.

On some platforms such as Windows where the fork() system call is not available, Perl can be built to emulate fork() at the interpreter level. This emulation has limitations related to kill that have to be considered, for code running on Windows and in code intended to be portable.

See Section 23.1 [perlfork NAME], page 337 for more details.

If there is no *LIST* of processes, no signal is sent, and the return value is 0. This form is sometimes used, however, because it causes tainting checks to be run. But see Section 70.4.2 [perlsec Laundering and Detecting Tainted Data], page 1200.

Portability issues: [perlport kill], page 976.

last LABEL
last EXPR
last

The `last` command is like the `break` statement in C (as used in loops); it immediately exits the loop in question. If the LABEL is omitted, the command refers to the innermost enclosing loop. The `last EXPR` form, available starting in Perl 5.18.0, allows a label name to be computed at run time, and is otherwise identical to `last LABEL`. The `continue` block, if any, is not executed:

```
LINE: while (<STDIN>) {
    last LINE if /^$/;  # exit when done with header
    #...
}
```

`last` cannot be used to exit a block that returns a value such as `eval {}`, `sub {}`, or `do {}`, and should not be used to exit a grep() or map() operation.

Note that a block by itself is semantically identical to a loop that executes once. Thus `last` can be used to effect an early exit out of such a block.

See also [continue], page 366 for an illustration of how `last`, `next`, and `redo` work.

Unlike most named operators, this has the same precedence as assignment. It is also exempt from the looks-like-a-function rule, so `last ("foo")."bar"` will cause "bar" to be part of the argument to `last`.

lc EXPR

lc

Returns a lowercased version of EXPR. This is the internal function implementing the `\L` escape in double-quoted strings.

If EXPR is omitted, uses `$_`.

What gets returned depends on several factors:

If `use bytes` is in effect:

The results follow ASCII rules. Only the characters `A-Z` change, to `a-z` respectively.

Otherwise, if `use locale` for `LC_CTYPE` is in effect:

Respects current `LC_CTYPE` locale for code points < 256; and uses Unicode rules for the remaining code points (this last can only happen if the UTF8 flag is also set). See Section 38.1 [perllocale NAME], page 701.

Starting in v5.20, Perl uses full Unicode rules if the locale is UTF-8. Otherwise, there is a deficiency in this scheme, which is that case changes that cross the 255/256 boundary are not well-defined. For example, the lower case of LATIN CAPITAL LETTER SHARP S (U+1E9E) in Unicode rules is U+00DF (on ASCII platforms). But under `use locale` (prior to v5.20 or not a UTF-8 locale), the lower case of U+1E9E is itself, because 0xDF may not be LATIN SMALL LETTER SHARP S in the current locale, and Perl has no way of knowing if that character even exists in the locale, much less what code point it is. Perl returns a result that is above 255 (almost always the input character unchanged, for all instances (and there aren't many) where the 255/256 boundary would otherwise be crossed; and starting in v5.22, it raises a [locale], page 148 warning.

Otherwise, If EXPR has the UTF8 flag set:

Unicode rules are used for the case change.

Otherwise, if `use feature 'unicode_strings'` or `use locale ':not_characters'` is in effect:

Unicode rules are used for the case change.

Otherwise:

ASCII rules are used for the case change. The lowercase of any character outside the ASCII range is the character itself.

lcfirst EXPR
lcfirst

Returns the value of EXPR with the first character lowercased. This is the internal function implementing the `\l` escape in double-quoted strings.

If EXPR is omitted, uses `$_`.

This function behaves the same way under various pragmata, such as in a locale, as [lc], page 399 does.

length EXPR
length

Returns the length in *characters* of the value of EXPR. If EXPR is omitted, returns the length of `$_`. If EXPR is undefined, returns `undef`.

This function cannot be used on an entire array or hash to find out how many elements these have. For that, use `scalar @array` and `scalar keys %hash`, respectively.

Like all Perl character operations, length() normally deals in logical characters, not physical bytes. For how many bytes a string encoded as UTF-8 would take up, use `length(Encode::encode_utf8(EXPR))` (you'll have to `use Encode` first). See `Encode` and Section 81.1 [perlunicode NAME], page 1317.

__LINE__

A special token that compiles to the current line number.

link OLDFILE,NEWFILE

Creates a new filename linked to the old filename. Returns true for success, false otherwise.

Portability issues: [perlport link], page 976.

listen SOCKET,QUEUESIZE

Does the same thing that the listen(2) system call does. Returns true if it succeeded, false otherwise. See the example in Section 36.6 [perlipc Sockets: Client/Server Communication], page 682.

local EXPR

You really probably want to be using `my` instead, because `local` isn't what most people think of as "local". See Section 73.3.2 [perlsub Private Variables via my()], page 1223 for details.

A local modifies the listed variables to be local to the enclosing block, file, or eval. If more than one value is listed, the list must be placed in parentheses. See Section 73.3.4 [perlsub Temporary Values via local()], page 1228 for details, including issues with tied arrays and hashes.

The `delete local EXPR` construct can also be used to localize the deletion of array/hash elements to the current block. See Section 73.3.4.5 [perlsub Localized deletion of elements of composite types], page 1230.

localtime EXPR
localtime

Converts a time as returned by the time function to a 9-element list with the time analyzed for the local time zone. Typically used as follows:

```
#  0    1    2     3     4    5     6     7     8
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
                                             localtime(time);
```

All list elements are numeric and come straight out of the C 'struct tm'. `$sec`, `$min`, and `$hour` are the seconds, minutes, and hours of the specified time.

`$mday` is the day of the month and `$mon` the month in the range `0..11`, with 0 indicating January and 11 indicating December. This makes it easy to get a month name from a list:

```
my @abbr = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
print "$abbr[$mon] $mday";
# $mon=9, $mday=18 gives "Oct 18"
```

`$year` contains the number of years since 1900. To get a 4-digit year write:

```
$year += 1900;
```

To get the last two digits of the year (e.g., "01" in 2001) do:

```
    $year = sprintf("%02d", $year % 100);
```

`$wday` is the day of the week, with 0 indicating Sunday and 3 indicating Wednesday. `$yday` is the day of the year, in the range `0..364` (or `0..365` in leap years.)

`$isdst` is true if the specified time occurs during Daylight Saving Time, false otherwise.

If EXPR is omitted, `localtime()` uses the current time (as returned by time(3)).

In scalar context, `localtime()` returns the ctime(3) value:

```
    $now_string = localtime;  # e.g., "Thu Oct 13 04:54:34 1994"
```

The format of this scalar value is **not** locale-dependent but built into Perl. For GMT instead of local time use the [gmtime], page 393 builtin. See also the `Time::Local` module (for converting seconds, minutes, hours, and such back to the integer value returned by time()), and the `POSIX` module's strftime(3) and mktime(3) functions.

To get somewhat similar but locale-dependent date strings, set up your locale environment variables appropriately (please see Section 38.1 [perllocale NAME], page 701) and try for example:

```
    use POSIX qw(strftime);
    $now_string = strftime "%a %b %e %H:%M:%S %Y", localtime;
    # or for GMT formatted appropriately for your locale:
    $now_string = strftime "%a %b %e %H:%M:%S %Y", gmtime;
```

Note that the `%a` and `%b`, the short forms of the day of the week and the month of the year, may not necessarily be three characters wide.

The `Time-gmtime` and `Time-localtime` modules provide a convenient, by-name access mechanism to the gmtime() and localtime() functions, respectively.

For a comprehensive date and time representation look at the `DateTime` module on CPAN.

Portability issues: [perlport localtime], page 977.

lock THING

This function places an advisory lock on a shared variable or referenced object contained in *THING* until the lock goes out of scope.

The value returned is the scalar itself, if the argument is a scalar, or a reference, if the argument is a hash, array or subroutine.

lock() is a `"weak keyword"` : this means that if you've defined a function by this name (before any calls to it), that function will be called instead. If you are not under `use threads::shared` this does nothing. See `threads-shared`.

log EXPR

log

Returns the natural logarithm (base e) of EXPR. If EXPR is omitted, returns the log of `$_`. To get the log of another base, use basic algebra: The base-N log of a number is equal to the natural log of that number divided by the natural log of N. For example:

```
sub log10 {
    my $n = shift;
    return log($n)/log(10);
}
```

See also [exp], page 383 for the inverse operation.

lstat FILEHANDLE
lstat EXPR
lstat DIRHANDLE
lstat

Does the same thing as the `stat` function (including setting the special `_` filehandle) but stats a symbolic link instead of the file the symbolic link points to. If symbolic links are unimplemented on your system, a normal `stat` is done. For much more detailed information, please see the documentation for `stat`.

If EXPR is omitted, stats `$_`.

Portability issues: [perlport lstat], page 977.

m//

The match operator. See Section 48.2.30 [perlop Regexp Quote-Like Operators], page 823.

map BLOCK LIST
map EXPR,LIST

Evaluates the BLOCK or EXPR for each element of LIST (locally setting `$_` to each element) and returns the list value composed of the results of each such evaluation. In scalar context, returns the total number of elements so generated. Evaluates BLOCK or EXPR in list context, so each element of LIST may produce zero, one, or more elements in the returned value.

```
@chars = map(chr, @numbers);
```

translates a list of numbers to the corresponding characters.

```
my @squares = map { $_ * $_ } @numbers;
```

translates a list of numbers to their squared values.

```
my @squares = map { $_ > 5 ? ($_ * $_) : () } @numbers;
```

shows that number of returned elements can differ from the number of input elements. To omit an element, return an empty list (). This could also be achieved by writing

```
my @squares = map { $_ * $_ } grep { $_ > 5 } @numbers;
```

which makes the intention more clear.

Map always returns a list, which can be assigned to a hash such that the elements become key/value pairs. See Section 11.1 [perldata NAME], page 70 for more details.

```
%hash = map { get_a_key_for($_) => $_ } @array;
```

is just a funny way to write

```
%hash = ();
foreach (@array) {
    $hash{get_a_key_for($_)} = $_;
}
```

Note that $_ is an alias to the list value, so it can be used to modify the elements of the LIST. While this is useful and supported, it can cause bizarre results if the elements of LIST are not variables. Using a regular foreach loop for this purpose would be clearer in most cases. See also ⟨undefined⟩ [grep], page ⟨undefined⟩ for an array composed of those items of the original list for which the BLOCK or EXPR evaluates to true.

If $_ is lexical in the scope where the map appears (because it has been declared with the deprecated my $_ construct), then, in addition to being locally aliased to the list elements, $_ keeps being lexical inside the block; that is, it can't be seen from the outside, avoiding any potential side-effects.

{ starts both hash references and blocks, so map { ... could be either the start of map BLOCK LIST or map EXPR, LIST. Because Perl doesn't look ahead for the closing } it has to take a guess at which it's dealing with based on what it finds just after the {. Usually it gets it right, but if it doesn't it won't realize something is wrong until it gets to the } and encounters the missing (or unexpected) comma. The syntax error will be reported close to the }, but you'll need to change something near the { such as using a unary + or semicolon to give Perl some help:

```
    %hash = map {  "\L$_" => 1  } @array # perl guesses EXPR. wrong
    %hash = map { +"\L$_" => 1  } @array # perl guesses BLOCK. right
    %hash = map {; "\L$_" => 1  } @array # this also works
    %hash = map { ("\L$_" => 1) } @array # as does this
    %hash = map {  lc($_) => 1  } @array # and this.
    %hash = map +( lc($_) => 1 ), @array # this is EXPR and works!

    %hash = map  ( lc($_), 1 ),   @array # evaluates to (1, @array)
```

or to force an anon hash constructor use +{:

```
    @hashes = map +{ lc($_) => 1 }, @array # EXPR, so needs
                                           # comma at end
```

to get a list of anonymous hashes each with only one entry apiece.

mkdir FILENAME,MASK
mkdir FILENAME
mkdir

Creates the directory specified by FILENAME, with permissions specified by MASK (as modified by umask). If it succeeds it returns true; otherwise it returns false and sets $! (errno). MASK defaults to 0777 if omitted, and FILENAME defaults to $_ if omitted.

In general, it is better to create directories with a permissive MASK and let the user modify that with their umask than it is to supply a restrictive MASK and give the user no way to be more permissive. The exceptions to this rule are when the file or directory should be kept private (mail files, for instance). The perlfunc(1) entry on umask discusses the choice of MASK in more detail.

Note that according to the POSIX 1003.1-1996 the FILENAME may have any number of trailing slashes. Some operating and filesystems do not get this right, so Perl automatically removes all trailing slashes to keep everyone happy.

To recursively create a directory structure, look at the `make_path` function of the `File-Path` module.

msgctl ID,CMD,ARG

Calls the System V IPC function msgctl(2). You'll probably have to say

```
use IPC::SysV;
```

first to get the correct constant definitions. If CMD is `IPC_STAT`, then ARG must be a variable that will hold the returned `msqid_ds` structure. Returns like `ioctl`: the undefined value for error, `"0 but true"` for zero, or the actual return value otherwise. See also Section 36.10 [perlipc SysV IPC], page 696 and the documentation for `IPC::SysV` and `IPC::Semaphore`.

Portability issues: [perlport msgctl], page 977.

msgget KEY,FLAGS

Calls the System V IPC function msgget(2). Returns the message queue id, or `undef` on error. See also Section 36.10 [perlipc SysV IPC], page 696 and the documentation for `IPC::SysV` and `IPC::Msg`.

Portability issues: [perlport msgget], page 977.

msgrcv ID,VAR,SIZE,TYPE,FLAGS

Calls the System V IPC function msgrcv to receive a message from message queue ID into variable VAR with a maximum message size of SIZE. Note that when a message is received, the message type as a native long integer will be the first thing in VAR, followed by the actual message. This packing may be opened with `unpack("l! a*")`. Taints the variable. Returns true if successful, false on error. See also Section 36.10 [perlipc SysV IPC], page 696 and the documentation for `IPC::SysV` and `IPC::SysV::Msg`.

Portability issues: [perlport msgrcv], page 977.

msgsnd ID,MSG,FLAGS

Calls the System V IPC function msgsnd to send the message MSG to the message queue ID. MSG must begin with the native long integer message type, be followed by the length of the actual message, and then finally the message itself. This kind of packing can be achieved with `pack("l! a*", $type, $message)`. Returns true if successful, false on error. See also the `IPC::SysV` and `IPC::SysV::Msg` documentation.

Portability issues: [perlport msgsnd], page 977.

my VARLIST
my TYPE VARLIST
my VARLIST : ATTRS
my TYPE VARLIST : ATTRS

A `my` declares the listed variables to be local (lexically) to the enclosing block, file, or `eval`. If more than one variable is listed, the list must be placed in parentheses.

The exact semantics and interface of TYPE and ATTRS are still evolving. TYPE may be a bareword, a constant declared with `use constant`, or `__PACKAGE__`. It is currently bound to the use of the `fields` pragma, and attributes are handled using the `attributes` pragma, or starting from Perl 5.8.0 also via the `Attribute::Handlers` module. See Section 73.3.2 [perlsub Private Variables via my()], page 1223 for details, and `fields`, `attributes`, and `Attribute-Handlers`.

Note that with a parenthesised list, `undef` can be used as a dummy placeholder, for example to skip assignment of initial values:

```
my ( undef, $min, $hour ) = localtime;
```

next LABEL
next EXPR
next

The `next` command is like the `continue` statement in C; it starts the next iteration of the loop:

```
LINE: while (<STDIN>) {
    next LINE if /^#/;  # discard comments
    #...
}
```

Note that if there were a `continue` block on the above, it would get executed even on discarded lines. If LABEL is omitted, the command refers to the innermost enclosing loop. The `next EXPR` form, available as of Perl 5.18.0, allows a label name to be computed at run time, being otherwise identical to `next LABEL`.

`next` cannot be used to exit a block which returns a value such as `eval {}`, `sub {}`, or `do {}`, and should not be used to exit a grep() or map() operation.

Note that a block by itself is semantically identical to a loop that executes once. Thus `next` will exit such a block early.

See also [continue], page 366 for an illustration of how `last`, `next`, and `redo` work.

Unlike most named operators, this has the same precedence as assignment. It is also exempt from the looks-like-a-function rule, so `next ("foo")."bar"` will cause "bar" to be part of the argument to `next`.

no MODULE VERSION LIST
no MODULE VERSION
no MODULE LIST
no MODULE
no VERSION

See the `use` function, of which `no` is the opposite.

oct EXPR

oct

Interprets EXPR as an octal string and returns the corresponding value. (If EXPR happens to start off with `0x`, interprets it as a hex string. If EXPR starts off with 0b, it is interpreted as a binary string. Leading whitespace is ignored in all three cases.) The following will handle decimal, binary, octal, and hex in standard Perl notation:

```
    $val = oct($val) if $val =~ /^0/;
```

If EXPR is omitted, uses $_. To go the other way (produce a number in octal), use sprintf() or printf():

```
    $dec_perms = (stat("filename"))[2] & 07777;
    $oct_perm_str = sprintf "%o", $perms;
```

The oct() function is commonly used when a string such as 644 needs to be converted into a file mode, for example. Although Perl automatically converts strings into numbers as needed, this automatic conversion assumes base 10.

Leading white space is ignored without warning, as too are any trailing non-digits, such as a decimal point (oct only handles non-negative integers, not negative integers or floating point).

open FILEHANDLE,EXPR
open FILEHANDLE,MODE,EXPR
open FILEHANDLE,MODE,EXPR,LIST
open FILEHANDLE,MODE,REFERENCE
open FILEHANDLE

Opens the file whose filename is given by EXPR, and associates it with FILEHANDLE.

Simple examples to open a file for reading:

```
    open(my $fh, "<", "input.txt")
        or die "cannot open < input.txt: $!";
```

and for writing:

```
    open(my $fh, ">", "output.txt")
        or die "cannot open > output.txt: $!";
```

(The following is a comprehensive reference to open(): for a gentler introduction you may consider Section 49.1 [perlopentut NAME], page 852.)

If FILEHANDLE is an undefined scalar variable (or array or hash element), a new filehandle is autovivified, meaning that the variable is assigned a reference to a newly allocated anonymous filehandle. Otherwise if FILEHANDLE is an expression, its value is the real filehandle. (This is considered a symbolic reference, so use strict "refs" should *not* be in effect.)

If three (or more) arguments are specified, the open mode (including optional encoding) in the second argument are distinct from the filename in the third. If MODE is < or nothing, the file is opened for input. If MODE is >, the file is opened for output, with existing files first being truncated ("clobbered") and nonexisting files newly created. If MODE is >>, the file is opened for appending, again being created if necessary.

You can put a + in front of the > or < to indicate that you want both read and write access to the file; thus +< is almost always preferred for read/write updates–the +> mode would clobber the file first. You can't usually use either read-write mode for updating textfiles, since they have variable-length records.

See the **-i** switch in Section 69.1 [perlrun NAME], page 1176 for a better approach. The file is created with permissions of `0666` modified by the process's `umask` value.

These various prefixes correspond to the fopen(3) modes of `r`, `r+`, `w`, `w+`, `a`, and `a+`.

In the one- and two-argument forms of the call, the mode and filename should be concatenated (in that order), preferably separated by white space. You can–but shouldn't–omit the mode in these forms when that mode is `<`. It is always safe to use the two-argument form of `open` if the filename argument is a known literal.

For three or more arguments if MODE is `|-`, the filename is interpreted as a command to which output is to be piped, and if MODE is `-|`, the filename is interpreted as a command that pipes output to us. In the two-argument (and one-argument) form, one should replace dash (`-`) with the command. See Section 36.5 [perlipc Using open() for IPC], page 674 for more examples of this. (You are not allowed to `open` to a command that pipes both in *and* out, but see `IPC-Open2`, `IPC-Open3`, and Section 36.5.6 [perlipc Bidirectional Communication with Another Process], page 680 for alternatives.)

In the form of pipe opens taking three or more arguments, if LIST is specified (extra arguments after the command name) then LIST becomes arguments to the command invoked if the platform supports it. The meaning of `open` with more than three arguments for non-pipe modes is not yet defined, but experimental "layers" may give extra LIST arguments meaning.

In the two-argument (and one-argument) form, opening `<-` or `-` opens STDIN and opening `>-` opens STDOUT.

You may (and usually should) use the three-argument form of open to specify I/O layers (sometimes referred to as "disciplines") to apply to the handle that affect how the input and output are processed (see `open` and `PerlIO` for more details). For example:

```
open(my $fh, "<:encoding(UTF-8)", "filename")
  || die "can't open UTF-8 encoded filename: $!";
```

opens the UTF8-encoded file containing Unicode characters; see Section 83.1 [perluniintro NAME], page 1352. Note that if layers are specified in the three-argument form, then default layers stored in ${^OPEN} (see Section 86.1 [perlvar NAME], page 1375; usually set by the **open** pragma or the switch **-CioD**) are ignored. Those layers will also be ignored if you specifying a colon with no name following it. In that case the default layer for the operating system (:raw on Unix, :crlf on Windows) is used.

Open returns nonzero on success, the undefined value otherwise. If the `open` involved a pipe, the return value happens to be the pid of the subprocess.

If you're running Perl on a system that distinguishes between text files and binary files, then you should check out ⟨undefined⟩ [binmode], page ⟨undefined⟩ for tips for dealing with this. The key distinction between systems that need `binmode` and those that don't is their text file formats. Systems like Unix, Mac

OS, and Plan 9, that end lines with a single character and encode that character in C as `"\n"` do not need `binmode`. The rest need it.

When opening a file, it's seldom a good idea to continue if the request failed, so `open` is frequently used with `die`. Even if `die` won't do what you want (say, in a CGI script, where you want to format a suitable error message (but there are modules that can help with that problem)) always check the return value from opening a file.

The filehandle will be closed when its reference count reaches zero. If it is a lexically scoped variable declared with `my`, that usually means the end of the enclosing scope. However, this automatic close does not check for errors, so it is better to explicitly close filehandles, especially those used for writing:

```
close($handle)
   || warn "close failed: $!";
```

An older style is to use a bareword as the filehandle, as

```
open(FH, "<", "input.txt")
   or die "cannot open < input.txt: $!";
```

Then you can use `FH` as the filehandle, in `close FH` and `<FH>` and so on. Note that it's a global variable, so this form is not recommended in new code.

As a shortcut a one-argument call takes the filename from the global scalar variable of the same name as the filehandle:

```
$ARTICLE = 100;
open(ARTICLE) or die "Can't find article $ARTICLE: $!\n";
```

Here `$ARTICLE` must be a global (package) scalar variable - not one declared with `my` or `state`.

As a special case the three-argument form with a read/write mode and the third argument being `undef`:

```
open(my $tmp, "+>", undef) or die ...
```

opens a filehandle to an anonymous temporary file. Also using `+<` works for symmetry, but you really should consider writing something to the temporary file first. You will need to seek() to do the reading.

Perl is built using PerlIO by default; Unless you've changed this (such as building Perl with `Configure -Uuseperlio`), you can open filehandles directly to Perl scalars via:

```
open($fh, ">", \$variable) || ..
```

To (re)open `STDOUT` or `STDERR` as an in-memory file, close it first:

```
close STDOUT;
open(STDOUT, ">", \$variable)
    or die "Can't open STDOUT: $!";
```

General examples:

```
open(LOG, ">>/usr/spool/news/twitlog");  # (log is reserved)
# if the open fails, output is discarded

open(my $dbase, "+<", "dbase.mine")      # open for update
```

```
        or die "Can’t open ’dbase.mine’ for update: $!";

    open(my $dbase, "+<dbase.mine")          # ditto
        or die "Can’t open ’dbase.mine’ for update: $!";

    open(ARTICLE, "-|", "caesar <$article")  # decrypt article
        or die "Can’t start caesar: $!";

    open(ARTICLE, "caesar <$article |")      # ditto
        or die "Can’t start caesar: $!";

    open(EXTRACT, "|sort >Tmp$$")            # $$ is our process id
        or die "Can’t start sort: $!";

    # in-memory files
    open(MEMORY, ">", \$var)
        or die "Can’t open memory file: $!";
    print MEMORY "foo!\n";              # output will appear in $var

    # process argument list of files along with any includes

    foreach $file (@ARGV) {
        process($file, "fh00");
    }

    sub process {
        my($filename, $input) = @_;
        $input++;    # this is a string increment
        unless (open($input, "<", $filename)) {
            print STDERR "Can’t open $filename: $!\n";
            return;
        }

        local $_;
        while (<$input>) {    # note use of indirection
            if (/^#include "(.*)"/) {
                process($1, $input);
                next;
            }
            #...          # whatever
        }
    }
```

See Section 35.1 [perliol NAME], page 651 for detailed info on PerlIO.

You may also, in the Bourne shell tradition, specify an EXPR beginning with `>&`, in which case the rest of the string is interpreted as the name of a filehandle (or file descriptor, if numeric) to be duped (as `dup(2)`) and opened. You may use `&` after `>`, `>>`, `<`, `+>`, `+>>`, and `+<`. The mode you specify should match the mode of the original filehandle. (Duping a filehandle does not take into account any existing contents of IO buffers.) If you use the three-argument form, then you can pass either a number, the name of a filehandle, or the normal "reference to a glob".

Here is a script that saves, redirects, and restores `STDOUT` and `STDERR` using various methods:

```
    #!/usr/bin/perl
    open(my $oldout, ">&STDOUT")     or die "Can't dup STDOUT: $!";
    open(OLDERR,     ">&", \*STDERR) or die "Can't dup STDERR: $!";

    open(STDOUT, '>', "foo.out") or die "Can't redirect STDOUT: $!";
    open(STDERR, ">&STDOUT")    or die "Can't dup STDOUT: $!";

    select STDERR; $| = 1;  # make unbuffered
    select STDOUT; $| = 1;  # make unbuffered

    print STDOUT "stdout 1\n";  # this works for
    print STDERR "stderr 1\n";  # subprocesses too

    open(STDOUT, ">&", $oldout) or die "Can't dup \$oldout: $!";
    open(STDERR, ">&OLDERR")   or die "Can't dup OLDERR: $!";

    print STDOUT "stdout 2\n";
    print STDERR "stderr 2\n";
```

If you specify `'<&=X'`, where `X` is a file descriptor number or a filehandle, then Perl will do an equivalent of C's `fdopen` of that file descriptor (and not call `dup(2)`); this is more parsimonious of file descriptors. For example:

```
    # open for input, reusing the fileno of $fd
    open(FILEHANDLE, "<&=$fd")
```

or

```
    open(FILEHANDLE, "<&=", $fd)
```

or

```
    # open for append, using the fileno of OLDFH
    open(FH, ">>&=", OLDFH)
```

or

```
    open(FH, ">>&=OLDFH")
```

Being parsimonious on filehandles is also useful (besides being parsimonious) for example when something is dependent on file descriptors, like for example locking using flock(). If you do just `open(A, ">>&B")`, the filehandle A will not have the same file descriptor as B, and therefore flock(A) will not flock(B) nor vice versa. But with `open(A, ">>&=B")`, the filehandles will share the same underlying system file descriptor.

Note that under Perls older than 5.8.0, Perl uses the standard C library's' fdopen() to implement the `=` functionality. On many Unix systems, fdopen()

fails when file descriptors exceed a certain value, typically 255. For Perls 5.8.0 and later, PerlIO is (most often) the default.

You can see whether your Perl was built with PerlIO by running `perl -V` and looking for the `useperlio=` line. If `useperlio` is `define`, you have PerlIO; otherwise you don't.

If you open a pipe on the command `-` (that is, specify either `|-` or `-|` with the one- or two-argument forms of `open`), an implicit `fork` is done, so `open` returns twice: in the parent process it returns the pid of the child process, and in the child process it returns (a defined) `0`. Use `defined($pid)` or `//` to determine whether the open was successful.

For example, use either

```
$child_pid = open(FROM_KID, "-|")   // die "can't fork: $!";
```

or

```
$child_pid = open(TO_KID,   "|-")   // die "can't fork: $!";
```

followed by

```
if ($child_pid) {
    # am the parent:
    # either write TO_KID or else read FROM_KID
    ...
   waitpid $child_pid, 0;
} else {
    # am the child; use STDIN/STDOUT normally
    ...
    exit;
}
```

The filehandle behaves normally for the parent, but I/O to that filehandle is piped from/to the STDOUT/STDIN of the child process. In the child process, the filehandle isn't opened–I/O happens from/to the new STDOUT/STDIN. Typically this is used like the normal piped open when you want to exercise more control over just how the pipe command gets executed, such as when running setuid and you don't want to have to scan shell commands for metacharacters.

The following blocks are more or less equivalent:

```
open(FOO, "|tr '[a-z]' '[A-Z]'");
open(FOO, "|-", "tr '[a-z]' '[A-Z]'");
open(FOO, "|-") || exec 'tr', '[a-z]', '[A-Z]';
open(FOO, "|-", "tr", '[a-z]', '[A-Z]');

open(FOO, "cat -n '$file'|");
open(FOO, "-|", "cat -n '$file'");
open(FOO, "-|") || exec "cat", "-n", $file;
open(FOO, "-|", "cat", "-n", $file);
```

The last two examples in each block show the pipe as "list form", which is not yet supported on all platforms. A good rule of thumb is that if your platform has a real `fork()` (in other words, if your platform is Unix, including Linux and MacOS X), you can use the list form. You would want to use the list form of the pipe so you can pass literal arguments to the command without risk of the shell interpreting any shell metacharacters in them. However, this also bars you from opening pipes to commands that intentionally contain shell metacharacters, such as:

```
open(FOO, "|cat -n | expand -4 | lpr")
    // die "Can't open pipeline to lpr: $!";
```

See Section 36.5.4 [perlipc Safe Pipe Opens], page 676 for more examples of this.

Perl will attempt to flush all files opened for output before any operation that may do a fork, but this may not be supported on some platforms (see Section 56.1 [perlport NAME], page 951). To be safe, you may need to set `$|` ($AUTOFLUSH in English) or call the `autoflush()` method of `IO::Handle` on any open handles.

On systems that support a close-on-exec flag on files, the flag will be set for the newly opened file descriptor as determined by the value of `$^F`. See [perlvar $^F], page 1381.

Closing any piped filehandle causes the parent process to wait for the child to finish, then returns the status value in `$?` and `${^CHILD_ERROR_NATIVE}`.

The filename passed to the one- and two-argument forms of open() will have leading and trailing whitespace deleted and normal redirection characters honored. This property, known as "magic open", can often be used to good effect. A user could specify a filename of "`rsh cat file |`", or you could change certain filenames as needed:

```
$filename =~ s/(.*\.gz)\s*$/gzip -dc < $1|/;
open(FH, $filename) or die "Can't open $filename: $!";
```

Use the three-argument form to open a file with arbitrary weird characters in it,

```
open(FOO, "<", $file)
    || die "can't open < $file: $!";
```

otherwise it's necessary to protect any leading and trailing whitespace:

```
$file =~ s#^(\s)#./$1#;
open(FOO, "< $file\0")
    || die "open failed: $!";
```

(this may not work on some bizarre filesystems). One should conscientiously choose between the *magic* and *three-argument* form of open():

```
open(IN, $ARGV[0]) || die "can't open $ARGV[0]: $!";
```

will allow the user to specify an argument of the form "`rsh cat file |`", but will not work on a filename that happens to have a trailing space, while

```
open(IN, "<", $ARGV[0])
    || die "can't open < $ARGV[0]: $!";
```

will have exactly the opposite restrictions.

If you want a "real" C `open` (see open(2) on your system), then you should use the `sysopen` function, which involves no such magic (but may use subtly different filemodes than Perl open(), which is mapped to C fopen()). This is another way to protect your filenames from interpretation. For example:

```
use IO::Handle;
sysopen(HANDLE, $path, O_RDWR|O_CREAT|O_EXCL)
    or die "sysopen $path: $!";
$oldfh = select(HANDLE); $| = 1; select($oldfh);
print HANDLE "stuff $$\n";
seek(HANDLE, 0, 0);
print "File contains: ", <HANDLE>;
```

See ⟨undefined⟩ [seek], page ⟨undefined⟩ for some details about mixing reading and writing.

Portability issues: [perlport open], page 977.

opendir DIRHANDLE,EXPR

Opens a directory named EXPR for processing by `readdir`, `telldir`, `seekdir`, `rewinddir`, and `closedir`. Returns true if successful. DIRHANDLE may be an expression whose value can be used as an indirect dirhandle, usually the real dirhandle name. If DIRHANDLE is an undefined scalar variable (or array or hash element), the variable is assigned a reference to a new anonymous dirhandle; that is, it's autovivified. DIRHANDLEs have their own namespace separate from FILEHANDLEs.

See the example at `readdir`.

ord EXPR
ord

Returns the numeric value of the first character of EXPR. If EXPR is an empty string, returns 0. If EXPR is omitted, uses `$_`. (Note *character*, not byte.)

For the reverse, see [chr], page 364. See Section 81.1 [perlunicode NAME], page 1317 for more about Unicode.

our VARLIST
our TYPE VARLIST
our VARLIST : ATTRS
our TYPE VARLIST : ATTRS

`our` makes a lexical alias to a package (i.e. global) variable of the same name in the current package for use within the current lexical scope.

`our` has the same scoping rules as `my` or `state`, meaning that it is only valid within a lexical scope. Unlike `my` and `state`, which both declare new (lexical) variables, `our` only creates an alias to an existing variable: a package variable of the same name.

This means that when `use strict 'vars'` is in effect, `our` lets you use a package variable without qualifying it with the package name, but only within the lexical scope of the `our` declaration. This applies immediately–even within the same statement.

```
package Foo;
use strict;
```

```
    $Foo::foo = 23;

    {
        our $foo;   # alias to $Foo::foo
        print $foo; # prints 23
    }

    print $Foo::foo; # prints 23

    print $foo; # ERROR: requires explicit package name
```

This works even if the package variable has not been used before, as package variables spring into existence when first used.

```
    package Foo;
    use strict;

    our $foo = 23;   # just like $Foo::foo = 23

    print $Foo::foo; # prints 23
```

Because the variable becomes legal immediately under `use strict 'vars'`, so long as there is no variable with that name is already in scope, you can then reference the package variable again even within the same statement.

```
    package Foo;
    use strict;

    my  $foo = $foo; # error, undeclared $foo on right-hand side
    our $foo = $foo; # no errors
```

If more than one variable is listed, the list must be placed in parentheses.

```
    our($bar, $baz);
```

An `our` declaration declares an alias for a package variable that will be visible across its entire lexical scope, even across package boundaries. The package in which the variable is entered is determined at the point of the declaration, not at the point of use. This means the following behavior holds:

```
    package Foo;
    our $bar;      # declares $Foo::bar for rest of lexical scope
    $bar = 20;

    package Bar;
    print $bar;    # prints 20, as it refers to $Foo::bar
```

Multiple `our` declarations with the same name in the same lexical scope are allowed if they are in different packages. If they happen to be in the same package, Perl will emit warnings if you have asked for them, just like multiple `my` declarations. Unlike a second `my` declaration, which will bind the name to a fresh variable, a second `our` declaration in the same package, in the same scope, is merely redundant.

```
    use warnings;
```

```
package Foo;
our $bar;      # declares $Foo::bar for rest of lexical scope
$bar = 20;

package Bar;
our $bar = 30; # declares $Bar::bar for rest of lexical scope
print $bar;    # prints 30

our $bar;      # emits warning but has no other effect
print $bar;    # still prints 30
```

An `our` declaration may also have a list of attributes associated with it.

The exact semantics and interface of TYPE and ATTRS are still evolving. TYPE is currently bound to the use of the `fields` pragma, and attributes are handled using the `attributes` pragma, or, starting from Perl 5.8.0, also via the `Attribute::Handlers` module. See Section 73.3.2 [perlsub Private Variables via my()], page 1223 for details, and `fields`, `attributes`, and `Attribute-Handlers`.

Note that with a parenthesised list, `undef` can be used as a dummy placeholder, for example to skip assignment of initial values:

```
our ( undef, $min, $hour ) = localtime;
```

`our` differs from `use vars`, which allows use of an unqualified name *only* within the affected package, but across scopes.

pack TEMPLATE,LIST

Takes a LIST of values and converts it into a string using the rules given by the TEMPLATE. The resulting string is the concatenation of the converted values. Typically, each converted value looks like its machine-level representation. For example, on 32-bit machines an integer may be represented by a sequence of 4 bytes, which will in Perl be presented as a string that's 4 characters long.

See Section 50.1 [perlpacktut NAME], page 857 for an introduction to this function.

The TEMPLATE is a sequence of characters that give the order and type of values, as follows:

```
a  A string with arbitrary binary data, will be null padded.
A  A text (ASCII) string, will be space padded.
Z  A null-terminated (ASCIZ) string, will be null padded.

b  A bit string (ascending bit order inside each byte,
   like vec()).
B  A bit string (descending bit order inside each byte).
h  A hex string (low nybble first).
H  A hex string (high nybble first).

c  A signed char (8-bit) value.
C  An unsigned char (octet) value.
W  An unsigned char value (can be greater than 255).
```

```
s  A signed short (16-bit) value.
S  An unsigned short value.

l  A signed long (32-bit) value.
L  An unsigned long value.

q  A signed quad (64-bit) value.
Q  An unsigned quad value.
     (Quads are available only if your system supports 64-bit
      integer values _and_ if Perl has been compiled to support
      those.  Raises an exception otherwise.)

i  A signed integer value.
I  A unsigned integer value.
     (This 'integer' is _at_least_ 32 bits wide.  Its exact
      size depends on what a local C compiler calls 'int'.)

n  An unsigned short (16-bit) in "network" (big-endian) order.
N  An unsigned long (32-bit) in "network" (big-endian) order.
v  An unsigned short (16-bit) in "VAX" (little-endian) order.
V  An unsigned long (32-bit) in "VAX" (little-endian) order.

j  A Perl internal signed integer value (IV).
J  A Perl internal unsigned integer value (UV).

f  A single-precision float in native format.
d  A double-precision float in native format.

F  A Perl internal floating-point value (NV) in native format
D  A float of long-double precision in native format.
     (Long doubles are available only if your system supports
      long double values _and_ if Perl has been compiled to
      support those.  Raises an exception otherwise.
      Note that there are different long double formats.)

p  A pointer to a null-terminated string.
P  A pointer to a structure (fixed-length string).

u  A uuencoded string.
U  A Unicode character number.  Encodes to a character in char-
   acter mode and UTF-8 (or UTF-EBCDIC in EBCDIC platforms) in
   byte mode.

w  A BER compressed integer (not an ASN.1 BER, see perlpacktut
   for details).  Its bytes represent an unsigned integer in
   base 128, most significant digit first, with as few digits
```

```
       as possible.  Bit eight (the high bit) is set on each byte
       except the last.

    x  A null byte (a.k.a ASCII NUL, "\000", chr(0))
    X  Back up a byte.
    @  Null-fill or truncate to absolute position, counted from the
       start of the innermost ()-group.
    .  Null-fill or truncate to absolute position specified by
       the value.
    (  Start of a ()-group.
```

One or more modifiers below may optionally follow certain letters in the TEMPLATE (the second column lists letters for which the modifier is valid):

```
    !   sSlLiI     Forces native (short, long, int) sizes instead
                   of fixed (16-/32-bit) sizes.

    !   xX         Make x and X act as alignment commands.

    !   nNvV       Treat integers as signed instead of unsigned.

    !   @.         Specify position as byte offset in the internal
                   representation of the packed string.  Efficient
                   but dangerous.

    >   sSiIlLqQ   Force big-endian byte-order on the type.
        jJfFdDpP   (The "big end" touches the construct.)

    <   sSiIlLqQ   Force little-endian byte-order on the type.
        jJfFdDpP   (The "little end" touches the construct.)
```

The `>` and `<` modifiers can also be used on `()` groups to force a particular byte-order on all components in that group, including all its subgroups.

The following rules apply:

- Each letter may optionally be followed by a number indicating the repeat count. A numeric repeat count may optionally be enclosed in brackets, as in `pack("C[80]", @arr)`. The repeat count gobbles that many values from the LIST when used with all format types other than `a`, `A`, `Z`, `b`, `B`, `h`, `H`, `@`, `.`, `x`, `X`, and `P`, where it means something else, described below. Supplying a `*` for the repeat count instead of a number means to use however many items are left, except for:

  - `@`, `x`, and `X`, where it is equivalent to `0`.
  - `<.>`, where it means relative to the start of the string.
  - `u`, where it is equivalent to 1 (or 45, which here is equivalent).

  One can replace a numeric repeat count with a template letter enclosed in brackets to use the packed byte length of the bracketed template for the repeat count.

For example, the template `x[L]` skips as many bytes as in a packed long, and the template `"$t X[$t] $t"` unpacks twice whatever $t (when variable-expanded) unpacks. If the template in brackets contains alignment commands (such as `x![d]`), its packed length is calculated as if the start of the template had the maximal possible alignment.

When used with `Z`, a `*` as the repeat count is guaranteed to add a trailing null byte, so the resulting string is always one byte longer than the byte length of the item itself.

When used with `@`, the repeat count represents an offset from the start of the innermost `()` group.

When used with `.`, the repeat count determines the starting position to calculate the value offset as follows:

- If the repeat count is `0`, it's relative to the current position.
- If the repeat count is `*`, the offset is relative to the start of the packed string.
- And if it's an integer *n*, the offset is relative to the start of the *n*th innermost `( )` group, or to the start of the string if *n* is bigger then the group level.

The repeat count for `u` is interpreted as the maximal number of bytes to encode per line of output, with 0, 1 and 2 replaced by 45. The repeat count should not be more than 65.

- The `a`, `A`, and `Z` types gobble just one value, but pack it as a string of length count, padding with nulls or spaces as needed. When unpacking, `A` strips trailing whitespace and nulls, `Z` strips everything after the first null, and `a` returns data with no stripping at all.

  If the value to pack is too long, the result is truncated. If it's too long and an explicit count is provided, `Z` packs only `$count-1` bytes, followed by a null byte. Thus `Z` always packs a trailing null, except when the count is 0.

- Likewise, the `b` and `B` formats pack a string that's that many bits long. Each such format generates 1 bit of the result. These are typically followed by a repeat count like `B8` or `B64`.

  Each result bit is based on the least-significant bit of the corresponding input character, i.e., on `ord($char)%2`. In particular, characters `"0"` and `"1"` generate bits 0 and 1, as do characters `"\000"` and `"\001"`.

  Starting from the beginning of the input string, each 8-tuple of characters is converted to 1 character of output. With format `b`, the first character of the 8-tuple determines the least-significant bit of a character; with format `B`, it determines the most-significant bit of a character.

  If the length of the input string is not evenly divisible by 8, the remainder is packed as if the input string were padded by null characters at the end. Similarly during unpacking, `"extra"` bits are ignored.

  If the input string is longer than needed, remaining characters are ignored.

  A `*` for the repeat count uses all characters of the input field. On unpacking, bits are converted to a string of `0`s and `1`s.

- The `h` and `H` formats pack a string that many nybbles (4-bit groups, representable as hexadecimal digits, `"0".."9"` `"a".."f"`) long.

  For each such format, pack() generates 4 bits of result. With non-alphabetical characters, the result is based on the 4 least-significant bits of the input character, i.e., on `ord($char)%16`. In particular, characters `"0"` and `"1"` generate nybbles 0 and 1, as do bytes `"\000"` and `"\001"`. For characters `"a".."f"` and `"A".."F"`, the result is compatible with the usual hexadecimal digits, so that `"a"` and `"A"` both generate the nybble `0xA==10`. Use only these specific hex characters with this format.

  Starting from the beginning of the template to pack(), each pair of characters is converted to 1 character of output. With format `h`, the first character of the pair determines the least-significant nybble of the output character; with format `H`, it determines the most-significant nybble.

  If the length of the input string is not even, it behaves as if padded by a null character at the end. Similarly, "extra" nybbles are ignored during unpacking.

  If the input string is longer than needed, extra characters are ignored.

  A `*` for the repeat count uses all characters of the input field. For unpack(), nybbles are converted to a string of hexadecimal digits.

- The `p` format packs a pointer to a null-terminated string. You are responsible for ensuring that the string is not a temporary value, as that could potentially get deallocated before you got around to using the packed result. The `P` format packs a pointer to a structure of the size indicated by the length. A null pointer is created if the corresponding value for `p` or `P` is `undef`; similarly with unpack(), where a null pointer unpacks into `undef`.

  If your system has a strange pointer size–meaning a pointer is neither as big as an int nor as big as a long–it may not be possible to pack or unpack pointers in big- or little-endian byte order. Attempting to do so raises an exception.

- The `/` template character allows packing and unpacking of a sequence of items where the packed structure contains a packed item count followed by the packed items themselves. This is useful when the structure you're unpacking has encoded the sizes or repeat counts for some of its fields within the structure itself as separate fields.

  For `pack`, you write *length-item*`/`*sequence-item*, and the *length-item* describes how the length value is packed. Formats likely to be of most use are integer-packing ones like `n` for Java strings, `w` for ASN.1 or SNMP, and `N` for Sun XDR.

  For `pack`, *sequence-item* may have a repeat count, in which case the minimum of that and the number of available items is used as the argument for *length-item*. If it has no repeat count or uses a '*', the number of available items is used.

  For `unpack`, an internal stack of integer arguments unpacked so far is used. You write `/`*sequence-item* and the repeat count is obtained by popping off the last element from the stack. The *sequence-item* must not have a repeat count.

If *sequence-item* refers to a string type ("A", "a", or "Z"), the *length-item* is the string length, not the number of strings. With an explicit repeat count for pack, the packed string is adjusted to that length. For example:

```
 This code:                           gives this result:

 unpack("W/a", "\004Gurusamy")        ("Guru")
 unpack("a3/A A*", "007 Bond  J ")    (" Bond", "J")
 unpack("a3 x2 /A A*", "007: Bond, J.") ("Bond, J", ".")

 pack("n/a* w/a","hello,","world")   "\000\006hello,\005world"
 pack("a/W2", ord("a") .. ord("z"))  "2ab"
```

The *length-item* is not returned explicitly from `unpack`.

Supplying a count to the *length-item* format letter is only useful with `A`, `a`, or `Z`. Packing with a *length-item* of `a` or `Z` may introduce `"\000"` characters, which Perl does not regard as legal in numeric strings.

- The integer types `s`, `S`, `l`, and `L` may be followed by a `!` modifier to specify native shorts or longs. As shown in the example above, a bare `l` means exactly 32 bits, although the native `long` as seen by the local C compiler may be larger. This is mainly an issue on 64-bit platforms. You can see whether using `!` makes any difference this way:

```
    printf "format s is %d, s! is %d\n",
        length pack("s"), length pack("s!");

    printf "format l is %d, l! is %d\n",
        length pack("l"), length pack("l!");
```

`i!` and `I!` are also allowed, but only for completeness' sake: they are identical to `i` and `I`.

The actual sizes (in bytes) of native shorts, ints, longs, and long longs on the platform where Perl was built are also available from the command line:

```
    $ perl -V:{short,int,long{,long}}size
    shortsize='2';
    intsize='4';
    longsize='4';
    longlongsize='8';
```

or programmatically via the `Config` module:

```
       use Config;
       print $Config{shortsize},    "\n";
       print $Config{intsize},      "\n";
       print $Config{longsize},     "\n";
       print $Config{longlongsize}, "\n";
```

`$Config{longlongsize}` is undefined on systems without long long support.

- The integer formats `s`, `S`, `i`, `I`, `l`, `L`, `j`, and `J` are inherently non-portable between processors and operating systems because they obey native byteorder and endianness. For example, a 4-byte integer 0x12345678 (305419896 decimal) would be ordered natively (arranged in and handled by the CPU registers) into bytes as

  ```
  0x12 0x34 0x56 0x78  # big-endian
  0x78 0x56 0x34 0x12  # little-endian
  ```

  Basically, Intel and VAX CPUs are little-endian, while everybody else, including Motorola m68k/88k, PPC, Sparc, HP PA, Power, and Cray, are big-endian. Alpha and MIPS can be either: Digital/Compaq uses (well, used) them in little-endian mode, but SGI/Cray uses them in big-endian mode.

  The names *big-endian* and *little-endian* are comic references to the egg-eating habits of the little-endian Lilliputians and the big-endian Blefuscudians from the classic Jonathan Swift satire, *Gulliver's Travels*. This entered computer lingo via the paper "On Holy Wars and a Plea for Peace" by Danny Cohen, USC/ISI IEN 137, April 1, 1980.

  Some systems may have even weirder byte orders such as

  ```
  0x56 0x78 0x12 0x34
  0x34 0x12 0x78 0x56
  ```

  These are called mid-endian, middle-endian, mixed-endian, or just weird.

  You can determine your system endianness with this incantation:

  ```
  printf("%#02x ", $_) for unpack("W*", pack L=>0x12345678);
  ```

  The byteorder on the platform where Perl was built is also available via `Config`:

  ```
  use Config;
  print "$Config{byteorder}\n";
  ```

  or from the command line:

  ```
  $ perl -V:byteorder
  ```

  Byteorders `"1234"` and `"12345678"` are little-endian; `"4321"` and `"87654321"` are big-endian. Systems with multiarchitecture binaries will have `"ffff"`, signifying that static information doesn't work, one must use runtime probing.

  For portably packed integers, either use the formats `n`, `N`, `v`, and `V` or else use the `>` and `<` modifiers described immediately below. See also Section 56.1 [perlport NAME], page 951.

- Also floating point numbers have endianness. Usually (but not always) this agrees with the integer endianness. Even though most platforms these days use the IEEE 754 binary format, there are differences, especially if the long doubles are involved. You can see the `Config` variables `doublekind` and `longdblkind` (also `doublesize`, `longdblsize`): the "kind" values are enums, unlike `byteorder`.

Portability-wise the best option is probably to keep to the IEEE 754 64-bit doubles, and of agreed-upon endianness. Another possibility is the `"%a"`) format of `printf`.

- Starting with Perl 5.10.0, integer and floating-point formats, along with the `p` and `P` formats and `()` groups, may all be followed by the `>` or `<` endianness modifiers to respectively enforce big- or little-endian byte-order. These modifiers are especially useful given how `n`, `N`, `v`, and `V` don't cover signed integers, 64-bit integers, or floating-point values.

  Here are some concerns to keep in mind when using an endianness modifier:

  - Exchanging signed integers between different platforms works only when all platforms store them in the same format. Most platforms store signed integers in two's-complement notation, so usually this is not an issue.
  - The `>` or `<` modifiers can only be used on floating-point formats on big- or little-endian machines. Otherwise, attempting to use them raises an exception.
  - Forcing big- or little-endian byte-order on floating-point values for data exchange can work only if all platforms use the same binary representation such as IEEE floating-point. Even if all platforms are using IEEE, there may still be subtle differences. Being able to use `>` or `<` on floating-point values can be useful, but also dangerous if you don't know exactly what you're doing. It is not a general way to portably store floating-point values.
  - When using `>` or `<` on a `()` group, this affects all types inside the group that accept byte-order modifiers, including all subgroups. It is silently ignored for all other types. You are not allowed to override the byte-order within a group that already has a byte-order modifier suffix.

- Real numbers (floats and doubles) are in native machine format only. Due to the multiplicity of floating-point formats and the lack of a standard `"network"` representation for them, no facility for interchange has been made. This means that packed floating-point data written on one machine may not be readable on another, even if both use IEEE floating-point arithmetic (because the endianness of the memory representation is not part of the IEEE spec). See also Section 56.1 [perlport NAME], page 951.

  If you know *exactly* what you're doing, you can use the `>` or `<` modifiers to force big- or little-endian byte-order on floating-point values.

  Because Perl uses doubles (or long doubles, if configured) internally for all numeric calculation, converting from double into float and thence to double again loses precision, so `unpack("f", pack("f", $foo)`) will not in general equal $foo.

- Pack and unpack can operate in two modes: character mode (`C0` mode) where the packed string is processed per character, and UTF-8 byte mode (`U0` mode) where the packed string is processed in its UTF-8-encoded Unicode form on a byte-by-byte basis. Character mode is the default unless the format string starts with U. You can always switch mode mid-format with an explicit C0 or U0 in the format. This mode remains in effect until the next mode change, or until the end of the () group it (directly) applies to.

Using C0 to get Unicode characters while using U0 to get *non*-Unicode bytes is not necessarily obvious. Probably only the first of these is what you want:

```
$ perl -CS -E ’say "\x{3B1}\x{3C9}"’ |
  perl -CS -ne ’printf "%v04X\n", $_ for unpack("C0A*", $_)’
03B1.03C9
$ perl -CS -E ’say "\x{3B1}\x{3C9}"’ |
  perl -CS -ne ’printf "%v02X\n", $_ for unpack("U0A*", $_)’
CE.B1.CF.89
$ perl -CS -E ’say "\x{3B1}\x{3C9}"’ |
  perl -C0 -ne ’printf "%v02X\n", $_ for unpack("C0A*", $_)’
CE.B1.CF.89
$ perl -CS -E ’say "\x{3B1}\x{3C9}"’ |
  perl -C0 -ne ’printf "%v02X\n", $_ for unpack("U0A*", $_)’
C3.8E.C2.B1.C3.8F.C2.89
```

Those examples also illustrate that you should not try to use pack/unpack as a substitute for the Encode module.

- You must yourself do any alignment or padding by inserting, for example, enough "x"es while packing. There is no way for pack() and unpack() to know where characters are going to or coming from, so they handle their output and input as flat sequences of characters.
- A () group is a sub-TEMPLATE enclosed in parentheses. A group may take a repeat count either as postfix, or for unpack(), also via the / template character. Within each repetition of a group, positioning with @ starts over at 0. Therefore, the result of

  ```
  pack("@1A((@2A)@3A)", qw[X Y Z])
  ```

  is the string "\0X\0\0YZ".
- x and X accept the ! modifier to act as alignment commands: they jump forward or back to the closest position aligned at a multiple of count characters. For example, to pack() or unpack() a C structure like

  ```
  struct {
      char   c;    /* one signed, 8-bit character */
      double d;
      char   cc[2];
  }
  ```

  one may need to use the template c x![d] d c[2]. This assumes that doubles must be aligned to the size of double.

  For alignment commands, a count of 0 is equivalent to a count of 1; both are no-ops.
- n, N, v and V accept the ! modifier to represent signed 16-/32-bit integers in big-/little-endian order. This is portable only when all platforms sharing

packed data use the same binary representation for signed integers; for example, when all platforms use two's-complement representation.

- Comments can be embedded in a TEMPLATE using `#` through the end of line. White space can separate pack codes from each other, but modifiers and repeat counts must follow immediately. Breaking complex templates into individual line-by-line components, suitably annotated, can do as much to improve legibility and maintainability of pack/unpack formats as `/x` can for complicated pattern matches.
- If TEMPLATE requires more arguments than pack() is given, pack() assumes additional `""` arguments. If TEMPLATE requires fewer arguments than given, extra arguments are ignored.
- Attempting to pack the special floating point values `Inf` and `NaN` (infinity, also in negative, and not-a-number) into packed integer values (like `"L"`) is a fatal error. The reason for this is that there simply isn't any sensible mapping for these special values into integers.

Examples:

```
$foo = pack("WWWW",65,66,67,68);
# foo eq "ABCD"
$foo = pack("W4",65,66,67,68);
# same thing
$foo = pack("W4",0x24b6,0x24b7,0x24b8,0x24b9);
# same thing with Unicode circled letters.
$foo = pack("U4",0x24b6,0x24b7,0x24b8,0x24b9);
# same thing with Unicode circled letters.  You don't get the
# UTF-8 bytes because the U at the start of the format caused
# a switch to U0-mode, so the UTF-8 bytes get joined into
# characters
$foo = pack("C0U4",0x24b6,0x24b7,0x24b8,0x24b9);
# foo eq "\xe2\x92\xb6\xe2\x92\xb7\xe2\x92\xb8\xe2\x92\xb9"
# This is the UTF-8 encoding of the string in the
# previous example

$foo = pack("ccxxcc",65,66,67,68);
# foo eq "AB\0\0CD"

# NOTE: The examples above featuring "W" and "c" are true
# only on ASCII and ASCII-derived systems such as ISO Latin 1
# and UTF-8.  On EBCDIC systems, the first example would be
#      $foo = pack("WWWW",193,194,195,196);

$foo = pack("s2",1,2);
# "\001\000\002\000" on little-endian
# "\000\001\000\002" on big-endian

$foo = pack("a4","abcd","x","y","z");
# "abcd"
```

```
$foo = pack("aaaa","abcd","x","y","z");
# "axyz"

$foo = pack("a14","abcdefg");
# "abcdefg\0\0\0\0\0\0\0"

$foo = pack("i9pl", gmtime);
# a real struct tm (on my system anyway)

$utmp_template = "Z8 Z8 Z16 L";
$utmp = pack($utmp_template, @utmp1);
# a struct utmp (BSDish)

@utmp2 = unpack($utmp_template, $utmp);
# "@utmp1" eq "@utmp2"

sub bintodec {
    unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
}

$foo = pack('sx2l', 12, 34);
# short 12, two zero bytes padding, long 34
$bar = pack('s@4l', 12, 34);
# short 12, zero fill to position 4, long 34
# $foo eq $bar
$baz = pack('s.l', 12, 4, 34);
# short 12, zero fill to position 4, long 34

$foo = pack('nN', 42, 4711);
# pack big-endian 16- and 32-bit unsigned integers
$foo = pack('S>L>', 42, 4711);
# exactly the same
$foo = pack('s<l<', -42, 4711);
# pack little-endian 16- and 32-bit signed integers
$foo = pack('(sl)<', -42, 4711);
# exactly the same
```

The same template may generally also be used in unpack().

package NAMESPACE
package NAMESPACE VERSION
package NAMESPACE BLOCK
package NAMESPACE VERSION BLOCK

Declares the BLOCK or the rest of the compilation unit as being in the given namespace. The scope of the package declaration is either the supplied code BLOCK or, in the absence of a BLOCK, from the declaration itself through the end of current scope (the enclosing block, file, or `eval`). That is, the forms without a BLOCK are operative through the end of the current scope, just like the `my`, `state`, and `our` operators. All unqualified dynamic identifiers in this scope will be in the given namespace, except where overridden by another `package` declaration or when they're one of the special identifiers that qualify into `main::`, like `STDOUT`, `ARGV`, `ENV`, and the punctuation variables.

A package statement affects dynamic variables only, including those you've used `local` on, but *not* lexically-scoped variables, which are created with `my`, `state`, or `our`. Typically it would be the first declaration in a file included by `require` or `use`. You can switch into a package in more than one place, since this only determines which default symbol table the compiler uses for the rest of that block. You can refer to identifiers in other packages than the current one by prefixing the identifier with the package name and a double colon, as in `$SomePack::var` or `ThatPack::INPUT_HANDLE`. If package name is omitted, the `main` package as assumed. That is, `$::sail` is equivalent to `$main::sail` (as well as to `$main'sail`, still seen in ancient code, mostly from Perl 4).

If VERSION is provided, `package` sets the `$VERSION` variable in the given namespace to a `version` object with the VERSION provided. VERSION must be a "strict" style version number as defined by the `version` module: a positive decimal number (integer or decimal-fraction) without exponentiation or else a dotted-decimal v-string with a leading 'v' character and at least three components. You should set `$VERSION` only once per package.

See Section 40.2.2 [perlmod Packages], page 732 for more information about packages, modules, and classes. See Section 73.1 [perlsub NAME], page 1216 for other scoping issues.

__PACKAGE__

A special token that returns the name of the package in which it occurs.

pipe READHANDLE,WRITEHANDLE

Opens a pair of connected pipes like the corresponding system call. Note that if you set up a loop of piped processes, deadlock can occur unless you are very careful. In addition, note that Perl's pipes use IO buffering, so you may need to set `$|` to flush your WRITEHANDLE after each command, depending on the application.

Returns true on success.

See `IPC-Open2`, `IPC-Open3`, and Section 36.5.6 [perlipc Bidirectional Communication with Another Process], page 680 for examples of such things.

On systems that support a close-on-exec flag on files, that flag is set on all newly opened file descriptors whose `fileno`s are *higher* than the current value of $^F (by default 2 for `STDERR`). See [perlvar $^F], page 1381.

pop ARRAY
pop EXPR
pop

Pops and returns the last value of the array, shortening the array by one element.

Returns the undefined value if the array is empty, although this may also happen at other times. If ARRAY is omitted, pops the `@ARGV` array in the main program, but the `@_` array in subroutines, just like `shift`.

Starting with Perl 5.14, `pop` can take a scalar EXPR, which must hold a reference to an unblessed array. The argument will be dereferenced automatically. This aspect of `pop` is considered highly experimental. The exact behaviour may change in a future version of Perl.

To avoid confusing would-be users of your code who are running earlier versions of Perl with mysterious syntax errors, put this sort of thing at the top of your file to signal that your code will work *only* on Perls of a recent vintage:

```
use 5.014;  # so push/pop/etc work on scalars (experimental)
```

pos SCALAR
pos

Returns the offset of where the last `m//g` search left off for the variable in question (`$_` is used when the variable is not specified). Note that 0 is a valid match offset. `undef` indicates that the search position is reset (usually due to match failure, but can also be because no match has yet been run on the scalar).

`pos` directly accesses the location used by the regexp engine to store the offset, so assigning to `pos` will change that offset, and so will also influence the `\G` zero-width assertion in regular expressions. Both of these effects take place for the next match, so you can't affect the position with `pos` during the current match, such as in `(?{pos() = 5})` or `s//pos() = 5/e`.

Setting `pos` also resets the *matched with zero-length* flag, described under Section 58.2.9 [perlre Repeated Patterns Matching a Zero-length Substring], page 1026.

Because a failed `m//gc` match doesn't reset the offset, the return from `pos` won't change either in this case. See Section 58.1 [perlre NAME], page 989 and Section 48.1 [perlop NAME], page 798.

print FILEHANDLE LIST
print FILEHANDLE
print LIST
print

Prints a string or a list of strings. Returns true if successful. FILEHANDLE may be a scalar variable containing the name of or a reference to the filehandle, thus introducing one level of indirection. (NOTE: If FILEHANDLE is a variable and the next token is a term, it may be misinterpreted as an operator unless you interpose a `+` or put parentheses around the arguments.) If FILEHANDLE is omitted, prints to the last selected (see [select], page 443) output handle. If LIST is omitted, prints `$_` to the currently selected output handle. To use FILEHANDLE alone to print the content of `$_` to it, you must use a real filehandle like `FH`, not an indirect one like `$fh`. To set the default output handle to something other than STDOUT, use the select operation.

The current value of `$,` (if any) is printed between each LIST item. The current value of `$\` (if any) is printed after the entire LIST has been printed. Because print takes a LIST, anything in the LIST is evaluated in list context, including any subroutines whose return lists you pass to `print`. Be careful not to follow the print keyword with a left parenthesis unless you want the corresponding right parenthesis to terminate the arguments to the print; put parentheses around all arguments (or interpose a `+`, but that doesn't look as good).

If you're storing handles in an array or hash, or in general whenever you're using any expression more complex than a bareword handle or a plain, unsubscripted scalar variable to retrieve it, you will have to use a block returning the filehandle value instead, in which case the LIST may not be omitted:

```
print { $files[$i] } "stuff\n";
print { $OK ? STDOUT : STDERR } "stuff\n";
```

Printing to a closed pipe or socket will generate a SIGPIPE signal. See Section 36.1 [perlipc NAME], page 667 for more on signal handling.

printf FILEHANDLE FORMAT, LIST
printf FILEHANDLE
printf FORMAT, LIST
printf

Equivalent to `print FILEHANDLE sprintf(FORMAT, LIST)`, except that `$\` (the output record separator) is not appended. The FORMAT and the LIST are actually parsed as a single list. The first argument of the list will be interpreted as the `printf` format. This means that `printf(@_)` will use `$_[0]` as the format. See [sprintf], page 456 for an explanation of the format argument. If `use locale` for `LC_NUMERIC` Look for this throught pod is in effect and POSIX::setlocale() has been called, the character used for the decimal separator in formatted floating-point numbers is affected by the `LC_NUMERIC` locale setting. See Section 38.1 [perllocale NAME], page 701 and `POSIX`.

For historical reasons, if you omit the list, `$_` is used as the format; to use FILEHANDLE without a list, you must use a real filehandle like `FH`, not an indirect one like `$fh`. However, this will rarely do what you want; if $_ contains formatting codes, they will be replaced with the empty string and a warning will be emitted if warnings are enabled. Just use `print` if you want to print the contents of $_.

Don't fall into the trap of using a `printf` when a simple `print` would do. The `print` is more efficient and less error prone.

prototype FUNCTION
prototype

Returns the prototype of a function as a string (or `undef` if the function has no prototype). FUNCTION is a reference to, or the name of, the function whose prototype you want to retrieve. If FUNCTION is omitted, $_ is used.

If FUNCTION is a string starting with `CORE::`, the rest is taken as a name for a Perl builtin. If the builtin's arguments cannot be adequately expressed by a prototype (such as `system`), prototype() returns `undef`, because the builtin does not really behave like a Perl function. Otherwise, the string describing the equivalent prototype is returned.

push ARRAY,LIST
push EXPR,LIST

Treats ARRAY as a stack by appending the values of LIST to the end of ARRAY. The length of ARRAY increases by the length of LIST. Has the same effect as

```
for $value (LIST) {
    $ARRAY[++$#ARRAY] = $value;
}
```

but is more efficient. Returns the number of elements in the array following the completed `push`.

Starting with Perl 5.14, `push` can take a scalar EXPR, which must hold a reference to an unblessed array. The argument will be dereferenced automatically. This aspect of `push` is considered highly experimental. The exact behaviour may change in a future version of Perl.

To avoid confusing would-be users of your code who are running earlier versions of Perl with mysterious syntax errors, put this sort of thing at the top of your file to signal that your code will work *only* on Perls of a recent vintage:

```
use 5.014;  # so push/pop/etc work on scalars (experimental)
```

q/STRING/
qq/STRING/
qw/STRING/
qx/STRING/

Generalized quotes. See Section 48.2.31 [perlop Quote-Like Operators], page 832.

qr/STRING/

Regexp-like quote. See Section 48.2.30 [perlop Regexp Quote-Like Operators], page 823.

quotemeta EXPR
quotemeta

Returns the value of EXPR with all the ASCII non-"word" characters backslashed. (That is, all ASCII characters not matching `/[A-Za-z_0-9]/` will be preceded by a backslash in the returned string, regardless of any locale settings.) This is the internal function implementing the `\Q` escape in double-quoted strings. (See below for the behavior on non-ASCII code points.)

If EXPR is omitted, uses `$_`.

quotemeta (and `\Q` ... `\E`) are useful when interpolating strings into regular expressions, because by default an interpolated variable will be considered a mini-regular expression. For example:

```
my $sentence = 'The quick brown fox jumped over the lazy dog';
my $substring = 'quick.*?fox';
$sentence =~ s{$substring}{big bad wolf};
```

Will cause `$sentence` to become `'The big bad wolf jumped over...'`.

On the other hand:

```
    my $sentence = 'The quick brown fox jumped over the lazy dog';
    my $substring = 'quick.*?fox';
    $sentence =~ s{\Q$substring\E}{big bad wolf};
```

Or:

```
    my $sentence = 'The quick brown fox jumped over the lazy dog';
    my $substring = 'quick.*?fox';
    my $quoted_substring = quotemeta($substring);
    $sentence =~ s{$quoted_substring}{big bad wolf};
```

Will both leave the sentence as is. Normally, when accepting literal string input from the user, quotemeta() or `\Q` must be used.

In Perl v5.14, all non-ASCII characters are quoted in non-UTF-8-encoded strings, but not quoted in UTF-8 strings.

Starting in Perl v5.16, Perl adopted a Unicode-defined strategy for quoting non-ASCII characters; the quoting of ASCII characters is unchanged.

Also unchanged is the quoting of non-UTF-8 strings when outside the scope of a `use feature 'unicode_strings'`, which is to quote all characters in the upper Latin1 range. This provides complete backwards compatibility for old programs which do not use Unicode. (Note that `unicode_strings` is automatically enabled within the scope of a `use v5.12` or greater.)

Within the scope of `use locale`, all non-ASCII Latin1 code points are quoted whether the string is encoded as UTF-8 or not. As mentioned above, locale does not affect the quoting of ASCII-range characters. This protects against those locales where characters such as `"|"` are considered to be word characters.

Otherwise, Perl quotes non-ASCII characters using an adaptation from Unicode (see `http://www.unicode.org/reports/tr31/`). The only code points that are quoted are those that have any of the Unicode properties: Pattern_Syntax, Pattern_White_Space, White_Space, Default_Ignorable_Code_Point, or General_Category=Control.

Of these properties, the two important ones are Pattern_Syntax and Pattern_White_Space. They have been set up by Unicode for exactly this purpose of deciding which characters in a regular expression pattern should be quoted. No character that can be in an identifier has these properties.

Perl promises, that if we ever add regular expression pattern metacharacters to the dozen already defined (`\ | ( ) [ { ^ $ * + ? .`), that we will only use ones that have the Pattern_Syntax property. Perl also promises, that if we ever add characters that are considered to be white space in regular expressions (currently mostly affected by `/x`), they will all have the Pattern_White_Space property.

Unicode promises that the set of code points that have these two properties will never change, so something that is not quoted in v5.16 will never need to be quoted in any future Perl release. (Not all the code points that match Pattern_Syntax have actually had characters assigned to them; so there is room to grow, but they are quoted whether assigned or not. Perl, of course, would never use an unassigned code point as an actual metacharacter.)

Quoting characters that have the other 3 properties is done to enhance the readability of the regular expression and not because they actually need to be quoted for regular expression purposes (characters with the White_Space property are likely to be indistinguishable on the page or screen from those with the Pattern_White_Space property; and the other two properties contain non-printing characters).

rand EXPR
rand

Returns a random fractional number greater than or equal to `0` and less than the value of EXPR. (EXPR should be positive.) If EXPR is omitted, the value `1` is used. Currently EXPR with the value `0` is also special-cased as `1` (this was undocumented before Perl 5.8.0 and is subject to change in future versions of Perl). Automatically calls `srand` unless `srand` has already been called. See also `srand`.

Apply `int()` to the value returned by `rand()` if you want random integers instead of random fractional numbers. For example,

```
int(rand(10))
```

returns a random integer between `0` and `9`, inclusive.

(Note: If your rand function consistently returns numbers that are too large or too small, then your version of Perl was probably compiled with the wrong number of RANDBITS.)

`rand()` is not cryptographically secure. You should not rely on it in security-sensitive situations. As of this writing, a number of third-party CPAN modules offer random number generators intended by their authors to be cryptographically secure, including: `Data-Entropy`, `Crypt-Random`, `Math-Random-Secure`, and `Math-TrulyRandom`.

read FILEHANDLE,SCALAR,LENGTH,OFFSET
read FILEHANDLE,SCALAR,LENGTH

Attempts to read LENGTH *characters* of data into variable SCALAR from the specified FILEHANDLE. Returns the number of characters actually read, `0` at end of file, or undef if there was an error (in the latter case `$!` is also set). SCALAR will be grown or shrunk so that the last character actually read is the last character of the scalar after the read.

An OFFSET may be specified to place the read data at some place in the string other than the beginning. A negative OFFSET specifies placement at that many characters counting backwards from the end of the string. A positive OFFSET greater than the length of SCALAR results in the string being padded to the required size with `"\0"` bytes before the result of the read is appended.

The call is implemented in terms of either Perl's or your system's native fread(3) library function. To get a true read(2) system call, see [sysread], page 470.

Note the *characters*: depending on the status of the filehandle, either (8-bit) bytes or characters are read. By default, all filehandles operate on bytes, but for example if the filehandle has been opened with the `:utf8` I/O layer (see ⟨undefined⟩ [open], page ⟨undefined⟩, and the `open` pragma, `open`), the I/O will operate on UTF8-encoded Unicode characters, not bytes. Similarly for the `:encoding` pragma: in that case pretty much any characters can be read.

readdir DIRHANDLE

Returns the next directory entry for a directory opened by `opendir`. If used in list context, returns all the rest of the entries in the directory. If there are no more entries, returns the undefined value in scalar context and the empty list in list context.

If you're planning to filetest the return values out of a `readdir`, you'd better prepend the directory in question. Otherwise, because we didn't `chdir` there, it would have been testing the wrong file.

```
    opendir(my $dh, $some_dir) || die "can't opendir $some_dir: $!";
    @dots = grep { /^\./ && -f "$some_dir/$_" } readdir($dh);
    closedir $dh;
```

As of Perl 5.12 you can use a bare `readdir` in a `while` loop, which will set `$_` on every iteration.

```
    opendir(my $dh, $some_dir) || die;
    while(readdir $dh) {
        print "$some_dir/$_\n";
    }
    closedir $dh;
```

To avoid confusing would-be users of your code who are running earlier versions of Perl with mysterious failures, put this sort of thing at the top of your file to signal that your code will work *only* on Perls of a recent vintage:

```
    use 5.012; # so readdir assigns to $_ in a lone while test
```

readline EXPR
readline

Reads from the filehandle whose typeglob is contained in EXPR (or from `*ARGV` if EXPR is not provided). In scalar context, each call reads and returns the next line until end-of-file is reached, whereupon the subsequent call returns `undef`. In list context, reads until end-of-file is reached and returns a list of lines. Note that the notion of "line" used here is whatever you may have defined with `$/` or `$INPUT_RECORD_SEPARATOR`). See [perlvar $/], page 1395.

When `$/` is set to `undef`, when `readline` is in scalar context (i.e., file slurp mode), and when an empty file is read, it returns `''` the first time, followed by `undef` subsequently.

This is the internal function implementing the `<EXPR>` operator, but you can use it directly. The `<EXPR>` operator is discussed in more detail in Section 48.2.33 [perlop I/O Operators], page 844.

```
    $line = <STDIN>;
    $line = readline(*STDIN);    # same thing
```

If `readline` encounters an operating system error, `$!` will be set with the corresponding error message. It can be helpful to check `$!` when you are reading from filehandles you don't trust, such as a tty or a socket. The following example uses the operator form of `readline` and dies if the result is not defined.

```
    while ( ! eof($fh) ) {
        defined( $_ = <$fh> ) or die "readline failed: $!";
        ...
    }
```

Note that you have can't handle `readline` errors that way with the `ARGV` filehandle. In that case, you have to open each element of `@ARGV` yourself since `eof` handles `ARGV` differently.

```
    foreach my $arg (@ARGV) {
        open(my $fh, $arg) or warn "Can't open $arg: $!";

        while ( ! eof($fh) ) {
            defined( $_ = <$fh> )
                or die "readline failed for $arg: $!";
            ...
        }
    }
```

readlink EXPR
readlink

Returns the value of a symbolic link, if symbolic links are implemented. If not, raises an exception. If there is a system error, returns the undefined value and sets `$!` (errno). If EXPR is omitted, uses `$_`.

Portability issues: [perlport readlink], page 977.

readpipe EXPR
readpipe

EXPR is executed as a system command. The collected standard output of the command is returned. In scalar context, it comes back as a single (potentially multi-line) string. In list context, returns a list of lines (however you've defined lines with `$/` or `$INPUT_RECORD_SEPARATOR`). This is the internal function implementing the `qx/EXPR/` operator, but you can use it directly. The `qx/EXPR/` operator is discussed in more detail in Section 48.2.33 [perlop I/O Operators], page 844. If EXPR is omitted, uses `$_`.

recv SOCKET,SCALAR,LENGTH,FLAGS

Receives a message on a socket. Attempts to receive LENGTH characters of data into variable SCALAR from the specified SOCKET filehandle. SCALAR will be grown or shrunk to the length actually read. Takes the same flags as the system call of the same name. Returns the address of the sender if SOCKET's protocol supports this; returns an empty string otherwise. If there's an error, returns the undefined value. This call is actually implemented in terms of recvfrom(2) system call. See Section 36.9 [perlipc UDP: Message Passing], page 695 for examples.

Note the *characters*: depending on the status of the socket, either (8-bit) bytes or characters are received. By default all sockets operate on bytes, but for example if the socket has been changed using binmode() to operate with the `:encoding(utf8)` I/O layer (see the `open` pragma, `open`), the I/O will operate on UTF8-encoded Unicode characters, not bytes. Similarly for the `:encoding` pragma: in that case pretty much any characters can be read.

redo LABEL
redo EXPR
redo

The `redo` command restarts the loop block without evaluating the conditional again. The `continue` block, if any, is not executed. If the LABEL is omitted, the command refers to the innermost enclosing loop. The `redo EXPR` form, available starting in Perl 5.18.0, allows a label name to be computed at run time, and is otherwise identical to `redo LABEL`. Programs that want to lie to themselves about what was just input normally use this command:

```
# a simpleminded Pascal comment stripper
# (warning: assumes no { or } in strings)
LINE: while (<STDIN>) {
    while (s|({.*}.*){.*}|$1 |) {}
    s|{.*}| |;
    if (s|{.*| |) {
        $front = $_;
        while (<STDIN>) {
            if (/}/) {  # end of comment?
                s|^|$front\{|;
                redo LINE;
            }
        }
    }
    print;
}
```

`redo` cannot be used to retry a block that returns a value such as `eval {}`, `sub {}`, or `do {}`, and should not be used to exit a grep() or map() operation.

Note that a block by itself is semantically identical to a loop that executes once. Thus `redo` inside such a block will effectively turn it into a looping construct.

See also [continue], page 366 for an illustration of how `last`, `next`, and `redo` work.

Unlike most named operators, this has the same precedence as assignment. It is also exempt from the looks-like-a-function rule, so `redo ("foo")."bar"` will cause "bar" to be part of the argument to `redo`.

ref EXPR

ref

Returns a non-empty string if EXPR is a reference, the empty string otherwise. If EXPR is not specified, `$_` will be used. The value returned depends on the type of thing the reference is a reference to.

Builtin types include:

```
SCALAR
ARRAY
```

```
HASH
CODE
REF
GLOB
LVALUE
FORMAT
IO
VSTRING
Regexp
```

You can think of `ref` as a `typeof` operator.

```
if (ref($r) eq "HASH") {
    print "r is a reference to a hash.\n";
}
unless (ref($r)) {
    print "r is not a reference at all.\n";
}
```

The return value `LVALUE` indicates a reference to an lvalue that is not a variable. You get this from taking the reference of function calls like `pos()` or `substr()`. `VSTRING` is returned if the reference points to a Section 11.2.5.2 [version string], page 77.

The result `Regexp` indicates that the argument is a regular expression resulting from `qr//`.

If the referenced object has been blessed into a package, then that package name is returned instead. But don't use that, as it's now considered "bad practice". For one reason, an object could be using a class called `Regexp` or `IO`, or even `HASH`. Also, `ref` doesn't take into account subclasses, like `isa` does.

Instead, use `blessed` (in the `Scalar-Util` module) for boolean checks, `isa` for specific class checks and `reftype` (also from `Scalar-Util`) for type checks. (See Section 46.1 [perlobj NAME], page 769 for details and a `blessed/isa` example.)

See also Section 62.1 [perlref NAME], page 1077.

rename OLDNAME,NEWNAME

Changes the name of a file; an existing file NEWNAME will be clobbered. Returns true for success, false otherwise.

Behavior of this function varies wildly depending on your system implementation. For example, it will usually not work across file system boundaries, even though the system *mv* command sometimes compensates for this. Other restrictions include whether it works on directories, open files, or pre-existing files. Check Section 56.1 [perlport NAME], page 951 and either the rename(2) manpage or equivalent system documentation for details.

For a platform independent `move` function look at the `File-Copy` module.

Portability issues: [perlport rename], page 977.

require VERSION
require EXPR
require

Demands a version of Perl specified by VERSION, or demands some semantics specified by EXPR or by $_ if EXPR is not supplied.

VERSION may be either a numeric argument such as 5.006, which will be compared to $], or a literal of the form v5.6.1, which will be compared to $^V (aka $PERL_VERSION). An exception is raised if VERSION is greater than the version of the current Perl interpreter. Compare with ⟨undefined⟩ [use], page ⟨undefined⟩, which can do a similar check at compile time.

Specifying VERSION as a literal of the form v5.6.1 should generally be avoided, because it leads to misleading error messages under earlier versions of Perl that do not support this syntax. The equivalent numeric version should be used instead.

```
require v5.6.1;     # run time version check
require 5.6.1;      # ditto
require 5.006_001;  # ditto; preferred for backwards
                      compatibility
```

Otherwise, require demands that a library file be included if it hasn't already been included. The file is included via the do-FILE mechanism, which is essentially just a variety of eval with the caveat that lexical variables in the invoking script will be invisible to the included code. If it were implemented in pure Perl, it would have semantics similar to the following:

```
use Carp 'croak';
use version;

sub require {
    my ($filename) = @_;
    if ( my $version = eval { version->parse($filename) } ) {
        if ( $version > $^V ) {
           my $vn = $version->normal;
           croak "Perl $vn required--this is only $^V, stopped";
        }
        return 1;
    }

    if (exists $INC{$filename}) {
        return 1 if $INC{$filename};
        croak "Compilation failed in require";
    }

    foreach $prefix (@INC) {
        if (ref($prefix)) {
            #... do other stuff - see text below ....
        }
        # (see text below about possible appending of .pmc
        # suffix to $filename)
        my $realfilename = "$prefix/$filename";
        next if ! -e $realfilename || -d _ || -b _;
        $INC{$filename} = $realfilename;
        my $result = do($realfilename);
                     # but run in caller’s namespace

        if (!defined $result) {
            $INC{$filename} = undef;
            croak $@ ? "$@Compilation failed in require"
                     : "Can’t locate $filename: $!\n";
        }
        if (!$result) {
            delete $INC{$filename};
            croak "$filename did not return true value";
        }
        $! = 0;
        return $result;
    }
    croak "Can’t locate $filename in \@INC ...";
}
```

Note that the file will not be included twice under the same specified name.

The file must return true as the last statement to indicate successful execution of any initialization code, so it’s customary to end such a file with `1;` unless you’re sure it’ll return true otherwise. But it’s better just to put the `1;`, in case you add more statements.

If EXPR is a bareword, the require assumes a `".pm"` extension and replaces `"::"` with `"/"` in the filename for you, to make it easy to load standard modules. This form of loading of modules does not risk altering your namespace.

In other words, if you try this:

```
        require Foo::Bar;     # a splendid bareword
```

The require function will actually look for the `"Foo/Bar.pm"` file in the directories specified in the `@INC` array.

But if you try this:

```
        $class = ’Foo::Bar’;
        require $class;       # $class is not a bareword
    #or
        require "Foo::Bar";   # not a bareword because of the ""
```

The require function will look for the `"Foo::Bar"` file in the @INC array and will complain about not finding `"Foo::Bar"` there. In this case you can do:

```
        eval "require $class";
```

Now that you understand how `require` looks for files with a bareword argument, there is a little extra functionality going on behind the scenes. Before `require` looks for a `".pm"` extension, it will first look for a similar filename with a `".pmc"` extension. If this file is found, it will be loaded in place of any file ending in a ".pm" extension.

You can also insert hooks into the import facility by putting Perl code directly into the @INC array. There are three forms of hooks: subroutine references, array references, and blessed objects.

Subroutine references are the simplest case. When the inclusion system walks through @INC and encounters a subroutine, this subroutine gets called with two parameters, the first a reference to itself, and the second the name of the file to be included (e.g., "Foo/Bar.pm"). The subroutine should return either nothing or else a list of up to four values in the following order:

1. A reference to a scalar, containing any initial source code to prepend to the file or generator output.
2. A filehandle, from which the file will be read.
3. A reference to a subroutine. If there is no filehandle (previous item), then this subroutine is expected to generate one line of source code per call, writing the line into $_ and returning 1, then finally at end of file returning 0. If there is a filehandle, then the subroutine will be called to act as a simple source filter, with the line as read in $_. Again, return 1 for each valid line, and 0 after all lines have been returned.
4. Optional state for the subroutine. The state is passed in as $_[1]. A reference to the subroutine itself is passed in as $_[0].

If an empty list, undef, or nothing that matches the first 3 values above is returned, then require looks at the remaining elements of @INC. Note that this filehandle must be a real filehandle (strictly a typeglob or reference to a typeglob, whether blessed or unblessed); tied filehandles will be ignored and processing will stop there.

If the hook is an array reference, its first element must be a subroutine reference. This subroutine is called as above, but the first parameter is the array reference. This lets you indirectly pass arguments to the subroutine.

In other words, you can write:

```
push @INC, \&my_sub;
sub my_sub {
    my ($coderef, $filename) = @_;  # $coderef is \&my_sub
    ...
}
```

or:

```
push @INC, [ \&my_sub, $x, $y, ... ];
sub my_sub {
    my ($arrayref, $filename) = @_;
    # Retrieve $x, $y, ...
    my @parameters = @$arrayref[1..$#$arrayref];
    ...
}
```

If the hook is an object, it must provide an INC method that will be called as above, the first parameter being the object itself. (Note that you must fully qualify the sub's name, as unqualified `INC` is always forced into package `main`.) Here is a typical code layout:

```
# In Foo.pm
package Foo;
sub new { ... }
sub Foo::INC {
    my ($self, $filename) = @_;
    ...
}

# In the main program
push @INC, Foo->new(...);
```

These hooks are also permitted to set the %INC entry corresponding to the files they have loaded. See [perlvar %INC], page 1382.

For a yet-more-powerful import facility, see ⟨undefined⟩ [use], page ⟨undefined⟩ and Section 40.1 [perlmod NAME], page 732.

reset EXPR
reset

Generally used in a `continue` block at the end of a loop to clear variables and reset `??` searches so that they work again. The expression is interpreted as a list of single characters (hyphens allowed for ranges). All variables and arrays beginning with one of those letters are reset to their pristine state. If the expression is omitted, one-match searches (`?pattern?`) are reset to match again. Only resets variables or searches in the current package. Always returns 1. Examples:

```
reset 'X';      # reset all X variables
reset 'a-z';    # reset lower case variables
reset;          # just reset ?one-time? searches
```

Resetting `"A-Z"` is not recommended because you'll wipe out your `@ARGV` and `@INC` arrays and your `%ENV` hash. Resets only package variables; lexical variables are unaffected, but they clean themselves up on scope exit anyway, so you'll probably want to use them instead. See ⟨undefined⟩ [my], page ⟨undefined⟩.

return EXPR
return

Returns from a subroutine, `eval`, or `do FILE` with the value given in EXPR. Evaluation of EXPR may be in list, scalar, or void context, depending on how the return value will be used, and the context may vary from one execution to the next (see [wantarray], page 488). If no EXPR is given, returns an empty list in list context, the undefined value in scalar context, and (of course) nothing at all in void context.

(In the absence of an explicit `return`, a subroutine, eval, or do FILE automatically returns the value of the last expression evaluated.)

Unlike most named operators, this is also exempt from the looks-like-a-function rule, so `return ("foo")."bar"` will cause "bar" to be part of the argument to `return`.

reverse LIST

In list context, returns a list value consisting of the elements of LIST in the opposite order. In scalar context, concatenates the elements of LIST and returns a string value with all characters in the opposite order.

```
    print join(", ", reverse "world", "Hello"); # Hello, world

    print scalar reverse "dlrow ,", "olleH";    # Hello, world
```

Used without arguments in scalar context, reverse() reverses `$_`.

```
    $_ = "dlrow ,olleH";
    print reverse;                              # No output, list context
    print scalar reverse;                       # Hello, world
```

Note that reversing an array to itself (as in `@a = reverse @a`) will preserve non-existent elements whenever possible; i.e., for non-magical arrays or for tied arrays with `EXISTS` and `DELETE` methods.

This operator is also handy for inverting a hash, although there are some caveats. If a value is duplicated in the original hash, only one of those can be represented as a key in the inverted hash. Also, this has to unwind one hash and build a whole new one, which may take some time on a large hash, such as from a DBM file.

```
    %by_name = reverse %by_address;  # Invert the hash
```

rewinddir DIRHANDLE

Sets the current position to the beginning of the directory for the `readdir` routine on DIRHANDLE.

Portability issues: [perlport rewinddir], page 977.

rindex STR,SUBSTR,POSITION
rindex STR,SUBSTR

Works just like index() except that it returns the position of the *last* occurrence of SUBSTR in STR. If POSITION is specified, returns the last occurrence beginning at or before that position.

rmdir FILENAME
rmdir

Deletes the directory specified by FILENAME if that directory is empty. If it succeeds it returns true; otherwise it returns false and sets `$!` (errno). If FILENAME is omitted, uses `$_`.

To remove a directory tree recursively (`rm -rf` on Unix) look at the `rmtree` function of the `File-Path` module.

s///

The substitution operator. See Section 48.2.30 [perlop Regexp Quote-Like Operators], page 823.

say FILEHANDLE LIST
say FILEHANDLE
say LIST

say

Just like `print`, but implicitly appends a newline. `say LIST` is simply an abbreviation for `{ local $\ = "\n"; print LIST }`. To use FILEHANDLE without a LIST to print the contents of `$_` to it, you must use a real filehandle like `FH`, not an indirect one like `$fh`.

This keyword is available only when the `"say"` feature is enabled, or when prefixed with `CORE::`; see `feature`. Alternately, include a `use v5.10` or later to the current scope.

scalar EXPR

Forces EXPR to be interpreted in scalar context and returns the value of EXPR.

```
@counts = ( scalar @a, scalar @b, scalar @c );
```

There is no equivalent operator to force an expression to be interpolated in list context because in practice, this is never needed. If you really wanted to do so, however, you could use the construction `@{[ (some expression) ]}`, but usually a simple `(some expression)` suffices.

Because `scalar` is a unary operator, if you accidentally use a parenthesized list for the EXPR, this behaves as a scalar comma expression, evaluating all but the last element in void context and returning the final element evaluated in scalar context. This is seldom what you want.

The following single statement:

```
print uc(scalar(&foo,$bar)),$baz;
```

is the moral equivalent of these two:

```
&foo;
print(uc($bar),$baz);
```

See Section 48.1 [perlop NAME], page 798 for more details on unary operators and the comma operator.

seek FILEHANDLE,POSITION,WHENCE

Sets FILEHANDLE's position, just like the `fseek` call of `stdio`. FILEHANDLE may be an expression whose value gives the name of the filehandle. The values for WHENCE are `0` to set the new position *in bytes* to POSITION; `1` to set it to the current position plus POSITION; and `2` to set it to EOF plus POSITION, typically negative. For WHENCE you may use the constants `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` (start of the file, current position, end of the file) from the `Fcntl` module. Returns `1` on success, false otherwise.

Note the *in bytes*: even if the filehandle has been set to operate on characters (for example by using the `:encoding(utf8)` open layer), tell() will return byte offsets, not character offsets (because implementing that would render seek() and tell() rather slow).

If you want to position the file for `sysread` or `syswrite`, don't use `seek`, because buffering makes its effect on the file's read-write position unpredictable and non-portable. Use `sysseek` instead.

Due to the rules and rigors of ANSI C, on some systems you have to do a seek whenever you switch between reading and writing. Amongst other things, this may have the effect of calling stdio's clearerr(3). A WHENCE of `1` (`SEEK_CUR`) is useful for not moving the file position:

```
seek(TEST,0,1);
```

This is also useful for applications emulating `tail -f`. Once you hit EOF on your read and then sleep for a while, you (probably) have to stick in a dummy seek() to reset things. The `seek` doesn't change the position, but it *does* clear the end-of-file condition on the handle, so that the next `<FILE>` makes Perl try again to read something. (We hope.)

If that doesn't work (some I/O implementations are particularly cantankerous), you might need something like this:

```
for (;;) {
    for ($curpos = tell(FILE); $_ = <FILE>;
         $curpos = tell(FILE)) {
        # search for some stuff and put it into files
    }
    sleep($for_a_while);
    seek(FILE, $curpos, 0);
}
```

seekdir DIRHANDLE,POS

Sets the current position for the `readdir` routine on DIRHANDLE. POS must be a value returned by `telldir`. `seekdir` also has the same caveats about possible directory compaction as the corresponding system library routine.

select FILEHANDLE
select

Returns the currently selected filehandle. If FILEHANDLE is supplied, sets the new current default filehandle for output. This has two effects: first, a `write` or a `print` without a filehandle default to this FILEHANDLE. Second, references to variables related to output will refer to this output channel.

For example, to set the top-of-form format for more than one output channel, you might do the following:

```
select(REPORT1);
$^ = 'report1_top';
select(REPORT2);
$^ = 'report2_top';
```

FILEHANDLE may be an expression whose value gives the name of the actual filehandle. Thus:

```
$oldfh = select(STDERR); $| = 1; select($oldfh);
```

Some programmers may prefer to think of filehandles as objects with methods, preferring to write the last example as:

```
use IO::Handle;
STDERR->autoflush(1);
```

Portability issues: [perlport select], page 977.

select RBITS,WBITS,EBITS,TIMEOUT

This calls the select(2) syscall with the bit masks specified, which can be constructed using `fileno` and `vec`, along these lines:

```
$rin = $win = $ein = '';
vec($rin, fileno(STDIN),  1) = 1;
vec($win, fileno(STDOUT), 1) = 1;
$ein = $rin | $win;
```

If you want to select on many filehandles, you may wish to write a subroutine like this:

```
sub fhbits {
    my @fhlist = @_;
    my $bits = "";
    for my $fh (@fhlist) {
        vec($bits, fileno($fh), 1) = 1;
    }
    return $bits;
}
$rin = fhbits(*STDIN, *TTY, *MYSOCK);
```

The usual idiom is:

```
($nfound,$timeleft) =
  select($rout=$rin, $wout=$win, $eout=$ein, $timeout);
```

or to block until something becomes ready just do this

```
$nfound = select($rout=$rin, $wout=$win, $eout=$ein, undef);
```

Most systems do not bother to return anything useful in $timeleft, so calling select() in scalar context just returns $nfound.

Any of the bit masks can also be undef. The timeout, if specified, is in seconds, which may be fractional. Note: not all implementations are capable of returning the $timeleft. If not, they always return $timeleft equal to the supplied $timeout.

You can effect a sleep of 250 milliseconds this way:

```
select(undef, undef, undef, 0.25);
```

Note that whether `select` gets restarted after signals (say, SIGALRM) is implementation-dependent. See also Section 56.1 [perlport NAME], page 951 for notes on the portability of `select`.

On error, `select` behaves just like select(2): it returns -1 and sets `$!`.

On some Unixes, select(2) may report a socket file descriptor as "ready for reading" even when no data is available, and thus any subsequent `read` would block. This can be avoided if you always use O_NONBLOCK on the socket. See select(2) and fcntl(2) for further details.

The standard `IO::Select` module provides a user-friendlier interface to `select`, mostly because it does all the bit-mask work for you.

WARNING: One should not attempt to mix buffered I/O (like `read` or `<FH>`) with `select`, except as permitted by POSIX, and even then only on POSIX systems. You have to use `sysread` instead.

Portability issues: [perlport select], page 977.

semctl ID,SEMNUM,CMD,ARG

Calls the System V IPC function semctl(2). You'll probably have to say

```
use IPC::SysV;
```

first to get the correct constant definitions. If CMD is IPC_STAT or GETALL, then ARG must be a variable that will hold the returned semid_ds structure or semaphore value array. Returns like `ioctl`: the undefined value for error, `"0 but true"` for zero, or the actual return value otherwise. The ARG must consist of a vector of native short integers, which may be created with `pack("s!",(0)x$nsem)`. See also Section 36.10 [perlipc SysV IPC], page 696, `IPC::SysV`, `IPC::Semaphore` documentation.

Portability issues: [perlport semctl], page 977.

semget KEY,NSEMS,FLAGS

Calls the System V IPC function semget(2). Returns the semaphore id, or the undefined value on error. See also Section 36.10 [perlipc SysV IPC], page 696, `IPC::SysV`, `IPC::SysV::Semaphore` documentation.

Portability issues: [perlport semget], page 977.

semop KEY,OPSTRING

Calls the System V IPC function semop(2) for semaphore operations such as signalling and waiting. OPSTRING must be a packed array of semop structures. Each semop structure can be generated with `pack("s!3", $semnum, $semop, $semflag)`. The length of OPSTRING implies the number of semaphore operations. Returns true if successful, false on error. As an example, the following code waits on semaphore $semnum of semaphore id $semid:

```
$semop = pack("s!3", $semnum, -1, 0);
die "Semaphore trouble: $!\n" unless semop($semid, $semop);
```

To signal the semaphore, replace `-1` with `1`. See also Section 36.10 [perlipc SysV IPC], page 696, `IPC::SysV`, and `IPC::SysV::Semaphore` documentation.

Portability issues: [perlport semop], page 977.

send SOCKET,MSG,FLAGS,TO
send SOCKET,MSG,FLAGS

Sends a message on a socket. Attempts to send the scalar MSG to the SOCKET filehandle. Takes the same flags as the system call of the same name. On unconnected sockets, you must specify a destination to *send to*, in which case it does a sendto(2) syscall. Returns the number of characters sent, or the undefined value on error. The sendmsg(2) syscall is currently unimplemented. See Section 36.9 [perlipc UDP: Message Passing], page 695 for examples.

Note the *characters*: depending on the status of the socket, either (8-bit) bytes or characters are sent. By default all sockets operate on bytes, but for example if the socket has been changed using binmode() to operate with the `:encoding(utf8)` I/O layer (see ⟨undefined⟩ [open], page ⟨undefined⟩, or the `open` pragma, `open`), the I/O will operate on UTF-8 encoded Unicode characters, not bytes. Similarly for the `:encoding` pragma: in that case pretty much any characters can be sent.

setpgrp PID,PGRP

Sets the current process group for the specified PID, `0` for the current process. Raises an exception when used on a machine that doesn't implement POSIX setpgid(2) or BSD setpgrp(2). If the arguments are omitted, it defaults to `0,0`. Note that the BSD 4.2 version of `setpgrp` does not accept any arguments, so only `setpgrp(0,0)` is portable. See also `POSIX::setsid()`.

Portability issues: [perlport setpgrp], page 978.

setpriority WHICH,WHO,PRIORITY

Sets the current priority for a process, a process group, or a user. (See setpriority(2).) Raises an exception when used on a machine that doesn't implement setpriority(2).

Portability issues: [perlport setpriority], page 978.

setsockopt SOCKET,LEVEL,OPTNAME,OPTVAL

Sets the socket option requested. Returns `undef` on error. Use integer constants provided by the `Socket` module for LEVEL and OPNAME. Values for LEVEL can also be obtained from getprotobyname. OPTVAL might either be a packed string or an integer. An integer OPTVAL is shorthand for pack(`"i"`, OPTVAL).

An example disabling Nagle's algorithm on a socket:

```
use Socket qw(IPPROTO_TCP TCP_NODELAY);
setsockopt($socket, IPPROTO_TCP, TCP_NODELAY, 1);
```

Portability issues: [perlport setsockopt], page 978.

shift ARRAY
shift EXPR
shift

Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down. If there are no elements in the array, returns the undefined value. If ARRAY is omitted, shifts the `@_` array within the lexical scope of subroutines and formats, and the `@ARGV` array outside a subroutine and also within the lexical scopes established by the `eval STRING`, `BEGIN {}`, `INIT {}`, `CHECK {}`, `UNITCHECK {}`, and `END {}` constructs.

Starting with Perl 5.14, `shift` can take a scalar EXPR, which must hold a reference to an unblessed array. The argument will be dereferenced automatically. This aspect of `shift` is considered highly experimental. The exact behaviour may change in a future version of Perl.

To avoid confusing would-be users of your code who are running earlier versions of Perl with mysterious syntax errors, put this sort of thing at the top of your file to signal that your code will work *only* on Perls of a recent vintage:

```
use 5.014;  # so push/pop/etc work on scalars (experimental)
```

See also `unshift`, `push`, and `pop`. `shift` and `unshift` do the same thing to the left end of an array that `pop` and `push` do to the right end.

shmctl ID,CMD,ARG

Calls the System V IPC function shmctl. You'll probably have to say

```
use IPC::SysV;
```

first to get the correct constant definitions. If CMD is `IPC_STAT`, then ARG must be a variable that will hold the returned `shmid_ds` structure. Returns like ioctl: `undef` for error; `"0 but true"` for zero; and the actual return value otherwise. See also Section 36.10 [perlipc SysV IPC], page 696 and `IPC::SysV` documentation.

Portability issues: [perlport shmctl], page 978.

shmget KEY,SIZE,FLAGS

Calls the System V IPC function shmget. Returns the shared memory segment id, or `undef` on error. See also Section 36.10 [perlipc SysV IPC], page 696 and `IPC::SysV` documentation.

Portability issues: [perlport shmget], page 978.

shmread ID,VAR,POS,SIZE
shmwrite ID,STRING,POS,SIZE

Reads or writes the System V shared memory segment ID starting at position POS for size SIZE by attaching to it, copying in/out, and detaching from it. When reading, VAR must be a variable that will hold the data read. When writing, if STRING is too long, only SIZE bytes are used; if STRING is too short, nulls are written to fill out SIZE bytes. Return true if successful, false on error. shmread() taints the variable. See also Section 36.10 [perlipc SysV IPC], page 696, `IPC::SysV`, and the `IPC::Shareable` module from CPAN.

Portability issues: [perlport shmread], page 978 and [perlport shmwrite], page 978.

shutdown SOCKET,HOW

Shuts down a socket connection in the manner indicated by HOW, which has the same interpretation as in the syscall of the same name.

```
shutdown(SOCKET, 0);    # I/we have stopped reading data
shutdown(SOCKET, 1);    # I/we have stopped writing data
shutdown(SOCKET, 2);    # I/we have stopped using this socket
```

This is useful with sockets when you want to tell the other side you're done writing but not done reading, or vice versa. It's also a more insistent form of close because it also disables the file descriptor in any forked copies in other processes.

Returns `1` for success; on error, returns `undef` if the first argument is not a valid filehandle, or returns `0` and sets `$!` for any other failure.

sin EXPR

sin

Returns the sine of EXPR (expressed in radians). If EXPR is omitted, returns sine of `$_`.

For the inverse sine operation, you may use the `Math::Trig::asin` function, or use this relation:

```
sub asin { atan2($_[0], sqrt(1 - $_[0] * $_[0])) }
```

sleep EXPR
sleep

Causes the script to sleep for (integer) EXPR seconds, or forever if no argument is given. Returns the integer number of seconds actually slept.

May be interrupted if the process receives a signal such as `SIGALRM`.

```
    eval {
        local $SIG{ALARM} = sub { die "Alarm!\n" };
        sleep;
    };
    die $@ unless $@ eq "Alarm!\n";
```

You probably cannot mix `alarm` and `sleep` calls, because `sleep` is often implemented using `alarm`.

On some older systems, it may sleep up to a full second less than what you requested, depending on how it counts seconds. Most modern systems always sleep the full amount. They may appear to sleep longer than that, however, because your process might not be scheduled right away in a busy multitasking system.

For delays of finer granularity than one second, the Time::HiRes module (from CPAN, and starting from Perl 5.8 part of the standard distribution) provides usleep(). You may also use Perl's four-argument version of select() leaving the first three arguments undefined, or you might be able to use the `syscall` interface to access setitimer(2) if your system supports it. See `perlfaq8` for details.

See also the POSIX module's `pause` function.

socket SOCKET,DOMAIN,TYPE,PROTOCOL

Opens a socket of the specified kind and attaches it to filehandle SOCKET. DOMAIN, TYPE, and PROTOCOL are specified the same as for the syscall of the same name. You should `use Socket` first to get the proper definitions imported. See the examples in Section 36.6 [perlipc Sockets: Client/Server Communication], page 682.

On systems that support a close-on-exec flag on files, the flag will be set for the newly opened file descriptor, as determined by the value of $^F. See [perlvar $^F], page 1381.

socketpair SOCKET1,SOCKET2,DOMAIN,TYPE,PROTOCOL

Creates an unnamed pair of sockets in the specified domain, of the specified type. DOMAIN, TYPE, and PROTOCOL are specified the same as for the syscall of the same name. If unimplemented, raises an exception. Returns true if successful.

On systems that support a close-on-exec flag on files, the flag will be set for the newly opened file descriptors, as determined by the value of $^F. See [perlvar $^F], page 1381.

Some systems defined `pipe` in terms of `socketpair`, in which a call to `pipe(Rdr, Wtr)` is essentially:

```
use Socket;
socketpair(Rdr, Wtr, AF_UNIX, SOCK_STREAM, PF_UNSPEC);
shutdown(Rdr, 1);        # no more writing for reader
shutdown(Wtr, 0);        # no more reading for writer
```

See Section 36.1 [perlipc NAME], page 667 for an example of socketpair use. Perl 5.8 and later will emulate socketpair using IP sockets to localhost if your system implements sockets but not socketpair.

Portability issues: [perlport socketpair], page 978.

sort SUBNAME LIST
sort BLOCK LIST
sort LIST

In list context, this sorts the LIST and returns the sorted list value. In scalar context, the behaviour of `sort()` is undefined.

If SUBNAME or BLOCK is omitted, `sort`s in standard string comparison order. If SUBNAME is specified, it gives the name of a subroutine that returns an integer less than, equal to, or greater than `0`, depending on how the elements of the list are to be ordered. (The `<=>` and `cmp` operators are extremely useful in such routines.) SUBNAME may be a scalar variable name (unsubscripted), in which case the value provides the name of (or a reference to) the actual subroutine to use. In place of a SUBNAME, you can provide a BLOCK as an anonymous, in-line sort subroutine.

If the subroutine's prototype is `($$)`, the elements to be compared are passed by reference in `@_`, as for a normal subroutine. This is slower than unprototyped subroutines, where the elements to be compared are passed into the subroutine as the package global variables $a and $b (see example below). Note that in the latter case, it is usually highly counter-productive to declare $a and $b as lexicals.

If the subroutine is an XSUB, the elements to be compared are pushed on to the stack, the way arguments are usually passed to XSUBs. $a and $b are not set.

The values to be compared are always passed by reference and should not be modified.

You also cannot exit out of the sort block or subroutine using any of the loop control operators described in Section 74.1 [perlsyn NAME], page 1249 or with `goto`.

When `use locale` (but not `use locale 'not_characters'`) is in effect, `sort LIST` sorts LIST according to the current collation locale. See Section 38.1 [perllocale NAME], page 701.

sort() returns aliases into the original list, much as a for loop's index variable aliases the list elements. That is, modifying an element of a list returned by sort() (for example, in a `foreach`, `map` or `grep`) actually modifies the element in the original list. This is usually something to be avoided when writing clear code.

Perl 5.6 and earlier used a quicksort algorithm to implement sort. That algorithm was not stable, so *could* go quadratic. (A *stable* sort preserves the input order of elements that compare equal. Although quicksort's run time is O(NlogN) when averaged over all arrays of length N, the time can be O(N**2), *quadratic* behavior, for some inputs.) In 5.7, the quicksort implementation was replaced with a stable mergesort algorithm whose worst-case behavior is O(NlogN). But benchmarks indicated that for some inputs, on some platforms, the original quicksort was faster. 5.8 has a sort pragma for limited control of the sort. Its rather blunt control of the underlying algorithm may not persist into future Perls, but the ability to characterize the input or output in implementation independent ways quite probably will. See `sort`.

Examples:

```
# sort lexically
@articles = sort @files;

# same thing, but with explicit sort routine
@articles = sort {$a cmp $b} @files;

# now case-insensitively
@articles = sort {fc($a) cmp fc($b)} @files;

# same thing in reversed order
@articles = sort {$b cmp $a} @files;

# sort numerically ascending
@articles = sort {$a <=> $b} @files;

# sort numerically descending
@articles = sort {$b <=> $a} @files;

# this sorts the %age hash by value instead of key
# using an in-line function
@eldest = sort { $age{$b} <=> $age{$a} } keys %age;

# sort using explicit subroutine name
sub byage {
    $age{$a} <=> $age{$b};  # presuming numeric
}
@sortedclass = sort byage @class;

sub backwards { $b cmp $a }
@harry  = qw(dog cat x Cain Abel);
@george = qw(gone chased yz Punished Axed);
print sort @harry;
    # prints AbelCaincatdogx
print sort backwards @harry;
    # prints xdogcatCainAbel
print sort @george, 'to', @harry;
```

```
    # prints AbelAxedCainPunishedcatchaseddoggonetoxyz

# inefficiently sort by descending numeric compare using
# the first integer after the first = sign, or the
# whole record case-insensitively otherwise

my @new = sort {
    ($b =~ /=(\d+)/)[0] <=> ($a =~ /=(\d+)/)[0]
                        ||
                fc($a)  cmp  fc($b)
} @old;

# same thing, but much more efficiently;
# we'll build auxiliary indices instead
# for speed
my @nums = @caps = ();
for (@old) {
    push @nums, ( /=(\d+)/ ? $1 : undef );
    push @caps, fc($_);
}

my @new = @old[ sort {
                       $nums[$b] <=> $nums[$a]
                                ||
                       $caps[$a] cmp $caps[$b]
                     } 0..$#old
              ];

# same thing, but without any temps
@new = map { $_->[0] }
       sort { $b->[1] <=> $a->[1]
                       ||
              $a->[2] cmp $b->[2]
       } map { [$_, /=(\d+)/, fc($_)] } @old;

# using a prototype allows you to use any comparison subroutine
# as a sort subroutine (including other package's subroutines)
package other;
sub backwards ($$) { $_[1] cmp $_[0]; }  # $a and $b are
                                         # not set here
package main;
@new = sort other::backwards @old;

# guarantee stability, regardless of algorithm
use sort 'stable';
@new = sort { substr($a, 3, 5) cmp substr($b, 3, 5) } @old;
```

```
# force use of mergesort (not portable outside Perl 5.8)
use sort '_mergesort';  # note discouraging _
@new = sort { substr($a, 3, 5) cmp substr($b, 3, 5) } @old;
```

Warning: syntactical care is required when sorting the list returned from a function. If you want to sort the list returned by the function call `find_records(@key)`, you can use:

```
@contact = sort { $a cmp $b } find_records @key;
@contact = sort +find_records(@key);
@contact = sort &find_records(@key);
@contact = sort(find_records(@key));
```

If instead you want to sort the array @key with the comparison routine `find_records()` then you can use:

```
@contact = sort { find_records() } @key;
@contact = sort find_records(@key);
@contact = sort(find_records @key);
@contact = sort(find_records (@key));
```

If you're using strict, you *must not* declare $a and $b as lexicals. They are package globals. That means that if you're in the `main` package and type

```
@articles = sort {$b <=> $a} @files;
```

then `$a` and `$b` are `$main::a` and `$main::b` (or `$::a` and `$::b`), but if you're in the `FooPack` package, it's the same as typing

```
@articles = sort {$FooPack::b <=> $FooPack::a} @files;
```

The comparison function is required to behave. If it returns inconsistent results (sometimes saying `$x[1]` is less than `$x[2]` and sometimes saying the opposite, for example) the results are not well-defined.

Because `<=>` returns `undef` when either operand is `NaN` (not-a-number), be careful when sorting with a comparison function like `$a <=> $b` any lists that might contain a `NaN`. The following example takes advantage that `NaN != NaN` to eliminate any `NaN`s from the input list.

```
@result = sort { $a <=> $b } grep { $_ == $_ } @input;
```

splice ARRAY,OFFSET,LENGTH,LIST
splice ARRAY,OFFSET,LENGTH
splice ARRAY,OFFSET
splice ARRAY
splice EXPR,OFFSET,LENGTH,LIST
splice EXPR,OFFSET,LENGTH
splice EXPR,OFFSET
splice EXPR

Removes the elements designated by OFFSET and LENGTH from an array, and replaces them with the elements of LIST, if any. In list context, returns the elements removed from the array. In scalar context, returns the last element removed, or `undef` if no elements are removed. The array grows or shrinks as necessary. If OFFSET is negative then it starts that far from the end of the array. If LENGTH is omitted, removes everything from OFFSET onward. If

LENGTH is negative, removes the elements from OFFSET onward except for -LENGTH elements at the end of the array. If both OFFSET and LENGTH are omitted, removes everything. If OFFSET is past the end of the array and a LENGTH was provided, Perl issues a warning, and splices at the end of the array.

The following equivalences hold (assuming `$#a >= $i`)

```
push(@a,$x,$y)      splice(@a,@a,0,$x,$y)
pop(@a)             splice(@a,-1)
shift(@a)           splice(@a,0,1)
unshift(@a,$x,$y)   splice(@a,0,0,$x,$y)
$a[$i] = $y         splice(@a,$i,1,$y)
```

`splice` can be used, for example, to implement n-ary queue processing:

```
sub nary_print {
  my $n = shift;
  while (my @next_n = splice @_, 0, $n) {
    say join q{ -- }, @next_n;
  }
}

nary_print(3, qw(a b c d e f g h));
# prints:
#   a -- b -- c
#   d -- e -- f
#   g -- h
```

Starting with Perl 5.14, `splice` can take scalar EXPR, which must hold a reference to an unblessed array. The argument will be dereferenced automatically. This aspect of `splice` is considered highly experimental. The exact behaviour may change in a future version of Perl.

To avoid confusing would-be users of your code who are running earlier versions of Perl with mysterious syntax errors, put this sort of thing at the top of your file to signal that your code will work *only* on Perls of a recent vintage:

```
use 5.014;  # so push/pop/etc work on scalars (experimental)
```

split /PATTERN/,EXPR,LIMIT
split /PATTERN/,EXPR
split /PATTERN/
split

Splits the string EXPR into a list of strings and returns the list in list context, or the size of the list in scalar context.

If only PATTERN is given, EXPR defaults to `$_`.

Anything in EXPR that matches PATTERN is taken to be a separator that separates the EXPR into substrings (called "*fields*") that do **not** include the separator. Note that a separator may be longer than one character or even have no characters at all (the empty string, which is a zero-width match).

The PATTERN need not be constant; an expression may be used to specify a pattern that varies at runtime.

If PATTERN matches the empty string, the EXPR is split at the match position (between characters). As an example, the following:

```
    print join(':', split('b', 'abc')), "\n";
```

uses the 'b' in 'abc' as a separator to produce the output 'a:c'. However, this:

```
    print join(':', split('', 'abc')), "\n";
```

uses empty string matches as separators to produce the output 'a:b:c'; thus, the empty string may be used to split EXPR into a list of its component characters.

As a special case for `split`, the empty pattern given in [match operator], page 825 syntax (`//`) specifically matches the empty string, which is contrary to its usual interpretation as the last successful match.

If PATTERN is `/^/`, then it is treated as if it used the Section 67.2.1 [multiline modifier], page 1124 (`/^/m`), since it isn't much use otherwise.

As another special case, `split` emulates the default behavior of the command line tool **awk** when the PATTERN is either omitted or a *literal string* composed of a single space character (such as `' '` or `"\x20"`, but not e.g. `/ /`). In this case, any leading whitespace in EXPR is removed before splitting occurs, and the PATTERN is instead treated as if it were `/\s+/`; in particular, this means that *any* contiguous whitespace (not just a single space character) is used as a separator. However, this special treatment can be avoided by specifying the pattern `/ /` instead of the string `" "`, thereby allowing only a single space character to be a separator. In earlier Perls this special case was restricted to the use of a plain `" "` as the pattern argument to split, in Perl 5.18.0 and later this special case is triggered by any expression which evaluates as the simple string `" "`.

If omitted, PATTERN defaults to a single space, `" "`, triggering the previously described *awk* emulation.

If LIMIT is specified and positive, it represents the maximum number of fields into which the EXPR may be split; in other words, LIMIT is one greater than the maximum number of times EXPR may be split. Thus, the LIMIT value `1` means that EXPR may be split a maximum of zero times, producing a maximum of one field (namely, the entire value of EXPR). For instance:

```
    print join(':', split(//, 'abc', 1)), "\n";
```

produces the output 'abc', and this:

```
    print join(':', split(//, 'abc', 2)), "\n";
```

produces the output 'a:bc', and each of these:

```
    print join(':', split(//, 'abc', 3)), "\n";
    print join(':', split(//, 'abc', 4)), "\n";
```

produces the output 'a:b:c'.

If LIMIT is negative, it is treated as if it were instead arbitrarily large; as many fields as possible are produced.

If LIMIT is omitted (or, equivalently, zero), then it is usually treated as if it were instead negative but with the exception that trailing empty fields are stripped (empty leading fields are always preserved); if all fields are empty, then all fields are considered to be trailing (and are thus stripped in this case). Thus, the following:

```
print join(':', split(',', 'a,b,c,,,')), "\n";
```

produces the output 'a:b:c', but the following:

```
print join(':', split(',', 'a,b,c,,,', -1)), "\n";
```

produces the output 'a:b:c:::'.

In time-critical applications, it is worthwhile to avoid splitting into more fields than necessary. Thus, when assigning to a list, if LIMIT is omitted (or zero), then LIMIT is treated as though it were one larger than the number of variables in the list; for the following, LIMIT is implicitly 3:

```
($login, $passwd) = split(/:/);
```

Note that splitting an EXPR that evaluates to the empty string always produces zero fields, regardless of the LIMIT specified.

An empty leading field is produced when there is a positive-width match at the beginning of EXPR. For instance:

```
print join(':', split(/ /, ' abc')), "\n";
```

produces the output ':abc'. However, a zero-width match at the beginning of EXPR never produces an empty field, so that:

```
print join(':', split(//, ' abc'));
```

produces the output ' :a:b:c' (rather than ': :a:b:c').

An empty trailing field, on the other hand, is produced when there is a match at the end of EXPR, regardless of the length of the match (of course, unless a non-zero LIMIT is given explicitly, such fields are removed, as in the last example). Thus:

```
print join(':', split(//, ' abc', -1)), "\n";
```

produces the output ' :a:b:c:'.

If the PATTERN contains Section 68.3.4 [capturing groups], page 1140, then for each separator, an additional field is produced for each substring captured by a group (in the order in which the groups are specified, as per Section 68.3.6 [backreferences], page 1142); if any group does not match, then it captures the `undef` value instead of a substring. Also, note that any such additional field is produced whenever there is a separator (that is, whenever a split occurs), and such an additional field does **not** count towards the LIMIT. Consider the following expressions evaluated in list context (each returned list is provided in the associated comment):

```
split(/-|,/, "1-10,20", 3)
# ('1', '10', '20')

split(/(-|,)/, "1-10,20", 3)
# ('1', '-', '10', ',', '20')

split(/-|(,)/, "1-10,20", 3)
# ('1', undef, '10', ',', '20')
```

```
split(/(-)|,/, "1-10,20", 3)
# ('1', '-', '10', undef, '20')

split(/(-)|(,)/, "1-10,20", 3)
# ('1', '-', undef, '10', undef, ',', '20')
```

sprintf FORMAT, LIST

Returns a string formatted by the usual `printf` conventions of the C library function `sprintf`. See below for more details and see sprintf(3) or printf(3) on your system for an explanation of the general principles.

For example:

```
# Format number with up to 8 leading zeroes
$result = sprintf("%08d", $number);

# Round number to 3 digits after decimal point
$rounded = sprintf("%.3f", $number);
```

Perl does its own `sprintf` formatting: it emulates the C function sprintf(3), but doesn't use it except for floating-point numbers, and even then only standard modifiers are allowed. Non-standard extensions in your local sprintf(3) are therefore unavailable from Perl.

Unlike `printf`, `sprintf` does not do what you probably mean when you pass it an array as your first argument. The array is given scalar context, and instead of using the 0th element of the array as the format, Perl will use the count of elements in the array as the format, which is almost never useful.

Perl's `sprintf` permits the following universally-known conversions:

```
%%    a percent sign
%c    a character with the given number
%s    a string
%d    a signed integer, in decimal
%u    an unsigned integer, in decimal
%o    an unsigned integer, in octal
%x    an unsigned integer, in hexadecimal
%e    a floating-point number, in scientific notation
%f    a floating-point number, in fixed decimal notation
%g    a floating-point number, in %e or %f notation
```

In addition, Perl permits the following widely-supported conversions:

```
%X    like %x, but using upper-case letters
%E    like %e, but using an upper-case "E"
%G    like %g, but with an upper-case "E" (if applicable)
%b    an unsigned integer, in binary
%B    like %b, but using an upper-case "B" with the # flag
%p    a pointer (outputs the Perl value's address in hexadecimal)
%n    special: *stores* the number of characters output so far
      into the next argument in the parameter list
%a    hexadecimal floating point
```

```
%A    like %a, but using upper-case letters
```

Finally, for backward (and we do mean "backward") compatibility, Perl permits these unnecessary but widely-supported conversions:

```
%i    a synonym for %d
%D    a synonym for %ld
%U    a synonym for %lu
%O    a synonym for %lo
%F    a synonym for %f
```

Note that the number of exponent digits in the scientific notation produced by %e, %E, %g and %G for numbers with the modulus of the exponent less than 100 is system-dependent: it may be three or less (zero-padded as necessary). In other words, 1.23 times ten to the 99th may be either "1.23e99" or "1.23e099". Similarly for %a and %A: the exponent or the hexadecimal digits may float: especially the "long doubles" Perl configuration option may cause surprises.

Between the % and the format letter, you may specify several additional attributes controlling the interpretation of the format. In order, these are:

format parameter index

An explicit format parameter index, such as 2$. By default sprintf will format the next unused argument in the list, but this allows you to take the arguments out of order:

```
printf '%2$d %1$d', 12, 34;      # prints "34 12"
printf '%3$d %d %1$d', 1, 2, 3;  # prints "3 1 1"
```

flags

one or more of:

```
space   prefix non-negative number with a space
+       prefix non-negative number with a plus sign
-       left-justify within the field
0       use zeros, not spaces, to right-justify
#       ensure the leading "0" for any octal,
        prefix non-zero hexadecimal with "0x" or "0X",
        prefix non-zero binary with "0b" or "0B"
```

For example:

```
printf '<% d>',  12;   # prints "< 12>"
printf '<%+d>',  12;   # prints "<+12>"
printf '<%6s>',  12;   # prints "<    12>"
printf '<%-6s>', 12;   # prints "<12    >"
printf '<%06s>', 12;   # prints "<000012>"
printf '<%#o>',  12;   # prints "<014>"
printf '<%#x>',  12;   # prints "<0xc>"
printf '<%#X>',  12;   # prints "<0XC>"
printf '<%#b>',  12;   # prints "<0b1100>"
printf '<%#B>',  12;   # prints "<0B1100>"
```

When a space and a plus sign are given as the flags at once, a plus sign is used to prefix a positive number.

```
printf '<%+ d>', 12;   # prints "<+12>"
printf '<% +d>', 12;   # prints "<+12>"
```

When the # flag and a precision are given in the %o conversion, the precision is incremented if it's necessary for the leading "0".

```
printf '<%#.5o>', 012;      # prints "<00012>"
printf '<%#.5o>', 012345;   # prints "<012345>"
printf '<%#.0o>', 0;        # prints "<0>"
```

vector flag

This flag tells Perl to interpret the supplied string as a vector of integers, one for each character in the string. Perl applies the format to each integer in turn, then joins the resulting strings with a separator (a dot . by default). This can be useful for displaying ordinal values of characters in arbitrary strings:

```
printf "%vd", "AB\x{100}";           # prints "65.66.256"
printf "version is v%vd\n", $^V;     # Perl's version
```

Put an asterisk * before the v to override the string to use to separate the numbers:

```
printf "address is %*vX\n", ":", $addr;   # IPv6 address
printf "bits are %0*v8b\n", " ", $bits;   # random bitstring
```

You can also explicitly specify the argument number to use for the join string using something like *2$v; for example:

```
printf '%*4$vX %*4$vX %*4$vX',       # 3 IPv6 addresses
        @addr[1..3], ":";
```

(minimum) width

Arguments are usually formatted to be only as wide as required to display the given value. You can override the width by putting a number here, or get the width from the next argument (with *) or from a specified argument (e.g., with *2$):

```
printf "<%s>", "a";       # prints "<a>"
printf "<%6s>", "a";      # prints "<     a>"
printf "<%*s>", 6, "a";   # prints "<     a>"
printf '<%*2$s>', "a", 6; # prints "<     a>"
printf "<%2s>", "long";   # prints "<long>" (does not truncate)
```

If a field width obtained through * is negative, it has the same effect as the - flag: left-justification.

precision, or maximum width

You can specify a precision (for numeric conversions) or a maximum width (for string conversions) by specifying a . followed by a number. For floating-point formats except g and G, this specifies how many places right of the decimal point to show (the default being 6). For example:

```
# these examples are subject to system-specific variation
printf '<%f>', 1;    # prints "<1.000000>"
printf ’<%.1f>’, 1;  # prints "<1.0>"
printf ’<%.0f>’, 1;  # prints "<1>"
printf ’<%e>’, 10;   # prints "<1.000000e+01>"
printf ’<%.1e>’, 10; # prints "<1.0e+01>"
```

For "g" and "G", this specifies the maximum number of digits to show, including those prior to the decimal point and those after it; for example:

```
# These examples are subject to system-specific variation.
printf ’<%g>’, 1;        # prints "<1>"
printf ’<%.10g>’, 1;     # prints "<1>"
printf ’<%g>’, 100;      # prints "<100>"
printf ’<%.1g>’, 100;    # prints "<1e+02>"
printf ’<%.2g>’, 100.01; # prints "<1e+02>"
printf ’<%.5g>’, 100.01; # prints "<100.01>"
printf ’<%.4g>’, 100.01; # prints "<100>"
```

For integer conversions, specifying a precision implies that the output of the number itself should be zero-padded to this width, where the 0 flag is ignored:

```
printf ’<%.6d>’, 1;      # prints "<000001>"
printf ’<%+.6d>’, 1;     # prints "<+000001>"
printf ’<%-10.6d>’, 1;   # prints "<000001    >"
printf ’<%10.6d>’, 1;    # prints "<    000001>"
printf ’<%010.6d>’, 1;   # prints "<    000001>"
printf ’<%+10.6d>’, 1;   # prints "<   +000001>"

printf ’<%.6x>’, 1;      # prints "<000001>"
printf ’<%#.6x>’, 1;     # prints "<0x000001>"
printf ’<%-10.6x>’, 1;   # prints "<000001    >"
printf ’<%10.6x>’, 1;    # prints "<    000001>"
printf ’<%010.6x>’, 1;   # prints "<    000001>"
printf ’<%#10.6x>’, 1;   # prints "<  0x000001>"
```

For string conversions, specifying a precision truncates the string to fit the specified width:

```
printf ’<%.5s>’, "truncated";   # prints "<trunc>"
printf ’<%10.5s>’, "truncated"; # prints "<     trunc>"
```

You can also get the precision from the next argument using `.*`:

```
printf ’<%.6x>’, 1;       # prints "<000001>"
printf ’<%.*x>’, 6, 1;    # prints "<000001>"
```

If a precision obtained through `*` is negative, it counts as having no precision at all.

```
printf ’<%.*s>’,  7, "string";   # prints "<string>"
printf ’<%.*s>’,  3, "string";   # prints "<str>"
printf ’<%.*s>’,  0, "string";   # prints "<>"
printf ’<%.*s>’, -1, "string";   # prints "<string>"
printf '<%.*d>',  1, 0;   # prints "<0>"
printf '<%.*d>',  0, 0;   # prints "<>"
printf '<%.*d>', -1, 0;   # prints "<0>"
```

You cannot currently get the precision from a specified number, but it is intended that this will be possible in the future, for example using `.*2$`:

```
printf '<%.*2$x>', 1, 6;   # INVALID, but in future will print
                           # "<000001>"
```

size

For numeric conversions, you can specify the size to interpret the number as using `l`, `h`, `V`, `q`, `L`, or `ll`. For integer conversions (`d u o x X b i D U O`), numbers are usually assumed to be whatever the default integer size is on your platform (usually 32 or 64 bits), but you can override this to use instead one of the standard C types, as supported by the compiler used to build Perl:

```
hh          interpret integer as C type "char" or "unsigned
            char" on Perl 5.14 or later
h           interpret integer as C type "short" or
            "unsigned short"
j           interpret integer as C type "intmax_t" on Perl
            5.14 or later, and only with a C99 compiler
            (unportable)
l           interpret integer as C type "long" or
            "unsigned long"
q, L, or ll interpret integer as C type "long long",
            "unsigned long long", or "quad" (typically
            64-bit integers)
t           interpret integer as C type "ptrdiff_t" on Perl
            5.14 or later
z           interpret integer as C type "size_t" on Perl 5.14
            or later
```

As of 5.14, none of these raises an exception if they are not supported on your platform. However, if warnings are enabled, a warning of the `printf` warning class is issued on an unsupported conversion flag. Should you instead prefer an exception, do this:

```
use warnings FATAL => "printf";
```

If you would like to know about a version dependency before you start running the program, put something like this at its top:

```
use 5.014;  # for hh/j/t/z/ printf modifiers
```

You can find out whether your Perl supports quads via `Config`:

```
use Config;
if ($Config{use64bitint} eq "define"
    || $Config{longsize} >= 8) {
    print "Nice quads!\n";
    }
```

For floating-point conversions (`e f g E F G`), numbers are usually assumed to be the default floating-point size on your platform (double or long double), but you can force "long double" with `q`, `L`, or `ll` if your platform supports them. You can find out whether your Perl supports long doubles via `Config`:

```
    use Config;
    print "long doubles\n" if $Config{d_longdbl} eq "define";
```

You can find out whether Perl considers "long double" to be the default floating-point size to use on your platform via `Config`:

```
    use Config;
    if ($Config{uselongdouble} eq "define") {
        print "long doubles by default\n";
    }
```

It can also be that long doubles and doubles are the same thing:

```
        use Config;
        ($Config{doublesize} == $Config{longdblsize}) &&
                print "doubles are long doubles\n";
```

The size specifier `V` has no effect for Perl code, but is supported for compatibility with XS code. It means "use the standard size for a Perl integer or floating-point number", which is the default.

order of arguments

Normally, sprintf() takes the next unused argument as the value to format for each format specification. If the format specification uses `*` to require additional arguments, these are consumed from the argument list in the order they appear in the format specification *before* the value to format. Where an argument is specified by an explicit index, this does not affect the normal order for the arguments, even when the explicitly specified index would have been the next argument.

So:

```
    printf "<%*.*s>", $a, $b, $c;
```

uses `$a` for the width, `$b` for the precision, and `$c` as the value to format; while:

```
  printf '<%*1$.*s>', $a, $b;
```

would use `$a` for the width and precision, and `$b` as the value to format.

Here are some more examples; be aware that when using an explicit index, the `$` may need escaping:

```
  printf "%2\$d %d\n",    12, 34;      # will print "34 12\n"
  printf "%2\$d %d %d\n", 12, 34;      # will print "34 12 34\n"
  printf "%3\$d %d %d\n", 12, 34, 56;  # will print "56 12 34\n"
  printf "%2\$*3\$d %d\n", 12, 34, 3;  # will print " 34 12\n"
```

If `use locale` (including `use locale 'not_characters'`) is in effect and POSIX::setlocale() has been called, the character used for the decimal separator in formatted floating-point numbers is affected by the `LC_NUMERIC` locale. See Section 38.1 [perllocale NAME], page 701 and `POSIX`.

sqrt EXPR
sqrt

Return the positive square root of EXPR. If EXPR is omitted, uses `$_`. Works only for non-negative operands unless you've loaded the `Math::Complex` module.

```
use Math::Complex;
print sqrt(-4);    # prints 2i
```

srand EXPR
srand

Sets and returns the random number seed for the `rand` operator.

The point of the function is to "seed" the `rand` function so that `rand` can produce a different sequence each time you run your program. When called with a parameter, `srand` uses that for the seed; otherwise it (semi-)randomly chooses a seed. In either case, starting with Perl 5.14, it returns the seed. To signal that your code will work *only* on Perls of a recent vintage:

```
use 5.014;  # so srand returns the seed
```

If `srand()` is not called explicitly, it is called implicitly without a parameter at the first use of the `rand` operator. However, there are a few situations where programs are likely to want to call `srand`. One is for generating predictable results, generally for testing or debugging. There, you use `srand($seed)`, with the same `$seed` each time. Another case is that you may want to call `srand()` after a `fork()` to avoid child processes sharing the same seed value as the parent (and consequently each other).

Do **not** call `srand()` (i.e., without an argument) more than once per process. The internal state of the random number generator should contain more entropy than can be provided by any seed, so calling `srand()` again actually *loses* randomness.

Most implementations of `srand` take an integer and will silently truncate decimal numbers. This means `srand(42)` will usually produce the same results as `srand(42.1)`. To be safe, always pass `srand` an integer.

A typical use of the returned seed is for a test program which has too many combinations to test comprehensively in the time available to it each run. It can test a random subset each time, and should there be a failure, log the seed used for that run so that it can later be used to reproduce the same results.

`rand()` is not cryptographically secure. You should not rely on it in security-sensitive situations. As of this writing, a number of third-party CPAN modules offer random number generators intended by their authors to be cryptographically secure, including: `Data-Entropy`, `Crypt-Random`, `Math-Random-Secure`, and `Math-TrulyRandom`.

stat FILEHANDLE
stat EXPR
stat DIRHANDLE
stat

Returns a 13-element list giving the status info for a file, either the file opened via FILEHANDLE or DIRHANDLE, or named by EXPR. If EXPR is omitted, it stats `$_` (not _!). Returns the empty list if `stat` fails. Typically used as follows:

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
   $atime,$mtime,$ctime,$blksize,$blocks)
       = stat($filename);
```

Not all fields are supported on all filesystem types. Here are the meanings of the fields:

```
  0 dev      device number of filesystem
  1 ino      inode number
  2 mode     file mode  (type and permissions)
  3 nlink    number of (hard) links to the file
  4 uid      numeric user ID of file's owner
  5 gid      numeric group ID of file's owner
  6 rdev     the device identifier (special files only)
  7 size     total size of file, in bytes
  8 atime    last access time in seconds since the epoch
  9 mtime    last modify time in seconds since the epoch
 10 ctime    inode change time in seconds since the epoch (*)
 11 blksize  preferred I/O size in bytes for interacting with the
             file (may vary from file to file)
 12 blocks   actual number of system-specific blocks allocated
             on disk (often, but not always, 512 bytes each)
```

(The epoch was at 00:00 January 1, 1970 GMT.)

(*) Not all fields are supported on all filesystem types. Notably, the ctime field is non-portable. In particular, you cannot expect it to be a "creation time"; see Section 56.3.3 [perlport Files and Filesystems], page 955 for details.

If `stat` is passed the special filehandle consisting of an underline, no stat is done, but the current contents of the stat structure from the last `stat`, `lstat`, or filetest are returned. Example:

```
if (-x $file && (($d) = stat(_)) && $d < 0) {
    print "$file is executable NFS file\n";
}
```

(This works on machines only for which the device number is negative under NFS.)

Because the mode contains both the file type and its permissions, you should mask off the file type portion and (s)printf using a `"%o"` if you want to see the real permissions.

```
$mode = (stat($filename))[2];
printf "Permissions are %04o\n", $mode & 07777;
```

In scalar context, stat returns a boolean value indicating success or failure, and, if successful, sets the information associated with the special filehandle _.

The File-stat module provides a convenient, by-name access mechanism:

```
use File::stat;
$sb = stat($filename);
printf "File is %s, size is %s, perm %04o, mtime %s\n",
       $filename, $sb->size, $sb->mode & 07777,
       scalar localtime $sb->mtime;
```

You can import symbolic mode constants (S_IF*) and functions (S_IS*) from the Fcntl module:

```
use Fcntl ':mode';

$mode = (stat($filename))[2];

$user_rwx      = ($mode & S_IRWXU) >> 6;
$group_read    = ($mode & S_IRGRP) >> 3;
$other_execute =  $mode & S_IXOTH;

printf "Permissions are %04o\n", S_IMODE($mode), "\n";

$is_setuid     =  $mode & S_ISUID;
$is_directory  =  S_ISDIR($mode);
```

You could write the last two using the -u and -d operators. Commonly available S_IF* constants are:

```
# Permissions: read, write, execute, for user, group, others.

S_IRWXU S_IRUSR S_IWUSR S_IXUSR
S_IRWXG S_IRGRP S_IWGRP S_IXGRP
S_IRWXO S_IROTH S_IWOTH S_IXOTH

# Setuid/Setgid/Stickiness/SaveText.
# Note that the exact meaning of these is system-dependent.

S_ISUID S_ISGID S_ISVTX S_ISTXT

# File types.  Not all are necessarily available on
# your system.

S_IFREG S_IFDIR S_IFLNK S_IFBLK S_IFCHR
S_IFIFO S_IFSOCK S_IFWHT S_ENFMT

# The following are compatibility aliases for S_IRUSR,
# S_IWUSR, and S_IXUSR.

S_IREAD S_IWRITE S_IEXEC
```

and the S_IF* functions are

```
S_IMODE($mode)    the part of $mode containing the permission
                  bits and the setuid/setgid/sticky bits

S_IFMT($mode)     the part of $mode containing the file type
                  which can be bit-anded with (for example)
                  S_IFREG or with the following functions

# The operators -f, -d, -l, -b, -c, -p, and -S.

S_ISREG($mode) S_ISDIR($mode) S_ISLNK($mode)
S_ISBLK($mode) S_ISCHR($mode) S_ISFIFO($mode) S_ISSOCK($mode)

# No direct -X operator counterpart, but for the first one
# the -g operator is often equivalent.  The ENFMT stands for
# record flocking enforcement, a platform-dependent feature.

S_ISENFMT($mode) S_ISWHT($mode)
```

See your native chmod(2) and stat(2) documentation for more details about the S_* constants. To get status info for a symbolic link instead of the target file behind the link, use the lstat function.

Portability issues: [perlport stat], page 978.

state VARLIST
state TYPE VARLIST
state VARLIST : ATTRS
state TYPE VARLIST : ATTRS

state declares a lexically scoped variable, just like my. However, those variables will never be reinitialized, contrary to lexical variables that are reinitialized each time their enclosing block is entered. See Section 73.3.3 [perlsub Persistent Private Variables], page 1226 for details.

If more than one variable is listed, the list must be placed in parentheses. With a parenthesised list, undef can be used as a dummy placeholder. However, since initialization of state variables in list context is currently not possible this would serve no purpose.

state variables are enabled only when the use feature "state" pragma is in effect, unless the keyword is written as CORE::state. See also feature. Alternately, include a use v5.10 or later to the current scope.

study SCALAR
study

May take extra time to study SCALAR ($_ if unspecified) in anticipation of doing many pattern matches on the string before it is next modified. This may or may not save time, depending on the nature and number of patterns you are searching and the distribution of character frequencies in the string to be searched; you probably want to compare run times with and without it to see which is faster. Those loops that scan for many short constant strings (including the constant parts of more complex patterns) will benefit most.

Note that since Perl version 5.16 this function has been a no-op, but this might change in a future release.

(The way study works is this: a linked list of every character in the string to be searched is made, so we know, for example, where all the 'k' characters are. From each search string, the rarest character is selected, based on some static frequency tables constructed from some C programs and English text. Only those places that contain this "rarest" character are examined.)

For example, here is a loop that inserts index producing entries before any line containing a certain pattern:

```
while (<>) {
    study;
    print ".IX foo\n"    if /\bfoo\b/;
    print ".IX bar\n"    if /\bbar\b/;
    print ".IX blurfl\n" if /\bblurfl\b/;
    # ...
    print;
}
```

In searching for /\bfoo\b/, only locations in $_ that contain f will be looked at, because f is rarer than o. In general, this is a big win except in pathological cases. The only question is whether it saves you more time than it took to build the linked list in the first place.

Note that if you have to look for strings that you don't know till runtime, you can build an entire loop as a string and eval that to avoid recompiling all your patterns all the time. Together with undefining $/ to input entire files as one record, this can be quite fast, often faster than specialized programs like fgrep(1). The following scans a list of files (@files) for a list of words (@words), and prints out the names of those files that contain a match:

```
$search = 'while (<>) { study;';
foreach $word (@words) {
    $search .= "++\$seen{\$ARGV} if /\\b$word\\b/;\n";
}
$search .= "}";
@ARGV = @files;
undef $/;
eval $search;        # this screams
$/ = "\n";        # put back to normal input delimiter
foreach $file (sort keys(%seen)) {
    print $file, "\n";
}
```

sub NAME BLOCK
sub NAME (PROTO) BLOCK
sub NAME : ATTRS BLOCK
sub NAME (PROTO) : ATTRS BLOCK

This is subroutine definition, not a real function *per se*. Without a BLOCK it's just a forward declaration. Without a NAME, it's an anonymous function declaration, so does return a value: the CODE ref of the closure just created.

See Section 73.1 [perlsub NAME], page 1216 and Section 62.1 [perlref NAME], page 1077 for details about subroutines and references; see `attributes` and `Attribute-Handlers` for more information about attributes.

__SUB__

A special token that returns a reference to the current subroutine, or `undef` outside of a subroutine.

The behaviour of `__SUB__` within a regex code block (such as `/(?{...})/`) is subject to change.

This token is only available under `use v5.16` or the "current_sub" feature. See `feature`.

substr EXPR,OFFSET,LENGTH,REPLACEMENT
substr EXPR,OFFSET,LENGTH
substr EXPR,OFFSET

Extracts a substring out of EXPR and returns it. First character is at offset zero. If OFFSET is negative, starts that far back from the end of the string. If LENGTH is omitted, returns everything through the end of the string. If LENGTH is negative, leaves that many characters off the end of the string.

```
    my $s = "The black cat climbed the green tree";
    my $color  = substr $s, 4, 5;      # black
    my $middle = substr $s, 4, -11;    # black cat climbed the
    my $end    = substr $s, 14;        # climbed the green tree
    my $tail   = substr $s, -4;        # tree
    my $z      = substr $s, -4, 2;     # tr
```

You can use the substr() function as an lvalue, in which case EXPR must itself be an lvalue. If you assign something shorter than LENGTH, the string will shrink, and if you assign something longer than LENGTH, the string will grow to accommodate it. To keep the string the same length, you may need to pad or chop your value using `sprintf`.

If OFFSET and LENGTH specify a substring that is partly outside the string, only the part within the string is returned. If the substring is beyond either end of the string, substr() returns the undefined value and produces a warning. When used as an lvalue, specifying a substring that is entirely outside the string raises an exception. Here's an example showing the behavior for boundary cases:

```
    my $name = 'fred';
    substr($name, 4) = 'dy';         # $name is now 'freddy'
    my $null = substr $name, 6, 2;   # returns "" (no warning)
    my $oops = substr $name, 7;      # returns undef, with warning
    substr($name, 7) = 'gap';        # raises an exception
```

An alternative to using substr() as an lvalue is to specify the replacement string as the 4th argument. This allows you to replace parts of the EXPR and return what was there before in one operation, just as you can with splice().

```
    my $s = "The black cat climbed the green tree";
    my $z = substr $s, 14, 7, "jumped from";    # climbed
    # $s is now "The black cat jumped from the green tree"
```

Note that the lvalue returned by the three-argument version of substr() acts as a 'magic bullet'; each time it is assigned to, it remembers which part of the original string is being modified; for example:

```
    $x = '1234';
    for (substr($x,1,2)) {
        $_ = 'a';   print $x,"\n";    # prints 1a4
        $_ = 'xyz'; print $x,"\n";    # prints 1xyz4
        $x = '56789';
        $_ = 'pq';  print $x,"\n";    # prints 5pq9
    }
```

With negative offsets, it remembers its position from the end of the string when the target string is modified:

```
    $x = '1234';
    for (substr($x, -3, 2)) {
        $_ = 'a';   print $x,"\n";    # prints 1a4, as above
        $x = 'abcdefg';
        print $_,"\n";                # prints f
    }
```

Prior to Perl version 5.10, the result of using an lvalue multiple times was unspecified. Prior to 5.16, the result with negative offsets was unspecified.

symlink OLDFILE,NEWFILE

Creates a new filename symbolically linked to the old filename. Returns 1 for success, 0 otherwise. On systems that don't support symbolic links, raises an exception. To check for that, use eval:

```
    $symlink_exists = eval { symlink("",""); 1 };
```

Portability issues: [perlport symlink], page 979.

syscall NUMBER, LIST

Calls the system call specified as the first element of the list, passing the remaining elements as arguments to the system call. If unimplemented, raises an exception. The arguments are interpreted as follows: if a given argument is numeric, the argument is passed as an int. If not, the pointer to the string value is passed. You are responsible to make sure a string is pre-extended long enough to receive any result that might be written into a string. You can't use a string literal (or other read-only string) as an argument to `syscall` because Perl has to assume that any string pointer might be written through. If your integer arguments are not literals and have never been interpreted in a numeric context, you may need to add 0 to them to force them to look like numbers. This emulates the `syswrite` function (or vice versa):

```
require 'syscall.ph';        # may need to run h2ph
$s = "hi there\n";
syscall(&SYS_write, fileno(STDOUT), $s, length $s);
```

Note that Perl supports passing of up to only 14 arguments to your syscall, which in practice should (usually) suffice.

Syscall returns whatever value returned by the system call it calls. If the system call fails, `syscall` returns `-1` and sets `$!` (errno). Note that some system calls *can* legitimately return `-1`. The proper way to handle such calls is to assign `$!=0` before the call, then check the value of `$!` if `syscall` returns `-1`.

There's a problem with `syscall(&SYS_pipe)`: it returns the file number of the read end of the pipe it creates, but there is no way to retrieve the file number of the other end. You can avoid this problem by using `pipe` instead.

Portability issues: [perlport syscall], page 979.

sysopen FILEHANDLE,FILENAME,MODE
sysopen FILEHANDLE,FILENAME,MODE,PERMS

Opens the file whose filename is given by FILENAME, and associates it with FILEHANDLE. If FILEHANDLE is an expression, its value is used as the real filehandle wanted; an undefined scalar will be suitably autovivified. This function calls the underlying operating system's *open*(2) function with the parameters FILENAME, MODE, and PERMS.

The possible values and flag bits of the MODE parameter are system-dependent; they are available via the standard module `Fcntl`. See the documentation of your operating system's *open*(2) syscall to see which values and flag bits are available. You may combine several flags using the `|`-operator.

Some of the most common values are `O_RDONLY` for opening the file in read-only mode, `O_WRONLY` for opening the file in write-only mode, and `O_RDWR` for opening the file in read-write mode.

For historical reasons, some values work on almost every system supported by Perl: 0 means read-only, 1 means write-only, and 2 means read/write. We know that these values do *not* work under OS/390 and on the Macintosh; you probably don't want to use them in new code.

If the file named by FILENAME does not exist and the `open` call creates it (typically because MODE includes the `O_CREAT` flag), then the value of PERMS specifies the permissions of the newly created file. If you omit the PERMS argument to `sysopen`, Perl uses the octal value `0666`. These permission values need to be in octal, and are modified by your process's current `umask`.

In many systems the `O_EXCL` flag is available for opening files in exclusive mode. This is **not** locking: exclusiveness means here that if the file already exists, sysopen() fails. `O_EXCL` may not work on network filesystems, and has no effect unless the `O_CREAT` flag is set as well. Setting `O_CREAT|O_EXCL` prevents the file from being opened if it is a symbolic link. It does not protect against symbolic links in the file's path.

Sometimes you may want to truncate an already-existing file. This can be done using the `O_TRUNC` flag. The behavior of `O_TRUNC` with `O_RDONLY` is undefined.

You should seldom if ever use `0644` as argument to `sysopen`, because that takes away the user's option to have a more permissive umask. Better to omit it. See the perlfunc(1) entry on `umask` for more on this.

Note that `sysopen` depends on the fdopen() C library function. On many Unix systems, fdopen() is known to fail when file descriptors exceed a certain value, typically 255. If you need more file descriptors than that, consider using the POSIX::open() function.

See Section 49.1 [perlopentut NAME], page 852 for a kinder, gentler explanation of opening files.

Portability issues: [perlport sysopen], page 979.

sysread FILEHANDLE,SCALAR,LENGTH,OFFSET
sysread FILEHANDLE,SCALAR,LENGTH

Attempts to read LENGTH bytes of data into variable SCALAR from the specified FILEHANDLE, using the read(2). It bypasses buffered IO, so mixing this with other kinds of reads, `print`, `write`, `seek`, `tell`, or `eof` can cause confusion because the perlio or stdio layers usually buffers data. Returns the number of bytes actually read, `0` at end of file, or undef if there was an error (in the latter case `$!` is also set). SCALAR will be grown or shrunk so that the last byte actually read is the last byte of the scalar after the read.

An OFFSET may be specified to place the read data at some place in the string other than the beginning. A negative OFFSET specifies placement at that many characters counting backwards from the end of the string. A positive OFFSET greater than the length of SCALAR results in the string being padded to the required size with `"\0"` bytes before the result of the read is appended.

There is no syseof() function, which is ok, since eof() doesn't work well on device files (like ttys) anyway. Use sysread() and check for a return value for 0 to decide whether you're done.

Note that if the filehandle has been marked as `:utf8` Unicode characters are read instead of bytes (the LENGTH, OFFSET, and the return value of sysread() are in Unicode characters). The `:encoding(...)` layer implicitly introduces the `:utf8` layer. See ⟨undefined⟩ [binmode], page ⟨undefined⟩, ⟨undefined⟩ [open], page ⟨undefined⟩, and the `open` pragma, `open`.

sysseek FILEHANDLE,POSITION,WHENCE

Sets FILEHANDLE's system position in bytes using lseek(2). FILEHANDLE may be an expression whose value gives the name of the filehandle. The values for WHENCE are `0` to set the new position to POSITION; `1` to set the it to the current position plus POSITION; and `2` to set it to EOF plus POSITION, typically negative.

Note the *in bytes*: even if the filehandle has been set to operate on characters (for example by using the `:encoding(utf8)` I/O layer), tell() will return byte offsets, not character offsets (because implementing that would render sysseek() unacceptably slow).

sysseek() bypasses normal buffered IO, so mixing it with reads other than `sysread` (for example `<>` or read()) `print`, `write`, `seek`, `tell`, or `eof` may cause confusion.

For WHENCE, you may also use the constants SEEK_SET, SEEK_CUR, and SEEK_END (start of the file, current position, end of the file) from the Fcntl module. Use of the constants is also more portable than relying on 0, 1, and 2. For example to define a "systell" function:

```
use Fcntl 'SEEK_CUR';
sub systell { sysseek($_[0], 0, SEEK_CUR) }
```

Returns the new position, or the undefined value on failure. A position of zero is returned as the string "0 but true"; thus sysseek returns true on success and false on failure, yet you can still easily determine the new position.

system LIST
system PROGRAM LIST

Does exactly the same thing as exec LIST, except that a fork is done first and the parent process waits for the child process to exit. Note that argument processing varies depending on the number of arguments. If there is more than one argument in LIST, or if LIST is an array with more than one value, starts the program given by the first element of the list with arguments given by the rest of the list. If there is only one scalar argument, the argument is checked for shell metacharacters, and if there are any, the entire argument is passed to the system's command shell for parsing (this is /bin/sh -c on Unix platforms, but varies on other platforms). If there are no shell metacharacters in the argument, it is split into words and passed directly to execvp, which is more efficient. On Windows, only the system PROGRAM LIST syntax will reliably avoid using the shell; system LIST, even with more than one element, will fall back to the shell if the first spawn fails.

Perl will attempt to flush all files opened for output before any operation that may do a fork, but this may not be supported on some platforms (see Section 56.1 [perlport NAME], page 951). To be safe, you may need to set $| ($AUTOFLUSH in English) or call the autoflush() method of IO::Handle on any open handles.

The return value is the exit status of the program as returned by the wait call. To get the actual exit value, shift right by eight (see below). See also ⟨undefined⟩ [exec], page ⟨undefined⟩. This is *not* what you want to use to capture the output from a command; for that you should use merely backticks or qx//, as described in [perlop 'STRING'], page 832. Return value of -1 indicates a failure to start the program or an error of the wait(2) system call (inspect $! for the reason).

If you'd like to make system (and many other bits of Perl) die on error, have a look at the autodie pragma.

Like exec, system allows you to lie to a program about its name if you use the system PROGRAM LIST syntax. Again, see ⟨undefined⟩ [exec], page ⟨undefined⟩.

Since SIGINT and SIGQUIT are ignored during the execution of system, if you expect your program to terminate on receipt of these signals you will need to arrange to do so yourself based on the return value.

```
@args = ("command", "arg1", "arg2");
system(@args) == 0
```

```
        or die "system @args failed: $?"
```

If you'd like to manually inspect `system`'s failure, you can check all possible failure modes by inspecting `$?` like this:

```
    if ($? == -1) {
        print "failed to execute: $!\n";
    }
    elsif ($? & 127) {
        printf "child died with signal %d, %s coredump\n",
            ($? & 127),  ($? & 128) ? 'with' : 'without';
    }
    else {
        printf "child exited with value %d\n", $? >> 8;
    }
```

Alternatively, you may inspect the value of `${^CHILD_ERROR_NATIVE}` with the `W*()` calls from the POSIX module.

When `system`'s arguments are executed indirectly by the shell, results and return codes are subject to its quirks. See [perlop 'STRING'], page 832 and ⟨undefined⟩ [exec], page ⟨undefined⟩ for details.

Since `system` does a `fork` and `wait` it may affect a `SIGCHLD` handler. See Section 36.1 [perlipc NAME], page 667 for details.

Portability issues: [perlport system], page 979.

syswrite FILEHANDLE,SCALAR,LENGTH,OFFSET
syswrite FILEHANDLE,SCALAR,LENGTH
syswrite FILEHANDLE,SCALAR

Attempts to write LENGTH bytes of data from variable SCALAR to the specified FILEHANDLE, using write(2). If LENGTH is not specified, writes whole SCALAR. It bypasses buffered IO, so mixing this with reads (other than `sysread()`), `print`, `write`, `seek`, `tell`, or `eof` may cause confusion because the perlio and stdio layers usually buffer data. Returns the number of bytes actually written, or `undef` if there was an error (in this case the errno variable `$!` is also set). If the LENGTH is greater than the data available in the SCALAR after the OFFSET, only as much data as is available will be written.

An OFFSET may be specified to write the data from some part of the string other than the beginning. A negative OFFSET specifies writing that many characters counting backwards from the end of the string. If SCALAR is of length zero, you can only use an OFFSET of 0.

WARNING: If the filehandle is marked `:utf8`, Unicode characters encoded in UTF-8 are written instead of bytes, and the LENGTH, OFFSET, and return value of syswrite() are in (UTF8-encoded Unicode) characters. The `:encoding(...)` layer implicitly introduces the `:utf8` layer. Alternately, if the handle is not marked with an encoding but you attempt to write characters with code points over 255, raises an exception. See ⟨undefined⟩ [binmode], page ⟨undefined⟩, ⟨undefined⟩ [open], page ⟨undefined⟩, and the `open` pragma, `open`.

tell FILEHANDLE
tell

Returns the current position *in bytes* for FILEHANDLE, or -1 on error. FILEHANDLE may be an expression whose value gives the name of the actual filehandle. If FILEHANDLE is omitted, assumes the file last read.

Note the *in bytes*: even if the filehandle has been set to operate on characters (for example by using the `:encoding(utf8)` open layer), tell() will return byte offsets, not character offsets (because that would render seek() and tell() rather slow).

The return value of tell() for the standard streams like the STDIN depends on the operating system: it may return -1 or something else. tell() on pipes, fifos, and sockets usually returns -1.

There is no `systell` function. Use `sysseek(FH, 0, 1)` for that.

Do not use tell() (or other buffered I/O operations) on a filehandle that has been manipulated by sysread(), syswrite(), or sysseek(). Those functions ignore the buffering, while tell() does not.

telldir DIRHANDLE

Returns the current position of the `readdir` routines on DIRHANDLE. Value may be given to `seekdir` to access a particular location in a directory. `telldir` has the same caveats about possible directory compaction as the corresponding system library routine.

tie VARIABLE,CLASSNAME,LIST

This function binds a variable to a package class that will provide the implementation for the variable. VARIABLE is the name of the variable to be enchanted. CLASSNAME is the name of a class implementing objects of correct type. Any additional arguments are passed to the appropriate constructor method of the class (meaning `TIESCALAR`, `TIEHANDLE`, `TIEARRAY`, or `TIEHASH`). Typically these are arguments such as might be passed to the `dbm_open()` function of C. The object returned by the constructor is also returned by the `tie` function, which would be useful if you want to access other methods in CLASSNAME.

Note that functions such as `keys` and `values` may return huge lists when used on large objects, like DBM files. You may prefer to use the `each` function to iterate over such. Example:

```
# print out history file offsets
use NDBM_File;
tie(%HIST, 'NDBM_File', '/usr/lib/news/history', 1, 0);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
untie(%HIST);
```

A class implementing a hash should have the following methods:

```
TIEHASH classname, LIST
FETCH this, key
STORE this, key, value
```

```
DELETE this, key
CLEAR this
EXISTS this, key
FIRSTKEY this
NEXTKEY this, lastkey
SCALAR this
DESTROY this
UNTIE this
```

A class implementing an ordinary array should have the following methods:

```
TIEARRAY classname, LIST
FETCH this, key
STORE this, key, value
FETCHSIZE this
STORESIZE this, count
CLEAR this
PUSH this, LIST
POP this
SHIFT this
UNSHIFT this, LIST
SPLICE this, offset, length, LIST
EXTEND this, count
DELETE this, key
EXISTS this, key
DESTROY this
UNTIE this
```

A class implementing a filehandle should have the following methods:

```
TIEHANDLE classname, LIST
READ this, scalar, length, offset
READLINE this
GETC this
WRITE this, scalar, length, offset
PRINT this, LIST
PRINTF this, format, LIST
BINMODE this
EOF this
FILENO this
SEEK this, position, whence
TELL this
OPEN this, mode, LIST
CLOSE this
DESTROY this
UNTIE this
```

A class implementing a scalar should have the following methods:

```
TIESCALAR classname, LIST
FETCH this,
STORE this, value
```

```
DESTROY this
UNTIE this
```

Not all methods indicated above need be implemented. See Section 76.1 [perltie NAME], page 1289, `Tie-Hash`, `Tie-Array`, `Tie-Scalar`, and `Tie-Handle`.

Unlike `dbmopen`, the `tie` function will not `use` or `require` a module for you; you need to do that explicitly yourself. See `DB_File` or the `Config` module for interesting `tie` implementations.

For further details see Section 76.1 [perltie NAME], page 1289, [tied VARIABLE], page 475.

tied VARIABLE

Returns a reference to the object underlying VARIABLE (the same value that was originally returned by the `tie` call that bound the variable to a package.) Returns the undefined value if VARIABLE isn't tied to a package.

time

Returns the number of non-leap seconds since whatever time the system considers to be the epoch, suitable for feeding to `gmtime` and `localtime`. On most systems the epoch is 00:00:00 UTC, January 1, 1970; a prominent exception being Mac OS Classic which uses 00:00:00, January 1, 1904 in the current local time zone for its epoch.

For measuring time in better granularity than one second, use the `Time-HiRes` module from Perl 5.8 onwards (or from CPAN before then), or, if you have gettimeofday(2), you may be able to use the `syscall` interface of Perl. See `perlfaq8` for details.

For date and time processing look at the many related modules on CPAN. For a comprehensive date and time representation look at the `DateTime` module.

times

Returns a four-element list giving the user and system times in seconds for this process and any exited children of this process.

```
($user,$system,$cuser,$csystem) = times;
```

In scalar context, `times` returns `$user`.

Children's times are only included for terminated children.

Portability issues: [perlport times], page 980.

tr///

The transliteration operator. Same as `y///`. See Section 48.2.31 [perlop Quote-Like Operators], page 832.

truncate FILEHANDLE,LENGTH
truncate EXPR,LENGTH

Truncates the file opened on FILEHANDLE, or named by EXPR, to the specified length. Raises an exception if truncate isn't implemented on your system. Returns true if successful, `undef` on error.

The behavior is undefined if LENGTH is greater than the length of the file.

The position in the file of FILEHANDLE is left unchanged. You may want to call [seek], page 442 before writing to the file.

Portability issues: [perlport truncate], page 980.

uc EXPR

uc

Returns an uppercased version of EXPR. This is the internal function implementing the `\U` escape in double-quoted strings. It does not attempt to do titlecase mapping on initial letters. See [ucfirst], page 476 for that.

If EXPR is omitted, uses `$_`.

This function behaves the same way under various pragma, such as in a locale, as [lc], page 399 does.

ucfirst EXPR
ucfirst

Returns the value of EXPR with the first character in uppercase (titlecase in Unicode). This is the internal function implementing the `\u` escape in double-quoted strings.

If EXPR is omitted, uses `$_`.

This function behaves the same way under various pragma, such as in a locale, as [lc], page 399 does.

umask EXPR
umask

Sets the umask for the process to EXPR and returns the previous value. If EXPR is omitted, merely returns the current umask.

The Unix permission `rwxr-x---` is represented as three sets of three bits, or three octal digits: `0750` (the leading 0 indicates octal and isn't one of the digits). The `umask` value is such a number representing disabled permissions bits. The permission (or "mode") values you pass `mkdir` or `sysopen` are modified by your umask, so even if you tell `sysopen` to create a file with permissions `0777`, if your umask is `0022`, then the file will actually be created with permissions `0755`. If your `umask` were `0027` (group can't write; others can't read, write, or execute), then passing `sysopen` `0666` would create a file with mode `0640` (because `0666 &~ 027` is `0640`).

Here's some advice: supply a creation mode of `0666` for regular files (in `sysopen`) and one of `0777` for directories (in `mkdir`) and executable files. This gives users the freedom of choice: if they want protected files, they might choose process umasks of `022`, `027`, or even the particularly antisocial mask of `077`. Programs should rarely if ever make policy decisions better left to the user. The exception to this is when writing files that should be kept private: mail files, web browser cookies, *.rhosts* files, and so on.

If umask(2) is not implemented on your system and you are trying to restrict access for *yourself* (i.e., `(EXPR & 0700) > 0`), raises an exception. If umask(2) is not implemented and you are not trying to restrict access for yourself, returns `undef`.

Remember that a umask is a number, usually given in octal; it is *not* a string of octal digits. See also [oct], page 406, if all you have is a string.

Portability issues: [perlport umask], page 980.

undef EXPR
undef

Undefines the value of EXPR, which must be an lvalue. Use only on a scalar value, an array (using `@`), a hash (using `%`), a subroutine (using `&`), or a typeglob (using `*`). Saying `undef $hash{$key}` will probably not do what you expect on most predefined variables or DBM list values, so don't do that; see ⟨undefined⟩ [delete], page ⟨undefined⟩. Always returns the undefined value. You can omit the EXPR, in which case nothing is undefined, but you still get an undefined value that you could, for instance, return from a subroutine, assign to a variable, or pass as a parameter. Examples:

```
undef $foo;
undef $bar{'blurfl'};      # Compare to: delete $bar{'blurfl'};
undef @ary;
undef %hash;
undef &mysub;
undef *xyz;       # destroys $xyz, @xyz, %xyz, &xyz, etc.
return (wantarray ? (undef, $errmsg) : undef) if $they_blew_it;
select undef, undef, undef, 0.25;
($a, $b, undef, $c) = &foo;       # Ignore third value returned
```

Note that this is a unary operator, not a list operator.

unlink LIST
unlink

Deletes a list of files. On success, it returns the number of files it successfully deleted. On failure, it returns false and sets `$!` (errno):

```
my $unlinked = unlink 'a', 'b', 'c';
unlink @goners;
unlink glob "*.bak";
```

On error, `unlink` will not tell you which files it could not remove. If you want to know which files you could not remove, try them one at a time:

```
 foreach my $file ( @goners ) {
     unlink $file or warn "Could not unlink $file: $!";
 }
```

Note: `unlink` will not attempt to delete directories unless you are superuser and the **-U** flag is supplied to Perl. Even if these conditions are met, be warned that unlinking a directory can inflict damage on your filesystem. Finally, using `unlink` on directories is not supported on many operating systems. Use `rmdir` instead.

If LIST is omitted, `unlink` uses `$_`.

unpack TEMPLATE,EXPR
unpack TEMPLATE

unpack does the reverse of pack: it takes a string and expands it out into a list of values. (In scalar context, it returns merely the first value produced.)

If EXPR is omitted, unpacks the $_ string. See Section 50.1 [perlpacktut NAME], page 857 for an introduction to this function.

The string is broken into chunks described by the TEMPLATE. Each chunk is converted separately to a value. Typically, either the string is a result of pack, or the characters of the string represent a C structure of some kind.

The TEMPLATE has the same format as in the pack function. Here's a subroutine that does substring:

```
sub substr {
    my($what,$where,$howmuch) = @_;
    unpack("x$where a$howmuch", $what);
}
```

and then there's

```
sub ordinal { unpack("W",$_[0]); } # same as ord()
```

In addition to fields allowed in pack(), you may prefix a field with a %<number> to indicate that you want a <number>-bit checksum of the items instead of the items themselves. Default is a 16-bit checksum. Checksum is calculated by summing numeric values of expanded values (for string fields the sum of ord($char) is taken; for bit fields the sum of zeroes and ones).

For example, the following computes the same number as the System V sum program:

```
$checksum = do {
    local $/;  # slurp!
    unpack("%32W*",<>) % 65535;
};
```

The following efficiently counts the number of set bits in a bit vector:

```
$setbits = unpack("%32b*", $selectmask);
```

The p and P formats should be used with care. Since Perl has no way of checking whether the value passed to unpack() corresponds to a valid memory location, passing a pointer value that's not known to be valid is likely to have disastrous consequences.

If there are more pack codes or if the repeat count of a field or a group is larger than what the remainder of the input string allows, the result is not well defined: the repeat count may be decreased, or unpack() may produce empty strings or zeros, or it may raise an exception. If the input string is longer than one described by the TEMPLATE, the remainder of that input string is ignored.

See ⟨undefined⟩ [pack], page ⟨undefined⟩ for more examples and notes.

unshift ARRAY,LIST
unshift EXPR,LIST

Does the opposite of a `shift`. Or the opposite of a `push`, depending on how you look at it. Prepends list to the front of the array and returns the new number of elements in the array.

```
unshift(@ARGV, '-e') unless $ARGV[0] =~ /^-/;
```

Note the LIST is prepended whole, not one element at a time, so the prepended elements stay in the same order. Use `reverse` to do the reverse.

Starting with Perl 5.14, `unshift` can take a scalar EXPR, which must hold a reference to an unblessed array. The argument will be dereferenced automatically. This aspect of `unshift` is considered highly experimental. The exact behaviour may change in a future version of Perl.

To avoid confusing would-be users of your code who are running earlier versions of Perl with mysterious syntax errors, put this sort of thing at the top of your file to signal that your code will work *only* on Perls of a recent vintage:

```
use 5.014;  # so push/pop/etc work on scalars (experimental)
```

untie VARIABLE

Breaks the binding between a variable and a package. (See [tie], page 473.) Has no effect if the variable is not tied.

use Module VERSION LIST
use Module VERSION
use Module LIST
use Module
use VERSION

Imports some semantics into the current package from the named module, generally by aliasing certain subroutine or variable names into your package. It is exactly equivalent to

```
BEGIN { require Module; Module->import( LIST ); }
```

except that Module *must* be a bareword. The importation can be made conditional by using the `if` module.

In the peculiar `use VERSION` form, VERSION may be either a positive decimal fraction such as 5.006, which will be compared to `$]`, or a v-string of the form v5.6.1, which will be compared to `$^V` (aka $PERL_VERSION). An exception is raised if VERSION is greater than the version of the current Perl interpreter; Perl will not attempt to parse the rest of the file. Compare with [require], page 437, which can do a similar check at run time. Symmetrically, `no VERSION` allows you to specify that you want a version of Perl older than the specified one.

Specifying VERSION as a literal of the form v5.6.1 should generally be avoided, because it leads to misleading error messages under earlier versions of Perl (that is, prior to 5.6.0) that do not support this syntax. The equivalent numeric version should be used instead.

```
use v5.6.1;     # compile time version check
use 5.6.1;      # ditto
```

```
    use 5.006_001;  # ditto; preferred for backwards compatibility
```

This is often useful if you need to check the current Perl version before `use`ing library modules that won't work with older versions of Perl. (We try not to do this more than we have to.)

`use VERSION` also lexically enables all features available in the requested version as defined by the `feature` pragma, disabling any features not in the requested version's feature bundle. See `feature`. Similarly, if the specified Perl version is greater than or equal to 5.12.0, strictures are enabled lexically as with `use strict`. Any explicit use of `use strict` or `no strict` overrides `use VERSION`, even if it comes before it. Later use of `use VERSION` will override all behavior of a previous `use VERSION`, possibly removing the `strict` and `feature` added by `use VERSION`. `use VERSION` does not load the `feature.pm` or `strict.pm` files.

The `BEGIN` forces the `require` and `import` to happen at compile time. The `require` makes sure the module is loaded into memory if it hasn't been yet. The `import` is not a builtin; it's just an ordinary static method call into the `Module` package to tell the module to import the list of features back into the current package. The module can implement its `import` method any way it likes, though most modules just choose to derive their `import` method via inheritance from the `Exporter` class that is defined in the `Exporter` module. See `Exporter`. If no `import` method can be found then the call is skipped, even if there is an AUTOLOAD method.

If you do not want to call the package's `import` method (for instance, to stop your namespace from being altered), explicitly supply the empty list:

```
    use Module ();
```

That is exactly equivalent to

```
    BEGIN { require Module }
```

If the VERSION argument is present between Module and LIST, then the `use` will call the VERSION method in class Module with the given version as an argument. The default VERSION method, inherited from the UNIVERSAL class, croaks if the given version is larger than the value of the variable `$Module::VERSION`.

Again, there is a distinction between omitting LIST (`import` called with no arguments) and an explicit empty LIST `()` (`import` not called). Note that there is no comma after VERSION!

Because this is a wide-open interface, pragmas (compiler directives) are also implemented this way. Currently implemented pragmas are:

```
    use constant;
    use diagnostics;
    use integer;
    use sigtrap  qw(SEGV BUS);
    use strict   qw(subs vars refs);
    use subs     qw(afunc blurfl);
    use warnings qw(all);
    use sort     qw(stable _quicksort _mergesort);
```

Some of these pseudo-modules import semantics into the current block scope (like `strict` or `integer`, unlike ordinary modules, which import symbols into the current package (which are effective through the end of the file).

Because `use` takes effect at compile time, it doesn't respect the ordinary flow control of the code being compiled. In particular, putting a `use` inside the false branch of a conditional doesn't prevent it from being processed. If a module or pragma only needs to be loaded conditionally, this can be done using the `if` pragma:

```
use if $] < 5.008, "utf8";
use if WANT_WARNINGS, warnings => qw(all);
```

There's a corresponding `no` declaration that unimports meanings imported by `use`, i.e., it calls `unimport Module LIST` instead of `import`. It behaves just as `import` does with VERSION, an omitted or empty LIST, or no unimport method being found.

```
no integer;
no strict 'refs';
no warnings;
```

Care should be taken when using the `no VERSION` form of `no`. It is *only* meant to be used to assert that the running Perl is of a earlier version than its argument and *not* to undo the feature-enabling side effects of `use VERSION`.

See `perlmodlib` for a list of standard modules and pragmas. See Section 69.1 [perlrun NAME], page 1176 for the `-M` and `-m` command-line options to Perl that give `use` functionality from the command-line.

utime LIST

Changes the access and modification times on each file of a list of files. The first two elements of the list must be the NUMERIC access and modification times, in that order. Returns the number of files successfully changed. The inode change time of each file is set to the current time. For example, this code has the same effect as the Unix touch(1) command when the files *already exist* and belong to the user running the program:

```
#!/usr/bin/perl
$atime = $mtime = time;
utime $atime, $mtime, @ARGV;
```

Since Perl 5.8.0, if the first two elements of the list are `undef`, the utime(2) syscall from your C library is called with a null second argument. On most systems, this will set the file's access and modification times to the current time (i.e., equivalent to the example above) and will work even on files you don't own provided you have write permission:

```
for $file (@ARGV) {
    utime(undef, undef, $file)
        || warn "couldn't touch $file: $!";
}
```

Under NFS this will use the time of the NFS server, not the time of the local machine. If there is a time synchronization problem, the NFS server and local machine will have different times. The Unix touch(1) command will in fact normally use this form instead of the one shown in the first example.

Passing only one of the first two elements as `undef` is equivalent to passing a 0 and will not have the effect described when both are `undef`. This also triggers an uninitialized warning.

On systems that support futimes(2), you may pass filehandles among the files. On systems that don't support futimes(2), passing filehandles raises an exception. Filehandles must be passed as globs or glob references to be recognized; barewords are considered filenames.

Portability issues: [perlport utime], page 980.

values HASH
values ARRAY
values EXPR

In list context, returns a list consisting of all the values of the named hash. In Perl 5.12 or later only, will also return a list of the values of an array; prior to that release, attempting to use an array argument will produce a syntax error. In scalar context, returns the number of values.

Hash entries are returned in an apparently random order. The actual random order is specific to a given hash; the exact same series of operations on two hashes may result in a different order for each hash. Any insertion into the hash may change the order, as will any deletion, with the exception that the most recent key returned by `each` or `keys` may be deleted without changing the order. So long as a given hash is unmodified you may rely on `keys`, `values` and `each` to repeatedly return the same order as each other. See Section 70.4.9 [perlsec Algorithmic Complexity Attacks], page 1205 for details on why hash order is randomized. Aside from the guarantees provided here the exact details of Perl's hash algorithm and the hash traversal order are subject to change in any release of Perl. Tied hashes may behave differently to Perl's hashes with respect to changes in order on insertion and deletion of items.

As a side effect, calling values() resets the HASH or ARRAY's internal iterator, see ⟨undefined⟩ [each], page ⟨undefined⟩. (In particular, calling values() in void context resets the iterator with no other overhead. Apart from resetting the iterator, `values @array` in list context is the same as plain `@array`. (We recommend that you use void context `keys @array` for this, but reasoned that taking `values @array` out would require more documentation than leaving it in.)

Note that the values are not copied, which means modifying them will modify the contents of the hash:

```
    for (values %hash)       { s/foo/bar/g }  # modifies %hash values
    for (@hash{keys %hash}) { s/foo/bar/g }  # same
```

Starting with Perl 5.14, `values` can take a scalar EXPR, which must hold a reference to an unblessed hash or array. The argument will be dereferenced automatically. This aspect of `values` is considered highly experimental. The exact behaviour may change in a future version of Perl.

```
    for (values $hashref) { ... }
    for (values $obj->get_arrayref) { ... }
```

To avoid confusing would-be users of your code who are running earlier versions of Perl with mysterious syntax errors, put this sort of thing at the top of your file to signal that your code will work *only* on Perls of a recent vintage:

```
    use 5.012;  # so keys/values/each work on arrays
    use 5.014;  # so keys/values/each work on scalars (experimental)
```

See also `keys`, `each`, and `sort`.

vec EXPR,OFFSET,BITS

Treats the string in EXPR as a bit vector made up of elements of width BITS and returns the value of the element specified by OFFSET as an unsigned integer. BITS therefore specifies the number of bits that are reserved for each element in the bit vector. This must be a power of two from 1 to 32 (or 64, if your platform supports that).

If BITS is 8, "elements" coincide with bytes of the input string.

If BITS is 16 or more, bytes of the input string are grouped into chunks of size BITS/8, and each group is converted to a number as with pack()/unpack() with big-endian formats `n`/`N` (and analogously for BITS==64). See ⟨undefined⟩ [pack], page ⟨undefined⟩ for details.

If bits is 4 or less, the string is broken into bytes, then the bits of each byte are broken into 8/BITS groups. Bits of a byte are numbered in a little-endian-ish way, as in `0x01`, `0x02`, `0x04`, `0x08`, `0x10`, `0x20`, `0x40`, `0x80`. For example, breaking the single input byte `chr(0x36)` into two groups gives a list `(0x6, 0x3)`; breaking it into 4 groups gives `(0x2, 0x1, 0x3, 0x0)`.

`vec` may also be assigned to, in which case parentheses are needed to give the expression the correct precedence as in

```
    vec($image, $max_x * $x + $y, 8) = 3;
```

If the selected element is outside the string, the value 0 is returned. If an element off the end of the string is written to, Perl will first extend the string with sufficiently many zero bytes. It is an error to try to write off the beginning of the string (i.e., negative OFFSET).

If the string happens to be encoded as UTF-8 internally (and thus has the UTF8 flag set), this is ignored by `vec`, and it operates on the internal byte string, not the conceptual character string, even if you only have characters with values less than 256.

Strings created with `vec` can also be manipulated with the logical operators `|`, `&`, `^`, and `~`. These operators will assume a bit vector operation is desired when both operands are strings. See Section 48.2.36 [perlop Bitwise String Operators], page 848.

The following code will build up an ASCII string saying `'PerlPerlPerl'`. The comments show the string after each step. Note that this code works in the same way on big-endian or little-endian machines.

```
    my $foo = '';
    vec($foo,  0, 32) = 0x5065726C; # 'Perl'
    # $foo eq "Perl" eq "\x50\x65\x72\x6C", 32 bits
    print vec($foo, 0, 8);  # prints 80 == 0x50 == ord(’P’)

    vec($foo,  2, 16) = 0x5065; # ’PerlPe’
    vec($foo,  3, 16) = 0x726C; # ’PerlPerl’
    vec($foo,  8,  8) = 0x50;   # ’PerlPerlP’
    vec($foo,  9,  8) = 0x65;   # ’PerlPerlPe’
    vec($foo, 20,  4) = 2;      # ’PerlPerlPe’   . "\x02"
    vec($foo, 21,  4) = 7;      # ’PerlPerlPer’
                                   # ’r’ is "\x72"
    vec($foo, 45,  2) = 3;      # ’PerlPerlPer’  . "\x0c"
    vec($foo, 93,  1) = 1;      # ’PerlPerlPer’  . "\x2c"
    vec($foo, 94,  1) = 1;      # ’PerlPerlPerl’
                                   # ’l’ is "\x6c"
```

To transform a bit vector into a string or list of 0’s and 1’s, use these:

```
    $bits = unpack("b*", $vector);
    @bits = split(//, unpack("b*", $vector));
```

If you know the exact length in bits, it can be used in place of the `*`.

Here is an example to illustrate how the bits actually fall in place:

```
  #!/usr/bin/perl -wl

  print <<’EOT’;
                                    0         1         2         3
                     unpack("V",$_) 01234567890123456789012345678901
  ------------------------------------------------------------------
  EOT

  for $w (0..3) {
      $width = 2**$w;
      for ($shift=0; $shift < $width; ++$shift) {
          for ($off=0; $off < 32/$width; ++$off) {
              $str = pack("B*", "0"x32);
              $bits = (1<<$shift);
              vec($str, $off, $width) = $bits;
              $res = unpack("b*",$str);
              $val = unpack("V", $str);
              write;
          }
      }
  }

  format STDOUT =
  vec($_,@#,@#) = @<< == @######### @>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
  $off, $width, $bits, $val, $res
  .
  __END__
```

Regardless of the machine architecture on which it runs, the example above should print the following table:

```
                                  0         1         2         3
                   unpack("V",$_) 01234567890123456789012345678901
------------------------------------------------------------------
vec($_, 0, 1) = 1   ==          1 10000000000000000000000000000000
vec($_, 1, 1) = 1   ==          2 01000000000000000000000000000000
vec($_, 2, 1) = 1   ==          4 00100000000000000000000000000000
vec($_, 3, 1) = 1   ==          8 00010000000000000000000000000000
vec($_, 4, 1) = 1   ==         16 00001000000000000000000000000000
vec($_, 5, 1) = 1   ==         32 00000100000000000000000000000000
vec($_, 6, 1) = 1   ==         64 00000010000000000000000000000000
vec($_, 7, 1) = 1   ==        128 00000001000000000000000000000000
vec($_, 8, 1) = 1   ==        256 00000000100000000000000000000000
vec($_, 9, 1) = 1   ==        512 00000000010000000000000000000000
vec($_,10, 1) = 1   ==       1024 00000000001000000000000000000000
vec($_,11, 1) = 1   ==       2048 00000000000100000000000000000000
vec($_,12, 1) = 1   ==       4096 00000000000010000000000000000000
vec($_,13, 1) = 1   ==       8192 00000000000001000000000000000000
vec($_,14, 1) = 1   ==      16384 00000000000000100000000000000000
vec($_,15, 1) = 1   ==      32768 00000000000000010000000000000000
vec($_,16, 1) = 1   ==      65536 00000000000000001000000000000000
vec($_,17, 1) = 1   ==     131072 00000000000000000100000000000000
vec($_,18, 1) = 1   ==     262144 00000000000000000010000000000000
vec($_,19, 1) = 1   ==     524288 00000000000000000001000000000000
vec($_,20, 1) = 1   ==    1048576 00000000000000000000100000000000
vec($_,21, 1) = 1   ==    2097152 00000000000000000000010000000000
vec($_,22, 1) = 1   ==    4194304 00000000000000000000001000000000
vec($_,23, 1) = 1   ==    8388608 00000000000000000000000100000000
vec($_,24, 1) = 1   ==   16777216 00000000000000000000000010000000
vec($_,25, 1) = 1   ==   33554432 00000000000000000000000001000000
vec($_,26, 1) = 1   ==   67108864 00000000000000000000000000100000
vec($_,27, 1) = 1   ==  134217728 00000000000000000000000000010000
vec($_,28, 1) = 1   ==  268435456 00000000000000000000000000001000
vec($_,29, 1) = 1   ==  536870912 00000000000000000000000000000100
vec($_,30, 1) = 1   == 1073741824 00000000000000000000000000000010
vec($_,31, 1) = 1   == 2147483648 00000000000000000000000000000001
vec($_, 0, 2) = 1   ==          1 10000000000000000000000000000000
vec($_, 1, 2) = 1   ==          4 00100000000000000000000000000000
vec($_, 2, 2) = 1   ==         16 00001000000000000000000000000000
vec($_, 3, 2) = 1   ==         64 00000010000000000000000000000000
vec($_, 4, 2) = 1   ==        256 00000000100000000000000000000000
vec($_, 5, 2) = 1   ==       1024 00000000001000000000000000000000
vec($_, 6, 2) = 1   ==       4096 00000000000010000000000000000000
vec($_, 7, 2) = 1   ==      16384 00000000000000100000000000000000
vec($_, 8, 2) = 1   ==      65536 00000000000000001000000000000000
vec($_, 9, 2) = 1   ==     262144 00000000000000000010000000000000
vec($_,10, 2) = 1   ==    1048576 00000000000000000000100000000000
vec($_,11, 2) = 1   ==    4194304 00000000000000000000001000000000
vec($_,12, 2) = 1   ==   16777216 00000000000000000000000010000000
vec($_,13, 2) = 1   ==   67108864 00000000000000000000000000100000
vec($_,14, 2) = 1   ==  268435456 00000000000000000000000000001000
vec($_,15, 2) = 1   == 1073741824 00000000000000000000000000000010
vec($_, 0, 2) = 2   ==          2 01000000000000000000000000000000
vec($_, 1, 2) = 2   ==          8 00010000000000000000000000000000
vec($_, 2, 2) = 2   ==         32 00000100000000000000000000000000
vec($_, 3, 2) = 2   ==        128 00000001000000000000000000000000
vec($_, 4, 2) = 2   ==        512 00000000010000000000000000000000
vec($_, 5, 2) = 2   ==       2048 00000000000100000000000000000000
vec($_, 6, 2) = 2   ==       8192 00000000000001000000000000000000
vec($_, 7, 2) = 2   ==      32768 00000000000000010000000000000000
vec($_, 8, 2) = 2   ==     131072 00000000000000000100000000000000
vec($_, 9, 2) = 2   ==     524288 00000000000000000001000000000000
vec($_,10, 2) = 2   ==    2097152 00000000000000000000010000000000
vec($_,11, 2) = 2   ==    8388608 00000000000000000000000100000000
vec($_,12, 2) = 2   ==   33554432 00000000000000000000000001000000
vec($_,13, 2) = 2   ==  134217728 00000000000000000000000000010000
vec($_,14, 2) = 2   ==  536870912 00000000000000000000000000000100
vec($_,15, 2) = 2   == 2147483648 00000000000000000000000000000001
vec($_, 0, 4) = 1   ==          1 10000000000000000000000000000000
vec($_, 1, 4) = 1   ==         16 00001000000000000000000000000000
vec($_, 2, 4) = 1   ==        256 00000000100000000000000000000000
vec($_, 3, 4) = 1   ==       4096 00000000000010000000000000000000
vec($_, 4, 4) = 1   ==      65536 00000000000000001000000000000000
vec($_, 5, 4) = 1   ==    1048576 00000000000000000000100000000000
vec($_, 6, 4) = 1   ==   16777216 00000000000000000000000010000000
vec($_, 7, 4) = 1   ==  268435456 00000000000000000000000000001000
vec($_, 0, 4) = 2   ==          2 01000000000000000000000000000000
vec($_, 1, 4) = 2   ==         32 00000100000000000000000000000000
vec($_, 2, 4) = 2   ==        512 00000000010000000000000000000000
vec($_, 3, 4) = 2   ==       8192 00000000000001000000000000000000
vec($_, 4, 4) = 2   ==     131072 00000000000000000100000000000000
vec($_, 5, 4) = 2   ==    2097152 00000000000000000000010000000000
vec($_, 6, 4) = 2   ==   33554432 00000000000000000000000001000000
vec($_, 7, 4) = 2   ==  536870912 00000000000000000000000000000100
vec($_, 0, 4) = 4   ==          4 00100000000000000000000000000000
vec($_, 1, 4) = 4   ==         64 00000010000000000000000000000000
vec($_, 2, 4) = 4   ==       1024 00000000001000000000000000000000
vec($_, 3, 4) = 4   ==      16384 00000000000000100000000000000000
vec($_, 4, 4) = 4   ==     262144 00000000000000000010000000000000
vec($_, 5, 4) = 4   ==    4194304 00000000000000000000001000000000
vec($_, 6, 4) = 4   ==   67108864 00000000000000000000000000100000
vec($_, 7, 4) = 4   == 1073741824 00000000000000000000000000000010
vec($_, 0, 4) = 8   ==          8 00010000000000000000000000000000
vec($_, 1, 4) = 8   ==        128 00000001000000000000000000000000
vec($_, 2, 4) = 8   ==       2048 00000000000100000000000000000000
vec($_, 3, 4) = 8   ==      32768 00000000000000010000000000000000
vec($_, 4, 4) = 8   ==     524288 00000000000000000001000000000000
vec($_, 5, 4) = 8   ==    8388608 00000000000000000000000100000000
vec($_, 6, 4) = 8   ==  134217728 00000000000000000000000000010000
vec($_, 7, 4) = 8   == 2147483648 00000000000000000000000000000001
vec($_, 0, 8) = 1   ==          1 10000000000000000000000000000000
vec($_, 1, 8) = 1   ==        256 00000000100000000000000000000000
vec($_, 2, 8) = 1   ==      65536 00000000000000001000000000000000
vec($_, 3, 8) = 1   ==   16777216 00000000000000000000000010000000
vec($_, 0, 8) = 2   ==          2 01000000000000000000000000000000
vec($_, 1, 8) = 2   ==        512 00000000010000000000000000000000
vec($_, 2, 8) = 2   ==     131072 00000000000000000100000000000000
vec($_, 3, 8) = 2   ==   33554432 00000000000000000000000001000000
vec($_, 0, 8) = 4   ==          4 00100000000000000000000000000000
vec($_, 1, 8) = 4   ==       1024 00000000001000000000000000000000
vec($_, 2, 8) = 4   ==     262144 00000000000000000010000000000000
vec($_, 3, 8) = 4   ==   67108864 00000000000000000000000000100000
vec($_, 0, 8) = 8   ==          8 00010000000000000000000000000000
vec($_, 1, 8) = 8   ==       2048 00000000000100000000000000000000
vec($_, 2, 8) = 8   ==     524288 00000000000000000001000000000000
vec($_, 3, 8) = 8   ==  134217728 00000000000000000000000000010000
vec($_, 0, 8) = 16  ==         16 00001000000000000000000000000000
vec($_, 1, 8) = 16  ==       4096 00000000000010000000000000000000
vec($_, 2, 8) = 16  ==    1048576 00000000000000000000100000000000
vec($_, 3, 8) = 16  ==  268435456 00000000000000000000000000001000
vec($_, 0, 8) = 32  ==         32 00000100000000000000000000000000
vec($_, 1, 8) = 32  ==       8192 00000000000001000000000000000000
vec($_, 2, 8) = 32  ==    2097152 00000000000000000000010000000000
vec($_, 3, 8) = 32  ==  536870912 00000000000000000000000000000100
vec($_, 0, 8) = 64  ==         64 00000010000000000000000000000000
vec($_, 1, 8) = 64  ==      16384 00000000000000100000000000000000
vec($_, 2, 8) = 64  ==    4194304 00000000000000000000001000000000
vec($_, 3, 8) = 64  == 1073741824 00000000000000000000000000000010
vec($_, 0, 8) = 128 ==        128 00000001000000000000000000000000
vec($_, 1, 8) = 128 ==      32768 00000000000000010000000000000000
vec($_, 2, 8) = 128 ==    8388608 00000000000000000000000100000000
vec($_, 3, 8) = 128 == 2147483648 00000000000000000000000000000001
```

wait

Behaves like wait(2) on your system: it waits for a child process to terminate and returns the pid of the deceased process, or `-1` if there are no child processes. The status is returned in `$?` and `${^CHILD_ERROR_NATIVE}`. Note that a return value of `-1` could mean that child processes are being automatically reaped, as described in Section 36.1 [perlipc NAME], page 667.

If you use wait in your handler for $SIG{CHLD}, it may accidentally wait for the child created by qx() or system(). See Section 36.1 [perlipc NAME], page 667 for details.

Portability issues: [perlport wait], page 980.

waitpid PID,FLAGS

Waits for a particular child process to terminate and returns the pid of the deceased process, or -1 if there is no such child process. On some systems, a value of 0 indicates that there are processes still running. The status is returned in $? and ${^CHILD_ERROR_NATIVE}. If you say

```
use POSIX ":sys_wait_h";
#...
do {
    $kid = waitpid(-1, WNOHANG);
} while $kid > 0;
```

then you can do a non-blocking wait for all pending zombie processes. Non-blocking wait is available on machines supporting either the waitpid(2) or wait4(2) syscalls. However, waiting for a particular pid with FLAGS of 0 is implemented everywhere. (Perl emulates the system call by remembering the status values of processes that have exited but have not been harvested by the Perl script yet.)

Note that on some systems, a return value of -1 could mean that child processes are being automatically reaped. See Section 36.1 [perlipc NAME], page 667 for details, and for other examples.

Portability issues: [perlport waitpid], page 980.

wantarray

Returns true if the context of the currently executing subroutine or eval is looking for a list value. Returns false if the context is looking for a scalar. Returns the undefined value if the context is looking for no value (void context).

```
return unless defined wantarray; # don't bother doing more
my @a = complex_calculation();
return wantarray ? @a : "@a";
```

wantarray()'s result is unspecified in the top level of a file, in a BEGIN, UNITCHECK, CHECK, INIT or END block, or in a DESTROY method.

This function should have been named wantlist() instead.

warn LIST

Prints the value of LIST to STDERR. If the last element of LIST does not end in a newline, it appends the same file/line number text as die does.

If the output is empty and $@ already contains a value (typically from a previous eval) that value is used after appending "\t...caught" to $@. This is useful for staying almost, but not entirely similar to die.

If $@ is empty then the string "Warning: Something's wrong" is used.

No message is printed if there is a $SIG{__WARN__} handler installed. It is the handler's responsibility to deal with the message as it sees fit (like, for instance, converting it into a die). Most handlers must therefore arrange to actually display the warnings that they are not prepared to deal with, by calling warn again in the handler. Note that this is quite safe and will not produce an endless loop, since __WARN__ hooks are not called from inside one.

You will find this behavior is slightly different from that of $SIG{__DIE__} handlers (which don't suppress the error text, but can instead call die again to change it).

Using a __WARN__ handler provides a powerful way to silence all warnings (even the so-called mandatory ones). An example:

```
# wipe out *all* compile-time warnings
BEGIN { $SIG{'__WARN__'} = sub { warn $_[0] if $DOWARN } }
my $foo = 10;
my $foo = 20;          # no warning about duplicate my $foo,
                       # but hey, you asked for it!
# no compile-time or run-time warnings before here
$DOWARN = 1;

# run-time warnings enabled after here
warn "\$foo is alive and $foo!";     # does show up
```

See Section 86.1 [perlvar NAME], page 1375 for details on setting %SIG entries and for more examples. See the Carp module for other kinds of warnings using its carp() and cluck() functions.

write FILEHANDLE
write EXPR
write

Writes a formatted record (possibly multi-line) to the specified FILEHANDLE, using the format associated with that file. By default the format for a file is the one having the same name as the filehandle, but the format for the current output channel (see the select function) may be set explicitly by assigning the name of the format to the $~ variable.

Top of form processing is handled automatically: if there is insufficient room on the current page for the formatted record, the page is advanced by writing a form feed and a special top-of-page format is used to format the new page header before the record is written. By default, the top-of-page format is the name of the filehandle with "_TOP" appended, or "top" in the current package if the former does not exist. This would be a problem with autovivified filehandles, but it may be dynamically set to the format of your choice by assigning the name to the $^ variable while that filehandle is selected. The number of lines remaining on the current page is in variable $-, which can be set to 0 to force a new page.

If FILEHANDLE is unspecified, output goes to the current default output channel, which starts out as STDOUT but may be changed by the select operator. If the FILEHANDLE is an EXPR, then the expression is evaluated and the resulting string is used to look up the name of the FILEHANDLE at run time. For more on formats, see Section 24.1 [perlform NAME], page 343.

Note that write is *not* the opposite of `read`. Unfortunately.

y///

The transliteration operator. Same as `tr///`. See Section 48.2.31 [perlop Quote-Like Operators], page 832.

25.2.4 Non-function Keywords by Cross-reference

25.2.4.1 perldata

__DATA__

__END__

These keywords are documented in Section 11.2.5.3 [perldata Special Literals], page 78.

25.2.4.2 perlmod

BEGIN

CHECK

END

INIT

UNITCHECK

These compile phase keywords are documented in Section 40.2.4 [perlmod BEGIN, UNITCHECK, CHECK, INIT and END], page 736.

25.2.4.3 perlobj

DESTROY

This method keyword is documented in Section 46.2.14 [perlobj Destructors], page 782.

25.2.4.4 perlop

and

cmp

eq

ge

gt

le

lt

ne

not

or

x

xor

These operators are documented in Section 48.1 [perlop NAME], page 798.

25.2.4.5 perlsub

AUTOLOAD

This keyword is documented in Section 73.3.13 [perlsub Autoloading], page 1246.

25.2.4.6 perlsyn

else

elsif

for

foreach

if

unless

until

while

These flow-control keywords are documented in Section 74.2.6 [perlsyn Compound Statements], page 1252.

elseif

The "else if" keyword is spelled `elsif` in Perl. There's no `elif` or `else if` either. It does parse `elseif`, but only to warn you about not using it.

See the documentation for flow-control keywords in Section 74.2.6 [perlsyn Compound Statements], page 1252.

default

given

when

These flow-control keywords related to the experimental switch feature are documented in Section 74.2.11 [perlsyn Switch Statements], page 1258.

26 perlgit

26.1 NAME

perlgit - Detailed information about git and the Perl repository

26.2 DESCRIPTION

This document provides details on using git to develop Perl. If you are just interested in working on a quick patch, see Section 29.1 [perlhack NAME], page 562 first. This document is intended for people who are regular contributors to Perl, including those with write access to the git repository.

26.3 CLONING THE REPOSITORY

All of Perl's source code is kept centrally in a Git repository at *perl5.git.perl.org*.

You can make a read-only clone of the repository by running:

```
% git clone git://perl5.git.perl.org/perl.git perl
```

This uses the git protocol (port 9418).

If you cannot use the git protocol for firewall reasons, you can also clone via http, though this is much slower:

```
% git clone http://perl5.git.perl.org/perl.git perl
```

26.4 WORKING WITH THE REPOSITORY

Once you have changed into the repository directory, you can inspect it. After a clone the repository will contain a single local branch, which will be the current branch as well, as indicated by the asterisk.

```
% git branch
* blead
```

Using the -a switch to `branch` will also show the remote tracking branches in the repository:

```
% git branch -a
* blead
  origin/HEAD
  origin/blead
...
```

The branches that begin with "origin" correspond to the "git remote" that you cloned from (which is named "origin"). Each branch on the remote will be exactly tracked by these branches. You should NEVER do work on these remote tracking branches. You only ever do work in a local branch. Local branches can be configured to automerge (on pull) from a designated remote tracking branch. This is the case with the default branch `blead` which will be configured to merge from the remote tracking branch `origin/blead`.

You can see recent commits:

```
% git log
```

And pull new changes from the repository, and update your local repository (must be clean first)

```
% git pull
```

Assuming we are on the branch `blead` immediately after a pull, this command would be more or less equivalent to:

```
% git fetch
% git merge origin/blead
```

In fact if you want to update your local repository without touching your working directory you do:

```
% git fetch
```

And if you want to update your remote-tracking branches for all defined remotes simultaneously you can do

```
% git remote update
```

Neither of these last two commands will update your working directory, however both will update the remote-tracking branches in your repository.

To make a local branch of a remote branch:

```
% git checkout -b maint-5.10 origin/maint-5.10
```

To switch back to blead:

```
% git checkout blead
```

26.4.1 Finding out your status

The most common git command you will use will probably be

```
% git status
```

This command will produce as output a description of the current state of the repository, including modified files and unignored untracked files, and in addition it will show things like what files have been staged for the next commit, and usually some useful information about how to change things. For instance the following:

```
$ git status
# On branch blead
# Your branch is ahead of ’origin/blead’ by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   pod/perlgit.pod
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   pod/perlgit.pod
#
# Untracked files:
```

```
#   (use "git add <file>..." to include in what will be committed)
#
#       deliberate.untracked
```

This shows that there were changes to this document staged for commit, and that there were further changes in the working directory not yet staged. It also shows that there was an untracked file in the working directory, and as you can see shows how to change all of this. It also shows that there is one commit on the working branch `blead` which has not been pushed to the `origin` remote yet. **NOTE**: that this output is also what you see as a template if you do not provide a message to `git commit`.

26.4.2 Patch workflow

First, please read Section 29.1 [perlhack NAME], page 562 for details on hacking the Perl core. That document covers many details on how to create a good patch.

If you already have a Perl repository, you should ensure that you're on the *blead* branch, and your repository is up to date:

```
% git checkout blead
% git pull
```

It's preferable to patch against the latest blead version, since this is where new development occurs for all changes other than critical bug fixes. Critical bug fix patches should be made against the relevant maint branches, or should be submitted with a note indicating all the branches where the fix should be applied.

Now that we have everything up to date, we need to create a temporary new branch for these changes and switch into it:

```
% git checkout -b orange
```

which is the short form of

```
% git branch orange
% git checkout orange
```

Creating a topic branch makes it easier for the maintainers to rebase or merge back into the master blead for a more linear history. If you don't work on a topic branch the maintainer has to manually cherry pick your changes onto blead before they can be applied.

That'll get you scolded on perl5-porters, so don't do that. Be Awesome.

Then make your changes. For example, if Leon Brocard changes his name to Orange Brocard, we should change his name in the AUTHORS file:

```
% perl -pi -e 's{Leon Brocard}{Orange Brocard}' AUTHORS
```

You can see what files are changed:

```
% git status
# On branch orange
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    modified:   AUTHORS
#
```

And you can see the changes:

```
% git diff
diff --git a/AUTHORS b/AUTHORS
index 293dd70..722c93e 100644
--- a/AUTHORS
+++ b/AUTHORS
@@ -541,7 +541,7 @@    Lars Hecking                  <lhecking@nmrc.ucc.ie>
 Laszlo Molnar                  <laszlo.molnar@eth.ericsson.se>
 Leif Huhn                      <leif@hale.dkstat.com>
 Len Johnson                    <lenjay@ibm.net>
-Leon Brocard                   <acme@astray.com>
+Orange Brocard                 <acme@astray.com>
 Les Peters                     <lpeters@aol.net>
 Lesley Binks                   <lesley.binks@gmail.com>
 Lincoln D. Stein               <lstein@cshl.org>
```

Now commit your change locally:

```
% git commit -a -m 'Rename Leon Brocard to Orange Brocard'
Created commit 6196c1d: Rename Leon Brocard to Orange Brocard
 1 files changed, 1 insertions(+), 1 deletions(-)
```

The `-a` option is used to include all files that git tracks that you have changed. If at this time, you only want to commit some of the files you have worked on, you can omit the `-a` and use the command `git add` *`FILE ...`* before doing the commit. `git add --interactive` allows you to even just commit portions of files instead of all the changes in them.

The `-m` option is used to specify the commit message. If you omit it, git will open a text editor for you to compose the message interactively. This is useful when the changes are more complex than the sample given here, and, depending on the editor, to know that the first line of the commit message doesn't exceed the 50 character legal maximum.

Once you've finished writing your commit message and exited your editor, git will write your change to disk and tell you something like this:

```
Created commit daf8e63: explain git status and stuff about remotes
 1 files changed, 83 insertions(+), 3 deletions(-)
```

If you re-run `git status`, you should see something like this:

```
% git status
# On branch blead
# Your branch is ahead of 'origin/blead' by 2 commits.
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       deliberate.untracked
nothing added to commit but untracked files present (use "git add" to track)
```

When in doubt, before you do anything else, check your status and read it carefully, many questions are answered directly by the git status output.

You can examine your last commit with:

```
% git show HEAD
```

and if you are not happy with either the description or the patch itself you can fix it up by editing the files once more and then issue:

```
% git commit -a --amend
```

Now you should create a patch file for all your local changes:

```
% git format-patch -M blead..
0001-Rename-Leon-Brocard-to-Orange-Brocard.patch
```

Or for a lot of changes, e.g. from a topic branch:

```
% git format-patch --stdout -M blead.. > topic-branch-changes.patch
```

You should now send an email to perlbug@perl.org (`mailto:perlbug@perl.org`) with a description of your changes, and include this patch file as an attachment. In addition to being tracked by RT, mail to perlbug will automatically be forwarded to perl5-porters (with manual moderation, so please be patient). You should only send patches to perl5-porters@perl.org (`mailto:perl5-porters@perl.org`) directly if the patch is not ready to be applied, but intended for discussion.

Please do not use git-send-email(1) to send your patch. See Section 26.4.4 [Sending patch emails], page 497 for more information.

If you want to delete your temporary branch, you may do so with:

```
% git checkout blead
% git branch -d orange
error: The branch 'orange' is not an ancestor of your current HEAD.
If you are sure you want to delete it, run 'git branch -D orange'.
% git branch -D orange
Deleted branch orange.
```

26.4.3 Committing your changes

Assuming that you'd like to commit all the changes you've made as a single atomic unit, run this command:

```
 % git commit -a
```

(That `-a` tells git to add every file you've changed to this commit. New files aren't automatically added to your commit when you use `commit -a` If you want to add files or to commit some, but not all of your changes, have a look at the documentation for `git add`.)

Git will start up your favorite text editor, so that you can craft a commit message for your change. See Section 29.7.2.2 [perlhack Commit message], page 566 for more information about what makes a good commit message.

Once you've finished writing your commit message and exited your editor, git will write your change to disk and tell you something like this:

```
Created commit daf8e63: explain git status and stuff about remotes
 1 files changed, 83 insertions(+), 3 deletions(-)
```

If you re-run `git status`, you should see something like this:

```
% git status
# On branch blead
# Your branch is ahead of 'origin/blead' by 2 commits.
#
```

```
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       deliberate.untracked
nothing added to commit but untracked files present (use "git add" to track)
```

When in doubt, before you do anything else, check your status and read it carefully, many questions are answered directly by the git status output.

26.4.4 Sending patch emails

After you've generated your patch you should sent it to perlbug@perl.org (as discussed Section 26.4.2 [in the previous section], page 494) with a normal mail client as an attachment, along with a description of the patch.

You **must not** use git-send-email(1) to send patches generated with git-format-patch(1). The RT ticketing system living behind perlbug@perl.org does not respect the inline contents of E-Mails, sending an inline patch to RT guarantees that your patch will be destroyed.

Someone may download your patch from RT, which will result in the subject (the first line of the commit message) being omitted. See RT #74192 and commit a4583001 for an example. Alternatively someone may apply your patch from RT after it arrived in their mailbox, by which time RT will have modified the inline content of the message. See RT #74532 and commit f9bcfeac for a bad example of this failure mode.

26.4.5 A note on derived files

Be aware that many files in the distribution are derivative–avoid patching them, because git won't see the changes to them, and the build process will overwrite them. Patch the originals instead. Most utilities (like perldoc) are in this category, i.e. patch `utils/perldoc.PL` rather than `utils/perldoc`. Similarly, don't create patches for files under $src_root/ext from their copies found in $install_root/lib. If you are unsure about the proper location of a file that may have gotten copied while building the source distribution, consult the `MANIFEST`.

26.4.6 Cleaning a working directory

The command `git clean` can with varying arguments be used as a replacement for `make clean`.

To reset your working directory to a pristine condition you can do:

```
% git clean -dxf
```

However, be aware this will delete ALL untracked content. You can use

```
% git clean -Xf
```

to remove all ignored untracked files, such as build and test byproduct, but leave any manually created files alone.

If you only want to cancel some uncommitted edits, you can use `git checkout` and give it a list of files to be reverted, or `git checkout -f` to revert them all.

If you want to cancel one or several commits, you can use `git reset`.

26.4.7 Bisecting

git provides a built-in way to determine which commit should be blamed for introducing a given bug. git bisect performs a binary search of history to locate the first failing commit. It is fast, powerful and flexible, but requires some setup and to automate the process an auxiliary shell script is needed.

The core provides a wrapper program, Porting/bisect.pl, which attempts to simplify as much as possible, making bisecting as simple as running a Perl one-liner. For example, if you want to know when this became an error:

```
perl -e 'my $a := 2'
```

you simply run this:

```
.../Porting/bisect.pl -e 'my $a := 2;'
```

Using bisect.pl, with one command (and no other files) it's easy to find out

- Which commit caused this example code to break?
- Which commit caused this example code to start working?
- Which commit added the first file to match this regex?
- Which commit removed the last file to match this regex?

usually without needing to know which versions of perl to use as start and end revisions, as bisect.pl automatically searches to find the earliest stable version for which the test case passes. Run Porting/bisect.pl --help for the full documentation, including how to set the Configure and build time options.

If you require more flexibility than Porting/bisect.pl has to offer, you'll need to run git bisect yourself. It's most useful to use git bisect run to automate the building and testing of perl revisions. For this you'll need a shell script for git to call to test a particular revision. An example script is Porting/bisect-example.sh, which you should copy **outside** of the repository, as the bisect process will reset the state to a clean checkout as it runs. The instructions below assume that you copied it as ~/run and then edited it as appropriate.

You first enter in bisect mode with:

```
% git bisect start
```

For example, if the bug is present on HEAD but wasn't in 5.10.0, git will learn about this when you enter:

```
% git bisect bad
% git bisect good perl-5.10.0
Bisecting: 853 revisions left to test after this
```

This results in checking out the median commit between HEAD and perl-5.10.0. You can then run the bisecting process with:

```
% git bisect run ~/run
```

When the first bad commit is isolated, git bisect will tell you so:

```
ca4cfd28534303b82a216cfe83a1c80cbc3b9dc5 is first bad commit
commit ca4cfd28534303b82a216cfe83a1c80cbc3b9dc5
Author: Dave Mitchell <davem@fdisolutions.com>
Date:   Sat Feb 9 14:56:23 2008 +0000
```

```
    [perl #49472] Attributes + Unknown Error
    ...

bisect run success
```

You can peek into the bisecting process with `git bisect log` and `git bisect visualize`. `git bisect reset` will get you out of bisect mode.

Please note that the first `good` state must be an ancestor of the first `bad` state. If you want to search for the commit that *solved* some bug, you have to negate your test case (i.e. exit with `1` if OK and `0` if not) and still mark the lower bound as `good` and the upper as `bad`. The "first bad commit" has then to be understood as the "first commit where the bug is solved".

`git help bisect` has much more information on how you can tweak your binary searches.

26.4.8 Topic branches and rewriting history

Individual committers should create topic branches under **yourname/some_descriptive_name**. Other committers should check with a topic branch's creator before making any change to it.

The simplest way to create a remote topic branch that works on all versions of git is to push the current head as a new branch on the remote, then check it out locally:

```
$ branch="$yourname/$some_descriptive_name"
$ git push origin HEAD:$branch
$ git checkout -b $branch origin/$branch
```

Users of git 1.7 or newer can do it in a more obvious manner:

```
$ branch="$yourname/$some_descriptive_name"
$ git checkout -b $branch
$ git push origin -u $branch
```

If you are not the creator of **yourname/some_descriptive_name**, you might sometimes find that the original author has edited the branch's history. There are lots of good reasons for this. Sometimes, an author might simply be rebasing the branch onto a newer source point. Sometimes, an author might have found an error in an early commit which they wanted to fix before merging the branch to blead.

Currently the master repository is configured to forbid non-fast-forward merges. This means that the branches within can not be rebased and pushed as a single step.

The only way you will ever be allowed to rebase or modify the history of a pushed branch is to delete it and push it as a new branch under the same name. Please think carefully about doing this. It may be better to sequentially rename your branches so that it is easier for others working with you to cherry-pick their local changes onto the new version. (XXX: needs explanation).

If you want to rebase a personal topic branch, you will have to delete your existing topic branch and push as a new version of it. You can do this via the following formula (see the explanation about `refspec`'s in the git push documentation for details) after you have rebased your branch:

```
# first rebase
$ git checkout $user/$topic
$ git fetch
$ git rebase origin/blead

# then "delete-and-push"
$ git push origin :$user/$topic
$ git push origin $user/$topic
```

NOTE: it is forbidden at the repository level to delete any of the "primary" branches. That is any branch matching `m!^(blead|maint|perl)!`. Any attempt to do so will result in git producing an error like this:

```
$ git push origin :blead
*** It is forbidden to delete blead/maint branches in this repository
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/blead
To ssh://perl5.git.perl.org/perl
 ! [remote rejected] blead (hook declined)
 error: failed to push some refs to 'ssh://perl5.git.perl.org/perl'
```

As a matter of policy we do **not** edit the history of the blead and maint-* branches. If a typo (or worse) sneaks into a commit to blead or maint-*, we'll fix it in another commit. The only types of updates allowed on these branches are "fast-forward's", where all history is preserved.

Annotated tags in the canonical perl.git repository will never be deleted or modified. Think long and hard about whether you want to push a local tag to perl.git before doing so. (Pushing unannotated tags is not allowed.)

26.4.9 Grafts

The perl history contains one mistake which was not caught in the conversion: a merge was recorded in the history between blead and maint-5.10 where no merge actually occurred. Due to the nature of git, this is now impossible to fix in the public repository. You can remove this mis-merge locally by adding the following line to your `.git/info/grafts` file:

```
296f12bbbbaa06de9be9d09d3dcf8f4528898a49 434946e0cb7a32589ed92d18008aaa1d88515930
```

It is particularly important to have this graft line if any bisecting is done in the area of the "merge" in question.

26.5 WRITE ACCESS TO THE GIT REPOSITORY

Once you have write access, you will need to modify the URL for the origin remote to enable pushing. Edit `.git/config` with the git-config(1) command:

```
% git config remote.origin.url ssh://perl5.git.perl.org/perl.git
```

You can also set up your user name and e-mail address. Most people do this once globally in their `~/.gitconfig` by doing something like:

```
% git config --global user.name "Ævar Arnfjörð Bjarmason"
% git config --global user.email avarab@gmail.com
```

However, if you'd like to override that just for perl, execute something like the following in `perl`:

```
% git config user.email avar@cpan.org
```

It is also possible to keep `origin` as a git remote, and add a new remote for ssh access:

```
% git remote add camel perl5.git.perl.org:/perl.git
```

This allows you to update your local repository by pulling from `origin`, which is faster and doesn't require you to authenticate, and to push your changes back with the `camel` remote:

```
% git fetch camel
% git push camel
```

The `fetch` command just updates the `camel` refs, as the objects themselves should have been fetched when pulling from `origin`.

26.5.1 Accepting a patch

If you have received a patch file generated using the above section, you should try out the patch.

First we need to create a temporary new branch for these changes and switch into it:

```
% git checkout -b experimental
```

Patches that were formatted by `git format-patch` are applied with `git am`:

```
% git am 0001-Rename-Leon-Brocard-to-Orange-Brocard.patch
Applying Rename Leon Brocard to Orange Brocard
```

If just a raw diff is provided, it is also possible use this two-step process:

```
% git apply bugfix.diff
% git commit -a -m "Some fixing" --author="That Guy <that.guy@internets.com>"
```

Now we can inspect the change:

```
% git show HEAD
commit b1b3dab48344cff6de4087efca3dbd63548ab5e2
Author: Leon Brocard <acme@astray.com>
Date:   Fri Dec 19 17:02:59 2008 +0000

  Rename Leon Brocard to Orange Brocard

diff --git a/AUTHORS b/AUTHORS
index 293dd70..722c93e 100644
--- a/AUTHORS
+++ b/AUTHORS
@@ -541,7 +541,7 @@ Lars Hecking                        <lhecking@nmrc.ucc.ie>
 Laszlo Molnar                  <laszlo.molnar@eth.ericsson.se>
 Leif Huhn                      <leif@hale.dkstat.com>
 Len Johnson                    <lenjay@ibm.net>
-Leon Brocard                   <acme@astray.com>
+Orange Brocard                 <acme@astray.com>
 Les Peters                     <lpeters@aol.net>
 Lesley Binks                   <lesley.binks@gmail.com>
 Lincoln D. Stein              <lstein@cshl.org>
```

If you are a committer to Perl and you think the patch is good, you can then merge it into blead then push it out to the main repository:

```
% git checkout blead
% git merge experimental
% git push origin blead
```

If you want to delete your temporary branch, you may do so with:

```
% git checkout blead
% git branch -d experimental
error: The branch 'experimental' is not an ancestor of your current HEAD.
If you are sure you want to delete it, run 'git branch -D experimental'.
% git branch -D experimental
Deleted branch experimental.
```

26.5.2 Committing to blead

The 'blead' branch will become the next production release of Perl.

Before pushing *any* local change to blead, it's incredibly important that you do a few things, lest other committers come after you with pitchforks and torches:

- Make sure you have a good commit message. See Section 29.7.2.2 [perlhack Commit message], page 566 for details.
- Run the test suite. You might not think that one typo fix would break a test file. You'd be wrong. Here's an example of where not running the suite caused problems. A patch was submitted that added a couple of tests to an existing .t. It couldn't possibly affect anything else, so no need to test beyond the single affected .t, right? But, the submitter's email address had changed since the last of their submissions, and this caused other tests to fail. Running the test target given in the next item would have caught this problem.
- If you don't run the full test suite, at least `make test_porting`. This will run basic sanity checks. To see which sanity checks, have a look in `t/porting`.
- If you make any changes that affect miniperl or core routines that have different code paths for miniperl, be sure to run `make minitest`. This will catch problems that even the full test suite will not catch because it runs a subset of tests under miniperl rather than perl.

26.5.3 On merging and rebasing

Simple, one-off commits pushed to the 'blead' branch should be simple commits that apply cleanly. In other words, you should make sure your work is committed against the current position of blead, so that you can push back to the master repository without merging.

Sometimes, blead will move while you're building or testing your changes. When this happens, your push will be rejected with a message like this:

```
To ssh://perl5.git.perl.org/perl.git
 ! [rejected]        blead -> blead (non-fast-forward)
error: failed to push some refs to 'ssh://perl5.git.perl.org/perl.git'
To prevent you from losing history, non-fast-forward updates were rejected
```

```
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
```

When this happens, you can just *rebase* your work against the new position of blead, like this (assuming your remote for the master repository is "p5p"):

```
$ git fetch p5p
$ git rebase p5p/blead
```

You will see your commits being re-applied, and you will then be able to push safely. More information about rebasing can be found in the documentation for the git-rebase(1) command.

For larger sets of commits that only make sense together, or that would benefit from a summary of the set's purpose, you should use a merge commit. You should perform your work on a Section 26.4.8 [topic branch], page 499, which you should regularly rebase against blead to ensure that your code is not broken by blead moving. When you have finished your work, please perform a final rebase and test. Linear history is something that gets lost with every commit on blead, but a final rebase makes the history linear again, making it easier for future maintainers to see what has happened. Rebase as follows (assuming your work was on the branch `committer/somework`):

```
$ git checkout committer/somework
$ git rebase blead
```

Then you can merge it into master like this:

```
$ git checkout blead
$ git merge --no-ff --no-commit committer/somework
$ git commit -a
```

The switches above deserve explanation. `--no-ff` indicates that even if all your work can be applied linearly against blead, a merge commit should still be prepared. This ensures that all your work will be shown as a side branch, with all its commits merged into the mainstream blead by the merge commit.

`--no-commit` means that the merge commit will be *prepared* but not *committed*. The commit is then actually performed when you run the next command, which will bring up your editor to describe the commit. Without `--no-commit`, the commit would be made with nearly no useful message, which would greatly diminish the value of the merge commit as a placeholder for the work's description.

When describing the merge commit, explain the purpose of the branch, and keep in mind that this description will probably be used by the eventual release engineer when reviewing the next perldelta document.

26.5.4 Committing to maintenance versions

Maintenance versions should only be altered to add critical bug fixes, see Section 55.1 [perlpolicy NAME], page 943.

To commit to a maintenance version of perl, you need to create a local tracking branch:

```
% git checkout --track -b maint-5.005 origin/maint-5.005
```

This creates a local branch named `maint-5.005`, which tracks the remote branch `origin/maint-5.005`. Then you can pull, commit, merge and push as before.

You can also cherry-pick commits from blead and another branch, by using the `git cherry-pick` command. It is recommended to use the `-x` option to `git cherry-pick` in order to record the SHA1 of the original commit in the new commit message.

Before pushing any change to a maint version, make sure you've satisfied the steps in Section 26.5.2 [Committing to blead], page 502 above.

26.5.5 Merging from a branch via GitHub

While we don't encourage the submission of patches via GitHub, that will still happen. Here is a guide to merging patches from a GitHub repository.

```
% git remote add avar git://github.com/avar/perl.git
% git fetch avar
```

Now you can see the differences between the branch and blead:

```
% git diff avar/orange
```

And you can see the commits:

```
% git log avar/orange
```

If you approve of a specific commit, you can cherry pick it:

```
% git cherry-pick 0c24b290ae02b2ab3304f51d5e11e85eb3659eae
```

Or you could just merge the whole branch if you like it all:

```
% git merge avar/orange
```

And then push back to the repository:

```
% git push origin blead
```

26.5.6 Using a smoke-me branch to test changes

Sometimes a change affects code paths which you cannot test on the OSes which are directly available to you and it would be wise to have users on other OSes test the change before you commit it to blead.

Fortunately, there is a way to get your change smoke-tested on various OSes: push it to a "smoke-me" branch and wait for certain automated smoke-testers to report the results from their OSes.

The procedure for doing this is roughly as follows (using the example of of tonyc's smoke-me branch called win32stat):

First, make a local branch and switch to it:

```
% git checkout -b win32stat
```

Make some changes, build perl and test your changes, then commit them to your local branch. Then push your local branch to a remote smoke-me branch:

```
% git push origin win32stat:smoke-me/tonyc/win32stat
```

Now you can switch back to blead locally:

```
% git checkout blead
```

and continue working on other things while you wait a day or two, keeping an eye on the results reported for your smoke-me branch at `http://perl.develop-help.com/?b=smoke-me/tonyc/win32state`.

If all is well then update your blead branch:

```
% git pull
```

then checkout your smoke-me branch once more and rebase it on blead:

```
% git rebase blead win32stat
```

Now switch back to blead and merge your smoke-me branch into it:

```
% git checkout blead
% git merge win32stat
```

As described earlier, if there are many changes on your smoke-me branch then you should prepare a merge commit in which to give an overview of those changes by using the following command instead of the last command above:

```
% git merge win32stat --no-ff --no-commit
```

You should now build perl and test your (merged) changes one last time (ideally run the whole test suite, but failing that at least run the `t/porting/*.t` tests) before pushing your changes as usual:

```
% git push origin blead
```

Finally, you should then delete the remote smoke-me branch:

```
% git push origin :smoke-me/tonyc/win32stat
```

(which is likely to produce a warning like this, which can be ignored:

```
remote: fatal: ambiguous argument 'refs/heads/smoke-me/tonyc/win32stat':
unknown revision or path not in the working tree.
remote: Use '--' to separate paths from revisions
```

) and then delete your local branch:

```
% git branch -d win32stat
```

26.5.7 A note on camel and dromedary

The committers have SSH access to the two servers that serve `perl5.git.perl.org`. One is `perl5.git.perl.org` itself (*camel*), which is the 'master' repository. The second one is `users.perl5.git.perl.org` (*dromedary*), which can be used for general testing and development. Dromedary syncs the git tree from camel every few minutes, you should not push there. Both machines also have a full CPAN mirror in /srv/CPAN, please use this. To share files with the general public, dromedary serves your ~/public_html/ as `http://users.perl5.git.perl.org/~yourlogin/`

These hosts have fairly strict firewalls to the outside. Outgoing, only rsync, ssh and git are allowed. For http and ftp, you can use http://webproxy:3128 as proxy. Incoming, the firewall tries to detect attacks and blocks IP addresses with suspicious activity. This sometimes (but very rarely) has false positives and you might get blocked. The quickest way to get unblocked is to notify the admins.

These two boxes are owned, hosted, and operated by booking.com. You can reach the sysadmins in #p5p on irc.perl.org or via mail to `perl5-porters@perl.org`.

27 perlgpl

27.1 NAME

perlgpl - the GNU General Public License, version 1

27.2 SYNOPSIS

```
 You can refer to this document in Pod via "L<perlgpl>"
 Or you can see this document by entering "perldoc perlgpl"
```

27.3 DESCRIPTION

Perl is free software; you can redistribute it and/or modify it under the terms of either:

```
        a) the GNU General Public License as published by the Free
        Software Foundation; either version 1, or (at your option) any
        later version, or

        b) the "Artistic License" which comes with this Kit.
```

This is the "**GNU General Public License, version 1**". It's here so that modules, programs, etc., that want to declare this as their distribution license can link to it.

For the Perl Artistic License, see Section 3.1 [perlartistic NAME], page 18.

27.4 GNU GENERAL PUBLIC LICENSE

```
                    GNU GENERAL PUBLIC LICENSE
                     Version 1, February 1989

  Copyright (C) 1989 Free Software Foundation, Inc.
                     51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA

  Everyone is permitted to copy and distribute verbatim copies
  of this license document, but changing it is not allowed.

                            Preamble

   The license agreements of most software companies try to keep users
 at the mercy of those companies.  By contrast, our General Public
 License is intended to guarantee your freedom to share and change free
 software--to make sure the software is free for all its users.  The
 General Public License applies to the Free Software Foundation's
 software and to any other program whose authors commit to using it.
 You can use it for your programs, too.

   When we speak of free software, we are referring to freedom, not
 price.  Specifically, the General Public License is designed to make
 sure that you have the freedom to give away or sell copies of free
```

software, that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of a such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any work containing the Program or a portion of it, either verbatim or with modifications. Each licensee is addressed as "you".

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this General Public License and to the absence of any warranty; and give any other recipients of the Program a copy of this General Public License along with the Program. You may charge a fee for the physical act of transferring a copy.

2. You may modify your copy or copies of the Program or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:

a) cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and

b) cause the whole of any work that you distribute or publish, that in whole or in part contains the Program or any part thereof, either with or without modifications, to be licensed at no charge to all third parties under the terms of this General Public License (except that you may choose to grant warranty protection to some or all third parties, at your option).

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the simplest and most usual way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this General Public License.

d) You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

Mere aggregation of another independent work with the Program (or its derivative) on a volume of a storage or distribution medium does not bring the other work under the scope of these terms.

3. You may copy and distribute the Program (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:

a) accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,

b) accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal charge for the cost of distribution) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,

c) accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs, or for standard header files or definitions files that accompany that operating system.

4. You may not copy, modify, sublicense, distribute or transfer the Program except as expressly provided under this General Public License. Any attempt otherwise to copy, modify, sublicense, distribute or transfer the Program is void, and will automatically terminate your rights to use the Program under this License. However, parties who have received copies, or rights to use copies, from you under this General Public License will not have their licenses terminated so long as such parties remain in full compliance.

5. By copying, distributing or modifying the Program (or any work based on the Program) you indicate your acceptance of this license to do so, and all its terms and conditions.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.

7. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of the license which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the license, you may choose any version ever published by the Free Software Foundation.

8. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free

Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

9. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

10. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to humanity, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) 19yy  <name of author>

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 1, or (at your option)
    any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston MA
    02110-1301 USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
    Gnomovision version 69, Copyright (C) 19xx name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type 'show c' for details.
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here a sample; alter the names:

```
  Yoyodyne, Inc., hereby disclaims all copyright interest in the
  program 'Gnomovision' (a program to direct compilers to make passes
  at assemblers) written by James Hacker.

  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice
```

That's all there is to it!

28 perlguts

28.1 NAME

perlguts - Introduction to the Perl API

28.2 DESCRIPTION

This document attempts to describe how to use the Perl API, as well as to provide some info on the basic workings of the Perl core. It is far from complete and probably contains many errors. Please refer any questions or comments to the author below.

28.3 Variables

28.3.1 Datatypes

Perl has three typedefs that handle Perl's three main data types:

```
SV  Scalar Value
AV  Array Value
HV  Hash Value
```

Each typedef has specific routines that manipulate the various data types.

28.3.2 What is an "IV"?

Perl uses a special typedef IV which is a simple signed integer type that is guaranteed to be large enough to hold a pointer (as well as an integer). Additionally, there is the UV, which is simply an unsigned IV.

Perl also uses two special typedefs, I32 and I16, which will always be at least 32-bits and 16-bits long, respectively. (Again, there are U32 and U16, as well.) They will usually be exactly 32 and 16 bits long, but on Crays they will both be 64 bits.

28.3.3 Working with SVs

An SV can be created and loaded with one command. There are five types of values that can be loaded: an integer value (IV), an unsigned integer value (UV), a double (NV), a string (PV), and another scalar (SV). ("PV" stands for "Pointer Value". You might think that it is misnamed because it is described as pointing only to strings. However, it is possible to have it point to other things. For example, it could point to an array of UVs. But, using it for non-strings requires care, as the underlying assumption of much of the internals is that PVs are just for strings. Often, for example, a trailing `NUL` is tacked on automatically. The non-string use is documented only in this paragraph.)

The seven routines are:

```
SV*  newSViv(IV);
SV*  newSVuv(UV);
SV*  newSVnv(double);
SV*  newSVpv(const char*, STRLEN);
SV*  newSVpvn(const char*, STRLEN);
SV*  newSVpvf(const char*, ...);
```

```
SV*  newSVsv(SV*);
```

STRLEN is an integer type (Size_t, usually defined as size_t in config.h) guaranteed to be large enough to represent the size of any string that perl can handle.

In the unlikely case of a SV requiring more complex initialization, you can create an empty SV with newSV(len). If len is 0 an empty SV of type NULL is returned, else an SV of type PV is returned with len + 1 (for the NUL) bytes of storage allocated, accessible via SvPVX. In both cases the SV has the undef value.

```
SV *sv = newSV(0);   /* no storage allocated  */
SV *sv = newSV(10);  /* 10 (+1) bytes of uninitialised storage
                      * allocated */
```

To change the value of an *already-existing* SV, there are eight routines:

```
void  sv_setiv(SV*, IV);
void  sv_setuv(SV*, UV);
void  sv_setnv(SV*, double);
void  sv_setpv(SV*, const char*);
void  sv_setpvn(SV*, const char*, STRLEN)
void  sv_setpvf(SV*, const char*, ...);
void  sv_vsetpvfn(SV*, const char*, STRLEN, va_list *,
                                            SV **, I32, bool *);
void  sv_setsv(SV*, SV*);
```

Notice that you can choose to specify the length of the string to be assigned by using sv_setpvn, newSVpvn, or newSVpv, or you may allow Perl to calculate the length by using sv_setpv or by specifying 0 as the second argument to newSVpv. Be warned, though, that Perl will determine the string's length by using strlen, which depends on the string terminating with a NUL character, and not otherwise containing NULs.

The arguments of sv_setpvf are processed like sprintf, and the formatted output becomes the value.

sv_vsetpvfn is an analogue of vsprintf, but it allows you to specify either a pointer to a variable argument list or the address and length of an array of SVs. The last argument points to a boolean; on return, if that boolean is true, then locale-specific information has been used to format the string, and the string's contents are therefore untrustworthy (see Section 70.1 [perlsec NAME], page 1198). This pointer may be NULL if that information is not important. Note that this function requires you to specify the length of the format.

The sv_set*() functions are not generic enough to operate on values that have "magic". See Section 28.3.20 [Magic Virtual Tables], page 529 later in this document.

All SVs that contain strings should be terminated with a NUL character. If it is not NUL-terminated there is a risk of core dumps and corruptions from code which passes the string to C functions or system calls which expect a NUL-terminated string. Perl's own functions typically add a trailing NUL for this reason. Nevertheless, you should be very careful when you pass a string stored in an SV to a C function or system call.

To access the actual value that an SV points to, you can use the macros:

```
SvIV(SV*)
SvUV(SV*)
SvNV(SV*)
```

```
SvPV(SV*, STRLEN len)
SvPV_nolen(SV*)
```

which will automatically coerce the actual scalar type into an IV, UV, double, or string.

In the SvPV macro, the length of the string returned is placed into the variable len (this is a macro, so you do *not* use &len). If you do not care what the length of the data is, use the SvPV_nolen macro. Historically the SvPV macro with the global variable PL_na has been used in this case. But that can be quite inefficient because PL_na must be accessed in thread-local storage in threaded Perl. In any case, remember that Perl allows arbitrary strings of data that may both contain NULs and might not be terminated by a NUL.

Also remember that C doesn't allow you to safely say foo(SvPV(s, len), len);. It might work with your compiler, but it won't work for everyone. Break this sort of statement up into separate assignments:

```
SV *s;
STRLEN len;
char *ptr;
ptr = SvPV(s, len);
foo(ptr, len);
```

If you want to know if the scalar value is TRUE, you can use:

```
SvTRUE(SV*)
```

Although Perl will automatically grow strings for you, if you need to force Perl to allocate more memory for your SV, you can use the macro

```
SvGROW(SV*, STRLEN newlen)
```

which will determine if more memory needs to be allocated. If so, it will call the function sv_grow. Note that SvGROW can only increase, not decrease, the allocated memory of an SV and that it does not automatically add space for the trailing NUL byte (perl's own string functions typically do SvGROW(sv, len + 1)).

If you want to write to an existing SV's buffer and set its value to a string, use SvPV_force() or one of its variants to force the SV to be a PV. This will remove any of various types of non-stringness from the SV while preserving the content of the SV in the PV. This can be used, for example, to append data from an API function to a buffer without extra copying:

```
(void)SvPVbyte_force(sv, len);
s = SvGROW(sv, len + needlen + 1);
/* something that modifies up to needlen bytes at s+len, but
   modifies newlen bytes
     eg. newlen = read(fd, s + len, needlen);
   ignoring errors for these examples
 */
s[len + newlen] = '\0';
SvCUR_set(sv, len + newlen);
SvUTF8_off(sv);
SvSETMAGIC(sv);
```

If you already have the data in memory or if you want to keep your code simple, you can use one of the sv_cat*() variants, such as sv_catpvn(). If you want to insert anywhere in the string you can use sv_insert() or sv_insert_flags().

If you don't need the existing content of the SV, you can avoid some copying with:

```
 sv_setpvn(sv, "", 0);
 s = SvGROW(sv, needlen + 1);
 /* something that modifies up to needlen bytes at s, but modifies
    newlen bytes
      eg. newlen = read(fd, s. needlen);
  */
 s[newlen] = '\0';
 SvCUR_set(sv, newlen);
 SvPOK_only(sv); /* also clears SVf_UTF8 */
 SvSETMAGIC(sv);
```

Again, if you already have the data in memory or want to avoid the complexity of the above, you can use sv_setpvn().

If you have a buffer allocated with Newx() and want to set that as the SV's value, you can use sv_usepvn_flags(). That has some requirements if you want to avoid perl re-allocating the buffer to fit the trailing NUL:

```
Newx(buf, somesize+1, char);
/* ... fill in buf ... */
buf[somesize] = '\0';
sv_usepvn_flags(sv, buf, somesize, SV_SMAGIC | SV_HAS_TRAILING_NUL);
/* buf now belongs to perl, don't release it */
```

If you have an SV and want to know what kind of data Perl thinks is stored in it, you can use the following macros to check the type of SV you have.

```
 SvIOK(SV*)
 SvNOK(SV*)
 SvPOK(SV*)
```

You can get and set the current length of the string stored in an SV with the following macros:

```
 SvCUR(SV*)
 SvCUR_set(SV*, I32 val)
```

You can also get a pointer to the end of the string stored in the SV with the macro:

```
 SvEND(SV*)
```

But note that these last three macros are valid only if `SvPOK()` is true.

If you want to append something to the end of string stored in an `SV*`, you can use the following functions:

```
 void  sv_catpv(SV*, const char*);
 void  sv_catpvn(SV*, const char*, STRLEN);
 void  sv_catpvf(SV*, const char*, ...);
 void  sv_vcatpvfn(SV*, const char*, STRLEN, va_list *, SV **,
                                                          I32, bool);
 void  sv_catsv(SV*, SV*);
```

The first function calculates the length of the string to be appended by using `strlen`. In the second, you specify the length of the string yourself. The third function processes its arguments like `sprintf` and appends the formatted output. The fourth function works like vsprintf. You can specify the address and length of an array of SVs instead of the va_list argument. The fifth function extends the string stored in the first SV with the string stored in the second SV. It also forces the second SV to be interpreted as a string.

The sv_cat*() functions are not generic enough to operate on values that have "magic". See Section 28.3.20 [Magic Virtual Tables], page 529 later in this document.

If you know the name of a scalar variable, you can get a pointer to its SV by using the following:

```
SV*  get_sv("package::varname", 0);
```

This returns NULL if the variable does not exist.

If you want to know if this variable (or any other SV) is actually defined, you can call:

```
SvOK(SV*)
```

The scalar undef value is stored in an SV instance called PL_sv_undef.

Its address can be used whenever an SV* is needed. Make sure that you don't try to compare a random sv with &PL_sv_undef. For example when interfacing Perl code, it'll work correctly for:

```
foo(undef);
```

But won't work when called as:

```
$x = undef;
foo($x);
```

So to repeat always use SvOK() to check whether an sv is defined.

Also you have to be careful when using &PL_sv_undef as a value in AVs or HVs (see Section 28.3.9 [AVs, HVs and undefined values], page 521).

There are also the two values PL_sv_yes and PL_sv_no, which contain boolean TRUE and FALSE values, respectively. Like PL_sv_undef, their addresses can be used whenever an SV* is needed.

Do not be fooled into thinking that (SV *) 0 is the same as &PL_sv_undef. Take this code:

```
SV* sv = (SV*) 0;
if (I-am-to-return-a-real-value) {
        sv = sv_2mortal(newSViv(42));
}
sv_setsv(ST(0), sv);
```

This code tries to return a new SV (which contains the value 42) if it should return a real value, or undef otherwise. Instead it has returned a NULL pointer which, somewhere down the line, will cause a segmentation violation, bus error, or just weird results. Change the zero to &PL_sv_undef in the first line and all will be well.

To free an SV that you've created, call SvREFCNT_dec(SV*). Normally this call is not necessary (see Section 28.3.13 [Reference Counts and Mortality], page 524).

28.3.4 Offsets

Perl provides the function sv_chop to efficiently remove characters from the beginning of a string; you give it an SV and a pointer to somewhere inside the PV, and it discards everything before the pointer. The efficiency comes by means of a little hack: instead of actually removing the characters, sv_chop sets the flag OOK (offset OK) to signal to other functions that the offset hack is in effect, and it moves the PV pointer (called SvPVX) forward by the number of bytes chopped off, and adjusts SvCUR and SvLEN accordingly. (A portion of the space between the old and new PV pointers is used to store the count of chopped bytes.)

Hence, at this point, the start of the buffer that we allocated lives at SvPVX(sv) - SvIV(sv) in memory and the PV pointer is pointing into the middle of this allocated storage.

This is best demonstrated by example. Normally copy-on-write will prevent the substitution from operator from using this hack, but if you can craft a string for which copy-on-write is not possible, you can see it in play. In the current implementation, the final byte of a string buffer is used as a copy-on-write reference count. If the buffer is not big enough, then copy-on-write is skipped. First have a look at an empty string:

```
% ./perl -Ilib -MDevel::Peek -le ’$a=""; $a .= ""; Dump $a’
SV = PV(0x7ffb7c008a70) at 0x7ffb7c030390
  REFCNT = 1
  FLAGS = (POK,pPOK)
  PV = 0x7ffb7bc05b50 ""\0
  CUR = 0
  LEN = 10
```

Notice here the LEN is 10. (It may differ on your platform.) Extend the length of the string to one less than 10, and do a substitution:

```
% ./perl -Ilib -MDevel::Peek -le ’$a=""; $a.="123456789"; $a=~s/.//; Dump($a)’
SV = PV(0x7ffa04008a70) at 0x7ffa04030390
  REFCNT = 1
  FLAGS = (POK,OOK,pPOK)
  OFFSET = 1
  PV = 0x7ffa03c05b61 ( "\1" . ) "23456789"\0
  CUR = 8
  LEN = 9
```

Here the number of bytes chopped off (1) is shown next as the OFFSET. The portion of the string between the "real" and the "fake" beginnings is shown in parentheses, and the values of SvCUR and SvLEN reflect the fake beginning, not the real one. (The first character of the string buffer happens to have changed to "\1" here, not "1", because the current implementation stores the offset count in the string buffer. This is subject to change.)

Something similar to the offset hack is performed on AVs to enable efficient shifting and splicing off the beginning of the array; while AvARRAY points to the first element in the array that is visible from Perl, AvALLOC points to the real start of the C array. These are usually the same, but a shift operation can be carried out by increasing AvARRAY by one and decreasing AvFILL and AvMAX. Again, the location of the real start of the C array only comes into play when freeing the array. See av_shift in av.c.

28.3.5 What’s Really Stored in an SV?

Recall that the usual method of determining the type of scalar you have is to use Sv*OK macros. Because a scalar can be both a number and a string, usually these macros will always return TRUE and calling the Sv*V macros will do the appropriate conversion of string to integer/double or integer/double to string.

If you *really* need to know if you have an integer, double, or string pointer in an SV, you can use the following three macros instead:

```
SvIOKp(SV*)
SvNOKp(SV*)
SvPOKp(SV*)
```

These will tell you if you truly have an integer, double, or string pointer stored in your SV. The "p" stands for private.

There are various ways in which the private and public flags may differ. For example, in perl 5.16 and earlier a tied SV may have a valid underlying value in the IV slot (so SvIOKp is true), but the data should be accessed via the FETCH routine rather than directly, so SvIOK is false. (In perl 5.18 onwards, tied scalars use the flags the same way as untied scalars.) Another is when numeric conversion has occurred and precision has been lost: only the private flag is set on 'lossy' values. So when an NV is converted to an IV with loss, SvIOKp, SvNOKp and SvNOK will be set, while SvIOK wont be.

In general, though, it's best to use the Sv*V macros.

28.3.6 Working with AVs

There are two ways to create and load an AV. The first method creates an empty AV:

```
AV*  newAV();
```

The second method both creates the AV and initially populates it with SVs:

```
AV*  av_make(SSize_t num, SV **ptr);
```

The second argument points to an array containing num SV*'s. Once the AV has been created, the SVs can be destroyed, if so desired.

Once the AV has been created, the following operations are possible on it:

```
void  av_push(AV*, SV*);
SV*   av_pop(AV*);
SV*   av_shift(AV*);
void  av_unshift(AV*, SSize_t num);
```

These should be familiar operations, with the exception of av_unshift. This routine adds num elements at the front of the array with the undef value. You must then use av_store (described below) to assign values to these new elements.

Here are some other functions:

```
SSize_t av_top_index(AV*);
SV**    av_fetch(AV*, SSize_t key, I32 lval);
SV**    av_store(AV*, SSize_t key, SV* val);
```

The av_top_index function returns the highest index value in an array (just like $#array in Perl). If the array is empty, -1 is returned. The av_fetch function returns the value at index key, but if lval is non-zero, then av_fetch will store an undef value at that index. The av_store function stores the value val at index key, and does not increment the reference count of val. Thus the caller is responsible for taking care of that, and if av_store returns NULL, the caller will have to decrement the reference count to avoid a memory leak. Note that av_fetch and av_store both return SV**'s, not SV*'s as their return value.

A few more:

```
void  av_clear(AV*);
void  av_undef(AV*);
void  av_extend(AV*, SSize_t key);
```

The av_clear function deletes all the elements in the AV* array, but does not actually delete the array itself. The av_undef function will delete all the elements in the array plus the array itself. The av_extend function extends the array so that it contains at least key+1 elements. If key+1 is less than the currently allocated length of the array, then nothing is done.

If you know the name of an array variable, you can get a pointer to its AV by using the following:

```
AV*  get_av("package::varname", 0);
```

This returns NULL if the variable does not exist.

See Section 28.3.22 [Understanding the Magic of Tied Hashes and Arrays], page 533 for more information on how to use the array access functions on tied arrays.

28.3.7 Working with HVs

To create an HV, you use the following routine:

```
HV*  newHV();
```

Once the HV has been created, the following operations are possible on it:

```
SV**  hv_store(HV*, const char* key, U32 klen, SV* val, U32 hash);
SV**  hv_fetch(HV*, const char* key, U32 klen, I32 lval);
```

The klen parameter is the length of the key being passed in (Note that you cannot pass 0 in as a value of klen to tell Perl to measure the length of the key). The val argument contains the SV pointer to the scalar being stored, and hash is the precomputed hash value (zero if you want hv_store to calculate it for you). The lval parameter indicates whether this fetch is actually a part of a store operation, in which case a new undefined value will be added to the HV with the supplied key and hv_fetch will return as if the value had already existed.

Remember that hv_store and hv_fetch return SV**'s and not just SV*. To access the scalar value, you must first dereference the return value. However, you should check to make sure that the return value is not NULL before dereferencing it.

The first of these two functions checks if a hash table entry exists, and the second deletes it.

```
bool  hv_exists(HV*, const char* key, U32 klen);
SV*   hv_delete(HV*, const char* key, U32 klen, I32 flags);
```

If flags does not include the G_DISCARD flag then hv_delete will create and return a mortal copy of the deleted value.

And more miscellaneous functions:

```
void   hv_clear(HV*);
void   hv_undef(HV*);
```

Like their AV counterparts, hv_clear deletes all the entries in the hash table but does not actually delete the hash table. The hv_undef deletes both the entries and the hash table itself.

Perl keeps the actual data in a linked list of structures with a typedef of HE. These contain the actual key and value pointers (plus extra administrative overhead). The key is a string pointer; the value is an SV*. However, once you have an HE*, to get the actual key and value, use the routines specified below.

```
I32    hv_iterinit(HV*);
        /* Prepares starting point to traverse hash table */
HE*    hv_iternext(HV*);
        /* Get the next entry, and return a pointer to a
           structure that has both the key and value */
char*  hv_iterkey(HE* entry, I32* retlen);
        /* Get the key from an HE structure and also return
           the length of the key string */
SV*    hv_iterval(HV*, HE* entry);
        /* Return an SV pointer to the value of the HE
           structure */
SV*    hv_iternextsv(HV*, char** key, I32* retlen);
        /* This convenience routine combines hv_iternext,
           hv_iterkey, and hv_iterval.  The key and retlen
           arguments are return values for the key and its
           length.  The value is returned in the SV* argument */
```

If you know the name of a hash variable, you can get a pointer to its HV by using the following:

```
HV*  get_hv("package::varname", 0);
```

This returns NULL if the variable does not exist.

The hash algorithm is defined in the PERL_HASH macro:

```
PERL_HASH(hash, key, klen)
```

The exact implementation of this macro varies by architecture and version of perl, and the return value may change per invocation, so the value is only valid for the duration of a single perl process.

See Section 28.3.22 [Understanding the Magic of Tied Hashes and Arrays], page 533 for more information on how to use the hash access functions on tied hashes.

28.3.8 Hash API Extensions

Beginning with version 5.004, the following functions are also supported:

```
HE*     hv_fetch_ent  (HV* tb, SV* key, I32 lval, U32 hash);
HE*     hv_store_ent  (HV* tb, SV* key, SV* val, U32 hash);

bool    hv_exists_ent (HV* tb, SV* key, U32 hash);
SV*     hv_delete_ent (HV* tb, SV* key, I32 flags, U32 hash);

SV*     hv_iterkeysv  (HE* entry);
```

Note that these functions take SV* keys, which simplifies writing of extension code that deals with hash structures. These functions also allow passing of SV* keys to tie functions without forcing you to stringify the keys (unlike the previous set of functions).

They also return and accept whole hash entries (HE*), making their use more efficient (since the hash number for a particular string doesn't have to be recomputed every time). See perlapi for detailed descriptions.

The following macros must always be used to access the contents of hash entries. Note that the arguments to these macros must be simple variables, since they may get evaluated more than once. See perlapi for detailed descriptions of these macros.

```
HePV(HE* he, STRLEN len)
HeVAL(HE* he)
HeHASH(HE* he)
HeSVKEY(HE* he)
HeSVKEY_force(HE* he)
HeSVKEY_set(HE* he, SV* sv)
```

These two lower level macros are defined, but must only be used when dealing with keys that are not SV*s:

```
HeKEY(HE* he)
HeKLEN(HE* he)
```

Note that both hv_store and hv_store_ent do not increment the reference count of the stored val, which is the caller's responsibility. If these functions return a NULL value, the caller will usually have to decrement the reference count of val to avoid a memory leak.

28.3.9 AVs, HVs and undefined values

Sometimes you have to store undefined values in AVs or HVs. Although this may be a rare case, it can be tricky. That's because you're used to using &PL_sv_undef if you need an undefined SV.

For example, intuition tells you that this XS code:

```
AV *av = newAV();
av_store( av, 0, &PL_sv_undef );
```

is equivalent to this Perl code:

```
my @av;
$av[0] = undef;
```

Unfortunately, this isn't true. In perl 5.18 and earlier, AVs use &PL_sv_undef as a marker for indicating that an array element has not yet been initialized. Thus, exists $av[0] would be true for the above Perl code, but false for the array generated by the XS code. In perl 5.20, storing &PL_sv_undef will create a read-only element, because the scalar &PL_sv_undef itself is stored, not a copy.

Similar problems can occur when storing &PL_sv_undef in HVs:

```
hv_store( hv, "key", 3, &PL_sv_undef, 0 );
```

This will indeed make the value undef, but if you try to modify the value of key, you'll get the following error:

```
Modification of non-creatable hash value attempted
```

In perl 5.8.0, `&PL_sv_undef` was also used to mark placeholders in restricted hashes. This caused such hash entries not to appear when iterating over the hash or when checking for the keys with the `hv_exists` function.

You can run into similar problems when you store `&PL_sv_yes` or `&PL_sv_no` into AVs or HVs. Trying to modify such elements will give you the following error:

```
Modification of a read-only value attempted
```

To make a long story short, you can use the special variables `&PL_sv_undef`, `&PL_sv_yes` and `&PL_sv_no` with AVs and HVs, but you have to make sure you know what you're doing.

Generally, if you want to store an undefined value in an AV or HV, you should not use `&PL_sv_undef`, but rather create a new undefined value using the `newSV` function, for example:

```
av_store( av, 42, newSV(0) );
hv_store( hv, "foo", 3, newSV(0), 0 );
```

28.3.10 References

References are a special type of scalar that point to other data types (including other references).

To create a reference, use either of the following functions:

```
SV* newRV_inc((SV*) thing);
SV* newRV_noinc((SV*) thing);
```

The `thing` argument can be any of an `SV*`, `AV*`, or `HV*`. The functions are identical except that `newRV_inc` increments the reference count of the `thing`, while `newRV_noinc` does not. For historical reasons, `newRV` is a synonym for `newRV_inc`.

Once you have a reference, you can use the following macro to dereference the reference:

```
SvRV(SV*)
```

then call the appropriate routines, casting the returned `SV*` to either an `AV*` or `HV*`, if required.

To determine if an SV is a reference, you can use the following macro:

```
SvROK(SV*)
```

To discover what type of value the reference refers to, use the following macro and then check the return value.

```
SvTYPE(SvRV(SV*))
```

The most useful types that will be returned are:

```
< SVt_PVAV  Scalar
SVt_PVAV    Array
SVt_PVHV    Hash
SVt_PVCV    Code
SVt_PVGV    Glob (possibly a file handle)
```

See Section “svtype” in `perlapi` for more details.

28.3.11 Blessed References and Class Objects

References are also used to support object-oriented programming. In perl's OO lexicon, an object is simply a reference that has been blessed into a package (or class). Once blessed, the programmer may now use the reference to access the various methods in the class.

A reference can be blessed into a package with the following function:

```
SV* sv_bless(SV* sv, HV* stash);
```

The `sv` argument must be a reference value. The `stash` argument specifies which class the reference will belong to. See Section 28.3.14 [Stashes and Globs], page 525 for information on converting class names into stashes.

/* Still under construction */

The following function upgrades rv to reference if not already one. Creates a new SV for rv to point to. If `classname` is non-null, the SV is blessed into the specified class. SV is returned.

```
    SV* newSVrv(SV* rv, const char* classname);
```

The following three functions copy integer, unsigned integer or double into an SV whose reference is `rv`. SV is blessed if `classname` is non-null.

```
    SV* sv_setref_iv(SV* rv, const char* classname, IV iv);
    SV* sv_setref_uv(SV* rv, const char* classname, UV uv);
    SV* sv_setref_nv(SV* rv, const char* classname, NV iv);
```

The following function copies the pointer value (*the address, not the string!*) into an SV whose reference is rv. SV is blessed if `classname` is non-null.

```
    SV* sv_setref_pv(SV* rv, const char* classname, void* pv);
```

The following function copies a string into an SV whose reference is `rv`. Set length to 0 to let Perl calculate the string length. SV is blessed if `classname` is non-null.

```
SV* sv_setref_pvn(SV* rv, const char* classname, char* pv,
                                                     STRLEN length);
```

The following function tests whether the SV is blessed into the specified class. It does not check inheritance relationships.

```
    int  sv_isa(SV* sv, const char* name);
```

The following function tests whether the SV is a reference to a blessed object.

```
    int  sv_isobject(SV* sv);
```

The following function tests whether the SV is derived from the specified class. SV can be either a reference to a blessed object or a string containing a class name. This is the function implementing the `UNIVERSAL::isa` functionality.

```
    bool sv_derived_from(SV* sv, const char* name);
```

To check if you've got an object derived from a specific class you have to write:

```
    if (sv_isobject(sv) && sv_derived_from(sv, class)) { ... }
```

28.3.12 Creating New Variables

To create a new Perl variable with an undef value which can be accessed from your Perl script, use the following routines, depending on the variable type.

```
SV*  get_sv("package::varname", GV_ADD);
AV*  get_av("package::varname", GV_ADD);
HV*  get_hv("package::varname", GV_ADD);
```

Notice the use of GV_ADD as the second parameter. The new variable can now be set, using the routines appropriate to the data type.

There are additional macros whose values may be bitwise OR'ed with the GV_ADD argument to enable certain extra features. Those bits are:

GV_ADDMULTI

Marks the variable as multiply defined, thus preventing the:

```
Name <varname> used only once: possible typo
```

warning.

GV_ADDWARN

Issues the warning:

```
Had to create <varname> unexpectedly
```

if the variable did not exist before the function was called.

If you do not specify a package name, the variable is created in the current package.

28.3.13 Reference Counts and Mortality

Perl uses a reference count-driven garbage collection mechanism. SVs, AVs, or HVs (xV for short in the following) start their life with a reference count of 1. If the reference count of an xV ever drops to 0, then it will be destroyed and its memory made available for reuse.

This normally doesn't happen at the Perl level unless a variable is undef'ed or the last variable holding a reference to it is changed or overwritten. At the internal level, however, reference counts can be manipulated with the following macros:

```
int SvREFCNT(SV* sv);
SV* SvREFCNT_inc(SV* sv);
void SvREFCNT_dec(SV* sv);
```

However, there is one other function which manipulates the reference count of its argument. The newRV_inc function, you will recall, creates a reference to the specified argument. As a side effect, it increments the argument's reference count. If this is not what you want, use newRV_noinc instead.

For example, imagine you want to return a reference from an XSUB function. Inside the XSUB routine, you create an SV which initially has a reference count of one. Then you call newRV_inc, passing it the just-created SV. This returns the reference as a new SV, but the reference count of the SV you passed to newRV_inc has been incremented to two. Now you return the reference from the XSUB routine and forget about the SV. But Perl hasn't! Whenever the returned reference is destroyed, the reference count of the original SV is decreased to one and nothing happens. The SV will hang around without any way to access it until Perl itself terminates. This is a memory leak.

The correct procedure, then, is to use newRV_noinc instead of newRV_inc. Then, if and when the last reference is destroyed, the reference count of the SV will go to zero and it will be destroyed, stopping any memory leak.

There are some convenience functions available that can help with the destruction of xVs. These functions introduce the concept of "mortality". An xV that is mortal has had its reference count marked to be decremented, but not actually decremented, until "a short time later". Generally the term "short time later" means a single Perl statement, such as a call to an XSUB function. The actual determinant for when mortal xVs have their reference count decremented depends on two macros, SAVETMPS and FREETMPS. See Section 7.1 [perlcall NAME], page 28 and `perlxs` for more details on these macros.

"Mortalization" then is at its simplest a deferred `SvREFCNT_dec`. However, if you mortalize a variable twice, the reference count will later be decremented twice.

"Mortal" SVs are mainly used for SVs that are placed on perl's stack. For example an SV which is created just to pass a number to a called sub is made mortal to have it cleaned up automatically when it's popped off the stack. Similarly, results returned by XSUBs (which are pushed on the stack) are often made mortal.

To create a mortal variable, use the functions:

```
SV*  sv_newmortal()
SV*  sv_2mortal(SV*)
SV*  sv_mortalcopy(SV*)
```

The first call creates a mortal SV (with no value), the second converts an existing SV to a mortal SV (and thus defers a call to `SvREFCNT_dec`), and the third creates a mortal copy of an existing SV. Because `sv_newmortal` gives the new SV no value, it must normally be given one via `sv_setpv`, `sv_setiv`, etc. :

```
SV *tmp = sv_newmortal();
sv_setiv(tmp, an_integer);
```

As that is multiple C statements it is quite common so see this idiom instead:

```
SV *tmp = sv_2mortal(newSViv(an_integer));
```

You should be careful about creating mortal variables. Strange things can happen if you make the same value mortal within multiple contexts, or if you make a variable mortal multiple times. Thinking of "Mortalization" as deferred `SvREFCNT_dec` should help to minimize such problems. For example if you are passing an SV which you *know* has a high enough REFCNT to survive its use on the stack you need not do any mortalization. If you are not sure then doing an `SvREFCNT_inc` and `sv_2mortal`, or making a `sv_mortalcopy` is safer.

The mortal routines are not just for SVs; AVs and HVs can be made mortal by passing their address (type-casted to `SV*`) to the `sv_2mortal` or `sv_mortalcopy` routines.

28.3.14 Stashes and Globs

A **stash** is a hash that contains all variables that are defined within a package. Each key of the stash is a symbol name (shared by all the different types of objects that have the same name), and each value in the hash table is a GV (Glob Value). This GV in turn contains references to the various objects of that name, including (but not limited to) the following:

```
Scalar Value
Array Value
Hash Value
I/O Handle
```

```
Format
Subroutine
```

There is a single stash called `PL_defstash` that holds the items that exist in the `main` package. To get at the items in other packages, append the string "::" to the package name. The items in the `Foo` package are in the stash `Foo::` in PL_defstash. The items in the `Bar::Baz` package are in the stash `Baz::` in `Bar::`'s stash.

To get the stash pointer for a particular package, use the function:

```
HV*  gv_stashpv(const char* name, I32 flags)
HV*  gv_stashsv(SV*, I32 flags)
```

The first function takes a literal string, the second uses the string stored in the SV. Remember that a stash is just a hash table, so you get back an `HV*`. The `flags` flag will create a new package if it is set to GV_ADD.

The name that `gv_stash*v` wants is the name of the package whose symbol table you want. The default package is called `main`. If you have multiply nested packages, pass their names to `gv_stash*v`, separated by `::` as in the Perl language itself.

Alternately, if you have an SV that is a blessed reference, you can find out the stash pointer by using:

```
HV*  SvSTASH(SvRV(SV*));
```

then use the following to get the package name itself:

```
char*  HvNAME(HV* stash);
```

If you need to bless or re-bless an object you can use the following function:

```
SV*  sv_bless(SV*, HV* stash)
```

where the first argument, an `SV*`, must be a reference, and the second argument is a stash. The returned `SV*` can now be used in the same way as any other SV.

For more information on references and blessings, consult Section 62.1 [perlref NAME], page 1077.

28.3.15 Double-Typed SVs

Scalar variables normally contain only one type of value, an integer, double, pointer, or reference. Perl will automatically convert the actual scalar data from the stored type into the requested type.

Some scalar variables contain more than one type of scalar data. For example, the variable `$!` contains either the numeric value of `errno` or its string equivalent from either `strerror` or `sys_errlist[]`.

To force multiple data values into an SV, you must do two things: use the `sv_set*v` routines to add the additional scalar type, then set a flag so that Perl will believe it contains more than one type of data. The four macros to set the flags are:

```
SvIOK_on
SvNOK_on
SvPOK_on
SvROK_on
```

The particular macro you must use depends on which `sv_set*v` routine you called first. This is because every `sv_set*v` routine turns on only the bit for the particular type of data being set, and turns off all the rest.

For example, to create a new Perl variable called "dberror" that contains both the numeric and descriptive string error values, you could use the following code:

```
extern int  dberror;
extern char *dberror_list;

SV* sv = get_sv("dberror", GV_ADD);
sv_setiv(sv, (IV) dberror);
sv_setpv(sv, dberror_list[dberror]);
SvIOK_on(sv);
```

If the order of `sv_setiv` and `sv_setpv` had been reversed, then the macro `SvPOK_on` would need to be called instead of `SvIOK_on`.

28.3.16 Read-Only Values

In Perl 5.16 and earlier, copy-on-write (see the next section) shared a flag bit with read-only scalars. So the only way to test whether `sv_setsv`, etc., will raise a "Modification of a read-only value" error in those versions is:

```
SvREADONLY(sv) && !SvIsCOW(sv)
```

Under Perl 5.18 and later, SvREADONLY only applies to read-only variables, and, under 5.20, copy-on-write scalars can also be read-only, so the above check is incorrect. You just want:

```
SvREADONLY(sv)
```

If you need to do this check often, define your own macro like this:

```
#if PERL_VERSION >= 18
# define SvTRULYREADONLY(sv) SvREADONLY(sv)
#else
# define SvTRULYREADONLY(sv) (SvREADONLY(sv) && !SvIsCOW(sv))
#endif
```

28.3.17 Copy on Write

Perl implements a copy-on-write (COW) mechanism for scalars, in which string copies are not immediately made when requested, but are deferred until made necessary by one or the other scalar changing. This is mostly transparent, but one must take care not to modify string buffers that are shared by multiple SVs.

You can test whether an SV is using copy-on-write with `SvIsCOW(sv)`.

You can force an SV to make its own copy of its string buffer by calling `sv_force_normal(sv)` or SvPV_force_nolen(sv).

If you want to make the SV drop its string buffer, use `sv_force_normal_flags(sv, SV_COW_DROP_PV)` or simply `sv_setsv(sv, NULL)`.

All of these functions will croak on read-only scalars (see the previous section for more on those).

To test that your code is behaving correctly and not modifying COW buffers, on systems that support mmap(2) (i.e., Unix) you can configure perl with `-Accflags=-DPERL_DEBUG_READONLY_COW` and it will turn buffer violations into crashes. You will find it to be marvellously slow, so you may want to skip perl's own tests.

28.3.18 Magic Variables

[This section still under construction. Ignore everything here. Post no bills. Everything not permitted is forbidden.]

Any SV may be magical, that is, it has special features that a normal SV does not have. These features are stored in the SV structure in a linked list of `struct magic`'s, typedef'ed to `MAGIC`.

```
    struct magic {
        MAGIC*      mg_moremagic;
        MGVTBL*     mg_virtual;
        U16         mg_private;
        char        mg_type;
        U8          mg_flags;
        I32         mg_len;
        SV*         mg_obj;
        char*       mg_ptr;
    };
```

Note this is current as of patchlevel 0, and could change at any time.

28.3.19 Assigning Magic

Perl adds magic to an SV using the sv_magic function:

```
  void sv_magic(SV* sv, SV* obj, int how, const char* name, I32 namlen);
```

The `sv` argument is a pointer to the SV that is to acquire a new magical feature.

If `sv` is not already magical, Perl uses the `SvUPGRADE` macro to convert `sv` to type `SVt_PVMG`. Perl then continues by adding new magic to the beginning of the linked list of magical features. Any prior entry of the same type of magic is deleted. Note that this can be overridden, and multiple instances of the same type of magic can be associated with an SV.

The `name` and `namlen` arguments are used to associate a string with the magic, typically the name of a variable. `namlen` is stored in the `mg_len` field and if `name` is non-null then either a `savepvn` copy of `name` or `name` itself is stored in the `mg_ptr` field, depending on whether `namlen` is greater than zero or equal to zero respectively. As a special case, if `(name && namlen == HEf_SVKEY)` then `name` is assumed to contain an `SV*` and is stored as-is with its REFCNT incremented.

The sv_magic function uses `how` to determine which, if any, predefined "Magic Virtual Table" should be assigned to the `mg_virtual` field. See the Section 28.3.20 [Magic Virtual Tables], page 529 section below. The `how` argument is also stored in the `mg_type` field. The value of `how` should be chosen from the set of macros `PERL_MAGIC_foo` found in `perl.h`. Note that before these macros were added, Perl internals used to directly use character literals, so you may occasionally come across old code or documentation referring to 'U' magic rather than `PERL_MAGIC_uvar` for example.

The `obj` argument is stored in the `mg_obj` field of the `MAGIC` structure. If it is not the same as the `sv` argument, the reference count of the `obj` object is incremented. If it is the same, or if the `how` argument is `PERL_MAGIC_arylen`, or if it is a NULL pointer, then `obj` is merely stored, without the reference count being incremented.

See also `sv_magicext` in `perlapi` for a more flexible way to add magic to an SV.

There is also a function to add magic to an `HV`:

```
void hv_magic(HV *hv, GV *gv, int how);
```

This simply calls `sv_magic` and coerces the `gv` argument into an `SV`.

To remove the magic from an SV, call the function sv_unmagic:

```
int sv_unmagic(SV *sv, int type);
```

The `type` argument should be equal to the `how` value when the `SV` was initially made magical.

However, note that `sv_unmagic` removes all magic of a certain `type` from the `SV`. If you want to remove only certain magic of a `type` based on the magic virtual table, use `sv_unmagicext` instead:

```
 int sv_unmagicext(SV *sv, int type, MGVTBL *vtbl);
```

28.3.20 Magic Virtual Tables

The `mg_virtual` field in the `MAGIC` structure is a pointer to an `MGVTBL`, which is a structure of function pointers and stands for "Magic Virtual Table" to handle the various operations that might be applied to that variable.

The `MGVTBL` has five (or sometimes eight) pointers to the following routine types:

```
int  (*svt_get)(SV* sv, MAGIC* mg);
int  (*svt_set)(SV* sv, MAGIC* mg);
U32  (*svt_len)(SV* sv, MAGIC* mg);
int  (*svt_clear)(SV* sv, MAGIC* mg);
int  (*svt_free)(SV* sv, MAGIC* mg);

int  (*svt_copy)(SV *sv, MAGIC* mg, SV *nsv,
                                    const char *name, I32 namlen);
int  (*svt_dup)(MAGIC *mg, CLONE_PARAMS *param);
int  (*svt_local)(SV *nsv, MAGIC *mg);
```

This MGVTBL structure is set at compile-time in `perl.h` and there are currently 32 types. These different structures contain pointers to various routines that perform additional actions depending on which function is being called.

```
Function pointer    Action taken
----------------    ------------
svt_get             Do something before the value of the SV is
                    retrieved.
svt_set             Do something after the SV is assigned a value.
svt_len             Report on the SV's length.
svt_clear           Clear something the SV represents.
svt_free            Free any extra storage associated with the SV.

svt_copy            copy tied variable magic to a tied element
svt_dup             duplicate a magic structure during thread cloning
svt_local           copy magic to local value during 'local'
```

For instance, the MGVTBL structure called `vtbl_sv` (which corresponds to an `mg_type` of `PERL_MAGIC_sv`) contains:

```
  { magic_get, magic_set, magic_len, 0, 0 }
```

Thus, when an SV is determined to be magical and of type `PERL_MAGIC_sv`, if a get operation is being performed, the routine `magic_get` is called. All the various routines for the various magical types begin with `magic_`. NOTE: the magic routines are not considered part of the Perl API, and may not be exported by the Perl library.

The last three slots are a recent addition, and for source code compatibility they are only checked for if one of the three flags MGf_COPY, MGf_DUP or MGf_LOCAL is set in mg_flags. This means that most code can continue declaring a vtable as a 5-element value. These three are currently used exclusively by the threading code, and are highly subject to change.

The current kinds of Magic Virtual Tables are:

```
mg_type
(old-style char and macro)   MGVTBL         Type of magic
--------------------------   ------         -------------
\0 PERL_MAGIC_sv             vtbl_sv        Special scalar variable
#  PERL_MAGIC_arylen         vtbl_arylen    Array length ($#ary)
%  PERL_MAGIC_rhash          (none)         Extra data for restricted
                                            hashes
*  PERL_MAGIC_debugvar       vtbl_debugvar  $DB::single, signal, trace
                                            vars
.  PERL_MAGIC_pos            vtbl_pos       pos() lvalue
:  PERL_MAGIC_symtab         (none)         Extra data for symbol
                                            tables
<  PERL_MAGIC_backref        vtbl_backref   For weak ref data
@  PERL_MAGIC_arylen_p       (none)         To move arylen out of XPVAV
B  PERL_MAGIC_bm             vtbl_regexp    Boyer-Moore
                                            (fast string search)
c  PERL_MAGIC_overload_table vtbl_ovrld     Holds overload table
                                            (AMT) on stash
D  PERL_MAGIC_regdata        vtbl_regdata   Regex match position data
                                            (@+ and @- vars)
d  PERL_MAGIC_regdatum       vtbl_regdatum  Regex match position data
                                            element
E  PERL_MAGIC_env            vtbl_env       %ENV hash
e  PERL_MAGIC_envelem        vtbl_envelem   %ENV hash element
f  PERL_MAGIC_fm             vtbl_regexp    Formline
                                            ('compiled' format)
g  PERL_MAGIC_regex_global   vtbl_mglob     m//g target
H  PERL_MAGIC_hints          vtbl_hints     %^H hash
h  PERL_MAGIC_hintselem      vtbl_hintselem %^H hash element
I  PERL_MAGIC_isa            vtbl_isa       @ISA array
i  PERL_MAGIC_isaelem        vtbl_isaelem   @ISA array element
k  PERL_MAGIC_nkeys          vtbl_nkeys     scalar(keys()) lvalue
L  PERL_MAGIC_dbfile         (none)         Debugger %_<filename
```

l	PERL_MAGIC_dbline	vtbl_dbline	Debugger %_<filename element
N	PERL_MAGIC_shared	(none)	Shared between threads
n	PERL_MAGIC_shared_scalar	(none)	Shared between threads
o	PERL_MAGIC_collxfrm	vtbl_collxfrm	Locale transformation
P	PERL_MAGIC_tied	vtbl_pack	Tied array or hash
p	PERL_MAGIC_tiedelem	vtbl_packelem	Tied array or hash element
q	PERL_MAGIC_tiedscalar	vtbl_packelem	Tied scalar or handle
r	PERL_MAGIC_qr	vtbl_regexp	Precompiled qr// regex
S	PERL_MAGIC_sig	(none)	%SIG hash
s	PERL_MAGIC_sigelem	vtbl_sigelem	%SIG hash element
t	PERL_MAGIC_taint	vtbl_taint	Taintedness
U	PERL_MAGIC_uvar	vtbl_uvar	Available for use by extensions
u	PERL_MAGIC_uvar_elem	(none)	Reserved for use by extensions
V	PERL_MAGIC_vstring	(none)	SV was vstring literal
v	PERL_MAGIC_vec	vtbl_vec	vec() lvalue
w	PERL_MAGIC_utf8	vtbl_utf8	Cached UTF-8 information
x	PERL_MAGIC_substr	vtbl_substr	substr() lvalue
y	PERL_MAGIC_defelem	vtbl_defelem	Shadow "foreach" iterator variable / smart parameter vivification
\	PERL_MAGIC_lvref	vtbl_lvref	Lvalue reference constructor
]	PERL_MAGIC_checkcall	vtbl_checkcall	Inlining/mutation of call to this CV
~	PERL_MAGIC_ext	(none)	Available for use by extensions

When an uppercase and lowercase letter both exist in the table, then the uppercase letter is typically used to represent some kind of composite type (a list or a hash), and the lowercase letter is used to represent an element of that composite type. Some internals code makes use of this case relationship. However, 'v' and 'V' (vec and v-string) are in no way related.

The PERL_MAGIC_ext and PERL_MAGIC_uvar magic types are defined specifically for use by extensions and will not be used by perl itself. Extensions can use PERL_MAGIC_ext magic to 'attach' private information to variables (typically objects). This is especially useful because there is no way for normal perl code to corrupt this private information (unlike using extra elements of a hash object).

Similarly, PERL_MAGIC_uvar magic can be used much like tie() to call a C function any time a scalar's value is used or changed. The MAGIC's mg_ptr field points to a ufuncs structure:

```
struct ufuncs {
    I32 (*uf_val)(pTHX_ IV, SV*);
    I32 (*uf_set)(pTHX_ IV, SV*);
    IV uf_index;
};
```

When the SV is read from or written to, the `uf_val` or `uf_set` function will be called with `uf_index` as the first arg and a pointer to the SV as the second. A simple example of how to add `PERL_MAGIC_uvar` magic is shown below. Note that the ufuncs structure is copied by sv_magic, so you can safely allocate it on the stack.

```
void
Umagic(sv)
    SV *sv;
PREINIT:
    struct ufuncs uf;
CODE:
    uf.uf_val   = &my_get_fn;
    uf.uf_set   = &my_set_fn;
    uf.uf_index = 0;
    sv_magic(sv, 0, PERL_MAGIC_uvar, (char*)&uf, sizeof(uf));
```

Attaching `PERL_MAGIC_uvar` to arrays is permissible but has no effect.

For hashes there is a specialized hook that gives control over hash keys (but not values). This hook calls `PERL_MAGIC_uvar` 'get' magic if the "set" function in the `ufuncs` structure is NULL. The hook is activated whenever the hash is accessed with a key specified as an `SV` through the functions `hv_store_ent`, `hv_fetch_ent`, `hv_delete_ent`, and `hv_exists_ent`. Accessing the key as a string through the functions without the `..._ent` suffix circumvents the hook. See Section "GUTS" in `Hash-Util-FieldHash` for a detailed description.

Note that because multiple extensions may be using `PERL_MAGIC_ext` or `PERL_MAGIC_uvar` magic, it is important for extensions to take extra care to avoid conflict. Typically only using the magic on objects blessed into the same class as the extension is sufficient. For `PERL_MAGIC_ext` magic, it is usually a good idea to define an `MGVTBL`, even if all its fields will be `0`, so that individual `MAGIC` pointers can be identified as a particular kind of magic using their magic virtual table. `mg_findext` provides an easy way to do that:

```
STATIC MGVTBL my_vtbl = { 0, 0, 0, 0, 0, 0, 0, 0 };

MAGIC *mg;
if ((mg = mg_findext(sv, PERL_MAGIC_ext, &my_vtbl))) {
    /* this is really ours, not another module's PERL_MAGIC_ext */
    my_priv_data_t *priv = (my_priv_data_t *)mg->mg_ptr;
    ...
}
```

Also note that the `sv_set*()` and `sv_cat*()` functions described earlier do **not** invoke 'set' magic on their targets. This must be done by the user either by calling the `SvSETMAGIC()` macro after calling these functions, or by using one of the `sv_set*_mg()` or `sv_cat*_mg()` functions. Similarly, generic C code must call the `SvGETMAGIC()` macro to invoke any 'get' magic if they use an SV obtained from external sources in functions that don't handle magic. See `perlapi` for a description of these functions. For example, calls to the `sv_cat*()` functions typically need to be followed by `SvSETMAGIC()`, but they don't need a prior `SvGETMAGIC()` since their implementation handles 'get' magic.

28.3.21 Finding Magic

```
MAGIC *mg_find(SV *sv, int type); /* Finds the magic pointer of that
                                   * type */
```

This routine returns a pointer to a MAGIC structure stored in the SV. If the SV does not have that magical feature, NULL is returned. If the SV has multiple instances of that magical feature, the first one will be returned. mg_findext can be used to find a MAGIC structure of an SV based on both its magic type and its magic virtual table:

```
MAGIC *mg_findext(SV *sv, int type, MGVTBL *vtbl);
```

Also, if the SV passed to mg_find or mg_findext is not of type SVt_PVMG, Perl may core dump.

```
int mg_copy(SV* sv, SV* nsv, const char* key, STRLEN klen);
```

This routine checks to see what types of magic sv has. If the mg_type field is an uppercase letter, then the mg_obj is copied to nsv, but the mg_type field is changed to be the lowercase letter.

28.3.22 Understanding the Magic of Tied Hashes and Arrays

Tied hashes and arrays are magical beasts of the PERL_MAGIC_tied magic type.

WARNING: As of the 5.004 release, proper usage of the array and hash access functions requires understanding a few caveats. Some of these caveats are actually considered bugs in the API, to be fixed in later releases, and are bracketed with [MAYCHANGE] below. If you find yourself actually applying such information in this section, be aware that the behavior may change in the future, umm, without warning.

The perl tie function associates a variable with an object that implements the various GET, SET, etc methods. To perform the equivalent of the perl tie function from an XSUB, you must mimic this behaviour. The code below carries out the necessary steps – firstly it creates a new hash, and then creates a second hash which it blesses into the class which will implement the tie methods. Lastly it ties the two hashes together, and returns a reference to the new tied hash. Note that the code below does NOT call the TIEHASH method in the MyTie class - see Section 28.4.3 [Calling Perl Routines from within C Programs], page 538 for details on how to do this.

```
SV*
mytie()
PREINIT:
    HV *hash;
    HV *stash;
    SV *tie;
CODE:
    hash = newHV();
    tie = newRV_noinc((SV*)newHV());
    stash = gv_stashpv("MyTie", GV_ADD);
    sv_bless(tie, stash);
    hv_magic(hash, (GV*)tie, PERL_MAGIC_tied);
    RETVAL = newRV_noinc(hash);
OUTPUT:
    RETVAL
```

The av_store function, when given a tied array argument, merely copies the magic of the array onto the value to be "stored", using mg_copy. It may also return NULL, indicating that the value did not actually need to be stored in the array. [MAYCHANGE] After a call to av_store on a tied array, the caller will usually need to call mg_set(val) to actually invoke the perl level "STORE" method on the TIEARRAY object. If av_store did return NULL, a call to SvREFCNT_dec(val) will also be usually necessary to avoid a memory leak. [/MAYCHANGE]

The previous paragraph is applicable verbatim to tied hash access using the hv_store and hv_store_ent functions as well.

av_fetch and the corresponding hash functions hv_fetch and hv_fetch_ent actually return an undefined mortal value whose magic has been initialized using mg_copy. Note the value so returned does not need to be deallocated, as it is already mortal. [MAYCHANGE] But you will need to call mg_get() on the returned value in order to actually invoke the perl level "FETCH" method on the underlying TIE object. Similarly, you may also call mg_set() on the return value after possibly assigning a suitable value to it using sv_setsv, which will invoke the "STORE" method on the TIE object. [/MAYCHANGE]

[MAYCHANGE] In other words, the array or hash fetch/store functions don't really fetch and store actual values in the case of tied arrays and hashes. They merely call mg_copy to attach magic to the values that were meant to be "stored" or "fetched". Later calls to mg_get and mg_set actually do the job of invoking the TIE methods on the underlying objects. Thus the magic mechanism currently implements a kind of lazy access to arrays and hashes.

Currently (as of perl version 5.004), use of the hash and array access functions requires the user to be aware of whether they are operating on "normal" hashes and arrays, or on their tied variants. The API may be changed to provide more transparent access to both tied and normal data types in future versions. [/MAYCHANGE]

You would do well to understand that the TIEARRAY and TIEHASH interfaces are mere sugar to invoke some perl method calls while using the uniform hash and array syntax. The use of this sugar imposes some overhead (typically about two to four extra opcodes per FETCH/STORE operation, in addition to the creation of all the mortal variables required to invoke the methods). This overhead will be comparatively small if the TIE methods are themselves substantial, but if they are only a few statements long, the overhead will not be insignificant.

28.3.23 Localizing changes

Perl has a very handy construction

```
{
  local $var = 2;
  ...
}
```

This construction is *approximately* equivalent to

```
{
  my $oldvar = $var;
  $var = 2;
  ...
```

```
    $var = $oldvar;
  }
```

The biggest difference is that the first construction would reinstate the initial value of $var, irrespective of how control exits the block: `goto`, `return`, `die/eval`, etc. It is a little bit more efficient as well.

There is a way to achieve a similar task from C via Perl API: create a *pseudo-block*, and arrange for some changes to be automatically undone at the end of it, either explicit, or via a non-local exit (via die()). A *block*-like construct is created by a pair of `ENTER`/`LEAVE` macros (see Section 7.5.3 [perlcall Returning a Scalar], page 35). Such a construct may be created specially for some important localized task, or an existing one (like boundaries of enclosing Perl subroutine/block, or an existing pair for freeing TMPs) may be used. (In the second case the overhead of additional localization must be almost negligible.) Note that any XSUB is automatically enclosed in an `ENTER`/`LEAVE` pair.

Inside such a *pseudo-block* the following service is available:

`SAVEINT(int i)`
`SAVEIV(IV i)`
`SAVEI32(I32 i)`
`SAVELONG(long i)`

These macros arrange things to restore the value of integer variable `i` at the end of enclosing *pseudo-block*.

`SAVESPTR(s)`
`SAVEPPTR(p)`

These macros arrange things to restore the value of pointers `s` and `p`. `s` must be a pointer of a type which survives conversion to `SV*` and back, `p` should be able to survive conversion to `char*` and back.

`SAVEFREESV(SV *sv)`

The refcount of `sv` would be decremented at the end of *pseudo-block*. This is similar to `sv_2mortal` in that it is also a mechanism for doing a delayed `SvREFCNT_dec`. However, while `sv_2mortal` extends the lifetime of `sv` until the beginning of the next statement, `SAVEFREESV` extends it until the end of the enclosing scope. These lifetimes can be wildly different.

Also compare `SAVEMORTALIZESV`.

`SAVEMORTALIZESV(SV *sv)`

Just like `SAVEFREESV`, but mortalizes `sv` at the end of the current scope instead of decrementing its reference count. This usually has the effect of keeping `sv` alive until the statement that called the currently live scope has finished executing.

`SAVEFREEOP(OP *op)`

The `OP *` is op_free()ed at the end of *pseudo-block*.

`SAVEFREEPV(p)`

The chunk of memory which is pointed to by `p` is Safefree()ed at the end of *pseudo-block*.

`SAVECLEARSV(SV *sv)`

Clears a slot in the current scratchpad which corresponds to `sv` at the end of *pseudo-block*.

`SAVEDELETE(HV *hv, char *key, I32 length)`

The key `key` of `hv` is deleted at the end of *pseudo-block*. The string pointed to by `key` is Safefree()ed. If one has a *key* in short-lived storage, the corresponding string may be reallocated like this:

```
SAVEDELETE(PL_defstash, savepv(tmpbuf), strlen(tmpbuf));
```

`SAVEDESTRUCTOR(DESTRUCTORFUNC_NOCONTEXT_t f, void *p)`

At the end of *pseudo-block* the function `f` is called with the only argument `p`.

`SAVEDESTRUCTOR_X(DESTRUCTORFUNC_t f, void *p)`

At the end of *pseudo-block* the function `f` is called with the implicit context argument (if any), and `p`.

`SAVESTACK_POS()`

The current offset on the Perl internal stack (cf. `SP`) is restored at the end of *pseudo-block*.

The following API list contains functions, thus one needs to provide pointers to the modifiable data explicitly (either C pointers, or Perlish `GV *`s). Where the above macros take `int`, a similar function takes `int *`.

`SV* save_scalar(GV *gv)`

Equivalent to Perl code `local $gv`.

`AV* save_ary(GV *gv)`
`HV* save_hash(GV *gv)`

Similar to `save_scalar`, but localize `@gv` and `%gv`.

`void save_item(SV *item)`

Duplicates the current value of `SV`, on the exit from the current `ENTER`/`LEAVE` *pseudo-block* will restore the value of `SV` using the stored value. It doesn't handle magic. Use `save_scalar` if magic is affected.

`void save_list(SV **sarg, I32 maxsarg)`

A variant of `save_item` which takes multiple arguments via an array `sarg` of `SV*` of length `maxsarg`.

`SV* save_svref(SV **sptr)`

Similar to `save_scalar`, but will reinstate an `SV *`.

`void save_aptr(AV **aptr)`
`void save_hptr(HV **hptr)`

Similar to `save_svref`, but localize `AV *` and `HV *`.

The `Alias` module implements localization of the basic types within the *caller's scope*. People who are interested in how to localize things in the containing scope should take a look there too.

28.4 Subroutines

28.4.1 XSUBs and the Argument Stack

The XSUB mechanism is a simple way for Perl programs to access C subroutines. An XSUB routine will have a stack that contains the arguments from the Perl program, and a way to map from the Perl data structures to a C equivalent.

The stack arguments are accessible through the `ST(n)` macro, which returns the `n`'th stack argument. Argument 0 is the first argument passed in the Perl subroutine call. These arguments are `SV*`, and can be used anywhere an `SV*` is used.

Most of the time, output from the C routine can be handled through use of the RETVAL and OUTPUT directives. However, there are some cases where the argument stack is not already long enough to handle all the return values. An example is the POSIX tzname() call, which takes no arguments, but returns two, the local time zone's standard and summer time abbreviations.

To handle this situation, the PPCODE directive is used and the stack is extended using the macro:

```
EXTEND(SP, num);
```

where `SP` is the macro that represents the local copy of the stack pointer, and `num` is the number of elements the stack should be extended by.

Now that there is room on the stack, values can be pushed on it using `PUSHs` macro. The pushed values will often need to be "mortal" (See Section 28.3.13 [Reference Counts and Mortality], page 524):

```
PUSHs(sv_2mortal(newSViv(an_integer)))
PUSHs(sv_2mortal(newSVuv(an_unsigned_integer)))
PUSHs(sv_2mortal(newSVnv(a_double)))
PUSHs(sv_2mortal(newSVpv("Some String",0)))
/* Although the last example is better written as the more
 * efficient: */
PUSHs(newSVpvs_flags("Some String", SVs_TEMP))
```

And now the Perl program calling `tzname`, the two values will be assigned as in:

```
($standard_abbrev, $summer_abbrev) = POSIX::tzname;
```

An alternate (and possibly simpler) method to pushing values on the stack is to use the macro:

```
XPUSHs(SV*)
```

This macro automatically adjusts the stack for you, if needed. Thus, you do not need to call `EXTEND` to extend the stack.

Despite their suggestions in earlier versions of this document the macros `(X)PUSH[iunp]` are *not* suited to XSUBs which return multiple results. For that, either stick to the `(X)PUSHs` macros shown above, or use the new `m(X)PUSH[iunp]` macros instead; see Section 28.4.4 [Putting a C value on Perl stack], page 539.

For more information, consult `perlxs` and `perlxstut`.

28.4.2 Autoloading with XSUBs

If an AUTOLOAD routine is an XSUB, as with Perl subroutines, Perl puts the fully-qualified name of the autoloaded subroutine in the $AUTOLOAD variable of the XSUB's package.

But it also puts the same information in certain fields of the XSUB itself:

```
HV *stash           = CvSTASH(cv);
const char *subname = SvPVX(cv);
STRLEN name_length  = SvCUR(cv); /* in bytes */
U32 is_utf8         = SvUTF8(cv);
```

`SvPVX(cv)` contains just the sub name itself, not including the package. For an AUTOLOAD routine in UNIVERSAL or one of its superclasses, `CvSTASH(cv)` returns NULL during a method call on a nonexistent package.

Note: Setting $AUTOLOAD stopped working in 5.6.1, which did not support XS AUTOLOAD subs at all. Perl 5.8.0 introduced the use of fields in the XSUB itself. Perl 5.16.0 restored the setting of $AUTOLOAD. If you need to support 5.8-5.14, use the XSUB's fields.

28.4.3 Calling Perl Routines from within C Programs

There are four routines that can be used to call a Perl subroutine from within a C program. These four are:

```
I32  call_sv(SV*, I32);
I32  call_pv(const char*, I32);
I32  call_method(const char*, I32);
I32  call_argv(const char*, I32, char**);
```

The routine most often used is `call_sv`. The `SV*` argument contains either the name of the Perl subroutine to be called, or a reference to the subroutine. The second argument consists of flags that control the context in which the subroutine is called, whether or not the subroutine is being passed arguments, how errors should be trapped, and how to treat return values.

All four routines return the number of arguments that the subroutine returned on the Perl stack.

These routines used to be called `perl_call_sv`, etc., before Perl v5.6.0, but those names are now deprecated; macros of the same name are provided for compatibility.

When using any of these routines (except `call_argv`), the programmer must manipulate the Perl stack. These include the following macros and functions:

```
dSP
SP
PUSHMARK()
PUTBACK
SPAGAIN
ENTER
SAVETMPS
FREETMPS
LEAVE
XPUSH*()
```

```
POP*()
```

For a detailed description of calling conventions from C to Perl, consult Section 7.1 [perlcall NAME], page 28.

28.4.4 Putting a C value on Perl stack

A lot of opcodes (this is an elementary operation in the internal perl stack machine) put an SV* on the stack. However, as an optimization the corresponding SV is (usually) not recreated each time. The opcodes reuse specially assigned SVs (*target*s) which are (as a corollary) not constantly freed/created.

Each of the targets is created only once (but see Section 28.4.6 [Scratchpads and recursion], page 540 below), and when an opcode needs to put an integer, a double, or a string on stack, it just sets the corresponding parts of its *target* and puts the *target* on stack.

The macro to put this target on stack is `PUSHTARG`, and it is directly used in some opcodes, as well as indirectly in zillions of others, which use it via `(X)PUSH[iunp]`.

Because the target is reused, you must be careful when pushing multiple values on the stack. The following code will not do what you think:

```
XPUSHi(10);
XPUSHi(20);
```

This translates as "set `TARG` to 10, push a pointer to `TARG` onto the stack; set `TARG` to 20, push a pointer to `TARG` onto the stack". At the end of the operation, the stack does not contain the values 10 and 20, but actually contains two pointers to `TARG`, which we have set to 20.

If you need to push multiple different values then you should either use the `(X)PUSHs` macros, or else use the new `m(X)PUSH[iunp]` macros, none of which make use of `TARG`. The `(X)PUSHs` macros simply push an SV* on the stack, which, as noted under Section 28.4.1 [XSUBs and the Argument Stack], page 537, will often need to be "mortal". The new `m(X)PUSH[iunp]` macros make this a little easier to achieve by creating a new mortal for you (via `(X)PUSHmortal`), pushing that onto the stack (extending it if necessary in the case of the `mXPUSH[iunp]` macros), and then setting its value. Thus, instead of writing this to "fix" the example above:

```
XPUSHs(sv_2mortal(newSViv(10)))
XPUSHs(sv_2mortal(newSViv(20)))
```

you can simply write:

```
mXPUSHi(10)
mXPUSHi(20)
```

On a related note, if you do use `(X)PUSH[iunp]`, then you're going to need a `dTARG` in your variable declarations so that the `*PUSH*` macros can make use of the local variable `TARG`. See also `dTARGET` and `dXSTARG`.

28.4.5 Scratchpads

The question remains on when the SVs which are *target*s for opcodes are created. The answer is that they are created when the current unit–a subroutine or a file (for opcodes for statements outside of subroutines)–is compiled. During this time a special anonymous Perl array is created, which is called a scratchpad for the current unit.

A scratchpad keeps SVs which are lexicals for the current unit and are targets for opcodes. A previous version of this document stated that one can deduce that an SV lives on a scratchpad by looking on its flags: lexicals have `SVs_PADMY` set, and *target*s have `SVs_PADTMP` set. But this has never been fully true. `SVs_PADMY` could be set on a variable that no longer resides in any pad. While *target*s do have `SVs_PADTMP` set, it can also be set on variables that have never resided in a pad, but nonetheless act like *target*s. As of perl 5.21.5, the `SVs_PADMY` flag is no longer used and is defined as 0. `SvPADMY()` now returns true for anything without `SVs_PADTMP`.

The correspondence between OPs and *target*s is not 1-to-1. Different OPs in the compile tree of the unit can use the same target, if this would not conflict with the expected life of the temporary.

28.4.6 Scratchpads and recursion

In fact it is not 100% true that a compiled unit contains a pointer to the scratchpad AV. In fact it contains a pointer to an AV of (initially) one element, and this element is the scratchpad AV. Why do we need an extra level of indirection?

The answer is **recursion**, and maybe **threads**. Both these can create several execution pointers going into the same subroutine. For the subroutine-child not write over the temporaries for the subroutine-parent (lifespan of which covers the call to the child), the parent and the child should have different scratchpads. (*And* the lexicals should be separate anyway!)

So each subroutine is born with an array of scratchpads (of length 1). On each entry to the subroutine it is checked that the current depth of the recursion is not more than the length of this array, and if it is, new scratchpad is created and pushed into the array.

The *target*s on this scratchpad are `undef`s, but they are already marked with correct flags.

28.5 Memory Allocation

28.5.1 Allocation

All memory meant to be used with the Perl API functions should be manipulated using the macros described in this section. The macros provide the necessary transparency between differences in the actual malloc implementation that is used within perl.

It is suggested that you enable the version of malloc that is distributed with Perl. It keeps pools of various sizes of unallocated memory in order to satisfy allocation requests more quickly. However, on some platforms, it may cause spurious malloc or free errors.

The following three macros are used to initially allocate memory :

```
Newx(pointer, number, type);
Newxc(pointer, number, type, cast);
Newxz(pointer, number, type);
```

The first argument `pointer` should be the name of a variable that will point to the newly allocated memory.

The second and third arguments `number` and `type` specify how many of the specified type of data structure should be allocated. The argument `type` is passed to `sizeof`. The final argument to Newxc, cast, should be used if the pointer argument is different from the type argument.

Unlike the Newx and Newxc macros, the Newxz macro calls memzero to zero out all the newly allocated memory.

28.5.2 Reallocation

```
    Renew(pointer, number, type);
    Renewc(pointer, number, type, cast);
    Safefree(pointer)
```

These three macros are used to change a memory buffer size or to free a piece of memory no longer needed. The arguments to Renew and Renewc match those of New and Newc with the exception of not needing the "magic cookie" argument.

28.5.3 Moving

```
    Move(source, dest, number, type);
    Copy(source, dest, number, type);
    Zero(dest, number, type);
```

These three macros are used to move, copy, or zero out previously allocated memory. The source and dest arguments point to the source and destination starting points. Perl will move, copy, or zero out number instances of the size of the type data structure (using the sizeof function).

28.6 PerlIO

The most recent development releases of Perl have been experimenting with removing Perl's dependency on the "normal" standard I/O suite and allowing other stdio implementations to be used. This involves creating a new abstraction layer that then calls whichever implementation of stdio Perl was compiled with. All XSUBs should now use the functions in the PerlIO abstraction layer and not make any assumptions about what kind of stdio is being used.

For a complete description of the PerlIO abstraction, consult Section 2.1 [perlapio NAME], page 9.

28.7 Compiled code

28.7.1 Code tree

Here we describe the internal form your code is converted to by Perl. Start with a simple example:

```
  $a = $b + $c;
```

This is converted to a tree similar to this one:

```
             assign-to
           /           \
          +             $a
        /   \
      $b     $c
```

(but slightly more complicated). This tree reflects the way Perl parsed your code, but has nothing to do with the execution order. There is an additional "thread" going through the nodes of the tree which shows the order of execution of the nodes. In our simplified example above it looks like:

```
$b ---> $c ---> + ---> $a ---> assign-to
```

But with the actual compile tree for `$a = $b + $c` it is different: some nodes *optimized away*. As a corollary, though the actual tree contains more nodes than our simplified example, the execution order is the same as in our example.

28.7.2 Examining the tree

If you have your perl compiled for debugging (usually done with `-DDEBUGGING` on the `Configure` command line), you may examine the compiled tree by specifying `-Dx` on the Perl command line. The output takes several lines per node, and for `$b+$c` it looks like this:

```
5           TYPE = add  ===> 6
            TARG = 1
            FLAGS = (SCALAR,KIDS)
            {
                TYPE = null  ===> (4)
                  (was rv2sv)
                FLAGS = (SCALAR,KIDS)
                {
3                   TYPE = gvsv  ===> 4
                    FLAGS = (SCALAR)
                    GV = main::b
                }
            }
            {
                TYPE = null  ===> (5)
                  (was rv2sv)
                FLAGS = (SCALAR,KIDS)
                {
4                   TYPE = gvsv  ===> 5
                    FLAGS = (SCALAR)
                    GV = main::c
                }
            }
```

This tree has 5 nodes (one per `TYPE` specifier), only 3 of them are not optimized away (one per number in the left column). The immediate children of the given node correspond to `{}` pairs on the same level of indentation, thus this listing corresponds to the tree:

```
    add
  /     \
null    null
 |       |
gvsv    gvsv
```

The execution order is indicated by ===> marks, thus it is 3 4 5 6 (node 6 is not included into above listing), i.e., gvsv gvsv add whatever.

Each of these nodes represents an op, a fundamental operation inside the Perl core. The code which implements each operation can be found in the pp*.c files; the function which implements the op with type gvsv is pp_gvsv, and so on. As the tree above shows, different ops have different numbers of children: add is a binary operator, as one would expect, and so has two children. To accommodate the various different numbers of children, there are various types of op data structure, and they link together in different ways.

The simplest type of op structure is OP: this has no children. Unary operators, UNOPs, have one child, and this is pointed to by the op_first field. Binary operators (BINOPs) have not only an op_first field but also an op_last field. The most complex type of op is a LISTOP, which has any number of children. In this case, the first child is pointed to by op_first and the last child by op_last. The children in between can be found by iteratively following the OpSIBLING pointer from the first child to the last (but see below).

There are also some other op types: a PMOP holds a regular expression, and has no children, and a LOOP may or may not have children. If the op_children field is non-zero, it behaves like a LISTOP. To complicate matters, if a UNOP is actually a null op after optimization (see Section 28.7.5 [Compile pass 2: context propagation], page 544) it will still have children in accordance with its former type.

Finally, there is a LOGOP, or logic op. Like a LISTOP, this has one or more children, but it doesn't have an op_last field: so you have to follow op_first and then the OpSIBLING chain itself to find the last child. Instead it has an op_other field, which is comparable to the op_next field described below, and represents an alternate execution path. Operators like and, or and ? are LOGOPs. Note that in general, op_other may not point to any of the direct children of the LOGOP.

Starting in version 5.21.2, perls built with the experimental define -DPERL_OP_PARENT add an extra boolean flag for each op, op_moresib. When not set, this indicates that this is the last op in an OpSIBLING chain. This frees up the op_sibling field on the last sibling to point back to the parent op. Under this build, that field is also renamed op_sibparent to reflect its joint role. The macro OpSIBLING(o) wraps this special behaviour, and always returns NULL on the last sibling. With this build the op_parent(o) function can be used to find the parent of any op. Thus for forward compatibility, you should always use the OpSIBLING(o) macro rather than accessing op_sibling directly.

Another way to examine the tree is to use a compiler back-end module, such as B-Concise.

28.7.3 Compile pass 1: check routines

The tree is created by the compiler while *yacc* code feeds it the constructions it recognizes. Since *yacc* works bottom-up, so does the first pass of perl compilation.

What makes this pass interesting for perl developers is that some optimization may be performed on this pass. This is optimization by so-called "check routines". The correspondence between node names and corresponding check routines is described in opcode.pl (do not forget to run make regen_headers if you modify this file).

A check routine is called when the node is fully constructed except for the execution-order thread. Since at this time there are no back-links to the currently constructed node, one can do most any operation to the top-level node, including freeing it and/or creating new nodes above/below it.

The check routine returns the node which should be inserted into the tree (if the top-level node was not modified, check routine returns its argument).

By convention, check routines have names `ck_*`. They are usually called from `new*OP` subroutines (or `convert`) (which in turn are called from `perly.y`).

28.7.4 Compile pass 1a: constant folding

Immediately after the check routine is called the returned node is checked for being compile-time executable. If it is (the value is judged to be constant) it is immediately executed, and a *constant* node with the "return value" of the corresponding subtree is substituted instead. The subtree is deleted.

If constant folding was not performed, the execution-order thread is created.

28.7.5 Compile pass 2: context propagation

When a context for a part of compile tree is known, it is propagated down through the tree. At this time the context can have 5 values (instead of 2 for runtime context): void, boolean, scalar, list, and lvalue. In contrast with the pass 1 this pass is processed from top to bottom: a node's context determines the context for its children.

Additional context-dependent optimizations are performed at this time. Since at this moment the compile tree contains back-references (via "thread" pointers), nodes cannot be free()d now. To allow optimized-away nodes at this stage, such nodes are null()ified instead of free()ing (i.e. their type is changed to OP_NULL).

28.7.6 Compile pass 3: peephole optimization

After the compile tree for a subroutine (or for an `eval` or a file) is created, an additional pass over the code is performed. This pass is neither top-down or bottom-up, but in the execution order (with additional complications for conditionals). Optimizations performed at this stage are subject to the same restrictions as in the pass 2.

Peephole optimizations are done by calling the function pointed to by the global variable `PL_peepp`. By default, `PL_peepp` just calls the function pointed to by the global variable `PL_rpeepp`. By default, that performs some basic op fixups and optimisations along the execution-order op chain, and recursively calls `PL_rpeepp` for each side chain of ops (resulting from conditionals). Extensions may provide additional optimisations or fixups, hooking into either the per-subroutine or recursive stage, like this:

```
static peep_t prev_peepp;
static void my_peep(pTHX_ OP *o)
{
    /* custom per-subroutine optimisation goes here */
    prev_peepp(aTHX_ o);
    /* custom per-subroutine optimisation may also go here */
}
BOOT:
    prev_peepp = PL_peepp;
    PL_peepp = my_peep;
```

```
static peep_t prev_rpeepp;
static void my_rpeep(pTHX_ OP *o)
{
    OP *orig_o = o;
    for(; o; o = o->op_next) {
        /* custom per-op optimisation goes here */
    }
    prev_rpeepp(aTHX_ orig_o);
}
BOOT:
    prev_rpeepp = PL_rpeepp;
    PL_rpeepp = my_rpeep;
```

28.7.7 Pluggable runops

The compile tree is executed in a runops function. There are two runops functions, in run.c and in dump.c. Perl_runops_debug is used with DEBUGGING and Perl_runops_standard is used otherwise. For fine control over the execution of the compile tree it is possible to provide your own runops function.

It's probably best to copy one of the existing runops functions and change it to suit your needs. Then, in the BOOT section of your XS file, add the line:

```
PL_runops = my_runops;
```

This function should be as efficient as possible to keep your programs running as fast as possible.

28.7.8 Compile-time scope hooks

As of perl 5.14 it is possible to hook into the compile-time lexical scope mechanism using Perl_blockhook_register. This is used like this:

```
STATIC void my_start_hook(pTHX_ int full);
STATIC BHK my_hooks;

BOOT:
    BhkENTRY_set(&my_hooks, bhk_start, my_start_hook);
    Perl_blockhook_register(aTHX_ &my_hooks);
```

This will arrange to have my_start_hook called at the start of compiling every lexical scope. The available hooks are:

void bhk_start(pTHX_ int full)

This is called just after starting a new lexical scope. Note that Perl code like

```
if ($x) { ... }
```

creates two scopes: the first starts at the (and has full == 1, the second starts at the { and has full == 0. Both end at the }, so calls to start and pre/post_end will match. Anything pushed onto the save stack by this hook will be popped just before the scope ends (between the pre_ and post_end hooks, in fact).

void bhk_pre_end(pTHX_ OP **o)

This is called at the end of a lexical scope, just before unwinding the stack. *o* is the root of the optree representing the scope; it is a double pointer so you can replace the OP if you need to.

void bhk_post_end(pTHX_ OP **o)

This is called at the end of a lexical scope, just after unwinding the stack. *o* is as above. Note that it is possible for calls to pre_ and post_end to nest, if there is something on the save stack that calls string eval.

void bhk_eval(pTHX_ OP *const o)

This is called just before starting to compile an eval STRING, do FILE, require or use, after the eval has been set up. *o* is the OP that requested the eval, and will normally be an OP_ENTEREVAL, OP_DOFILE or OP_REQUIRE.

Once you have your hook functions, you need a BHK structure to put them in. It's best to allocate it statically, since there is no way to free it once it's registered. The function pointers should be inserted into this structure using the BhkENTRY_set macro, which will also set flags indicating which entries are valid. If you do need to allocate your BHK dynamically for some reason, be sure to zero it before you start.

Once registered, there is no mechanism to switch these hooks off, so if that is necessary you will need to do this yourself. An entry in %^H is probably the best way, so the effect is lexically scoped; however it is also possible to use the BhkDISABLE and BhkENABLE macros to temporarily switch entries on and off. You should also be aware that generally speaking at least one scope will have opened before your extension is loaded, so you will see some pre/post_end pairs that didn't have a matching start.

28.8 Examining internal data structures with the dump functions

To aid debugging, the source file dump.c contains a number of functions which produce formatted output of internal data structures.

The most commonly used of these functions is Perl_sv_dump; it's used for dumping SVs, AVs, HVs, and CVs. The Devel::Peek module calls sv_dump to produce debugging output from Perl-space, so users of that module should already be familiar with its format.

Perl_op_dump can be used to dump an OP structure or any of its derivatives, and produces output similar to perl -Dx; in fact, Perl_dump_eval will dump the main root of the code being evaluated, exactly like -Dx.

Other useful functions are Perl_dump_sub, which turns a GV into an op tree, Perl_dump_packsubs which calls Perl_dump_sub on all the subroutines in a package like so: (Thankfully, these are all xsubs, so there is no op tree)

```
 (gdb) print Perl_dump_packsubs(PL_defstash)

 SUB attributes::bootstrap = (xsub 0x811fedc 0)

 SUB UNIVERSAL::can = (xsub 0x811f50c 0)

 SUB UNIVERSAL::isa = (xsub 0x811f304 0)
```

```
    SUB UNIVERSAL::VERSION = (xsub 0x811f7ac 0)

    SUB DynaLoader::boot_DynaLoader = (xsub 0x805b188 0)
```

and `Perl_dump_all`, which dumps all the subroutines in the stash and the op tree of the main root.

28.9 How multiple interpreters and concurrency are supported

28.9.1 Background and PERL_IMPLICIT_CONTEXT

The Perl interpreter can be regarded as a closed box: it has an API for feeding it code or otherwise making it do things, but it also has functions for its own use. This smells a lot like an object, and there are ways for you to build Perl so that you can have multiple interpreters, with one interpreter represented either as a C structure, or inside a thread-specific structure. These structures contain all the context, the state of that interpreter.

One macro controls the major Perl build flavor: MULTIPLICITY. The MULTIPLICITY build has a C structure that packages all the interpreter state. With multiplicity-enabled perls, PERL_IMPLICIT_CONTEXT is also normally defined, and enables the support for passing in a "hidden" first argument that represents all three data structures. MULTIPLICITY makes multi-threaded perls possible (with the ithreads threading model, related to the macro USE_ITHREADS.)

Two other "encapsulation" macros are the PERL_GLOBAL_STRUCT and PERL_GLOBAL_STRUCT_PRIVATE (the latter turns on the former, and the former turns on MULTIPLICITY.) The PERL_GLOBAL_STRUCT causes all the internal variables of Perl to be wrapped inside a single global struct, struct perl_vars, accessible as (globals) &PL_Vars or PL_VarsPtr or the function Perl_GetVars(). The PERL_GLOBAL_STRUCT_PRIVATE goes one step further, there is still a single struct (allocated in main() either from heap or from stack) but there are no global data symbols pointing to it. In either case the global struct should be initialized as the very first thing in main() using Perl_init_global_struct() and correspondingly tear it down after perl_free() using Perl_free_global_struct(), please see `miniperlmain.c` for usage details. You may also need to use `dVAR` in your coding to "declare the global variables" when you are using them. dTHX does this for you automatically.

To see whether you have non-const data you can use a BSD (or GNU) compatible `nm`:

```
nm libperl.a | grep -v ' [TURtr] '
```

If this displays any `D` or `d` symbols (or possibly `C` or `c`), you have non-const data. The symbols the `grep` removed are as follows: `Tt` are *text*, or code, the `Rr` are *read-only* (const) data, and the `U` is <undefined>, external symbols referred to.

The test `t/porting/libperl.t` does this kind of symbol sanity checking on `libperl.a`.

For backward compatibility reasons defining just PERL_GLOBAL_STRUCT doesn't actually hide all symbols inside a big global struct: some PerlIO_xxx vtables are left visible. The PERL_GLOBAL_STRUCT_PRIVATE then hides everything (see how the PERLIO_FUNCS_DECL is used).

All this obviously requires a way for the Perl internal functions to be either subroutines taking some kind of structure as the first argument, or subroutines taking nothing as the first argument. To enable these two very different ways of building the interpreter, the Perl source (as it does in so many other situations) makes heavy use of macros and subroutine naming conventions.

First problem: deciding which functions will be public API functions and which will be private. All functions whose names begin `S_` are private (think "S" for "secret" or "static"). All other functions begin with "Perl_", but just because a function begins with "Perl_" does not mean it is part of the API. (See Section 28.10 [Internal Functions], page 552.) The easiest way to be **sure** a function is part of the API is to find its entry in `perlapi`. If it exists in `perlapi`, it's part of the API. If it doesn't, and you think it should be (i.e., you need it for your extension), send mail via `perlbug` explaining why you think it should be.

Second problem: there must be a syntax so that the same subroutine declarations and calls can pass a structure as their first argument, or pass nothing. To solve this, the subroutines are named and declared in a particular way. Here's a typical start of a static function used within the Perl guts:

```
STATIC void
S_incline(pTHX_ char *s)
```

STATIC becomes "static" in C, and may be #define'd to nothing in some configurations in the future.

A public function (i.e. part of the internal API, but not necessarily sanctioned for use in extensions) begins like this:

```
void
Perl_sv_setiv(pTHX_ SV* dsv, IV num)
```

`pTHX_` is one of a number of macros (in `perl.h`) that hide the details of the interpreter's context. THX stands for "thread", "this", or "thingy", as the case may be. (And no, George Lucas is not involved. :-) The first character could be 'p' for a **p**rototype, 'a' for **a**rgument, or 'd' for **d**eclaration, so we have `pTHX`, `aTHX` and `dTHX`, and their variants.

When Perl is built without options that set PERL_IMPLICIT_CONTEXT, there is no first argument containing the interpreter's context. The trailing underscore in the pTHX_ macro indicates that the macro expansion needs a comma after the context argument because other arguments follow it. If PERL_IMPLICIT_CONTEXT is not defined, pTHX_ will be ignored, and the subroutine is not prototyped to take the extra argument. The form of the macro without the trailing underscore is used when there are no additional explicit arguments.

When a core function calls another, it must pass the context. This is normally hidden via macros. Consider `sv_setiv`. It expands into something like this:

```
#ifdef PERL_IMPLICIT_CONTEXT
  #define sv_setiv(a,b)      Perl_sv_setiv(aTHX_ a, b)
  /* can't do this for vararg functions, see below */
#else
  #define sv_setiv           Perl_sv_setiv
#endif
```

This works well, and means that XS authors can gleefully write:

```
sv_setiv(foo, bar);
```

and still have it work under all the modes Perl could have been compiled with.

This doesn't work so cleanly for varargs functions, though, as macros imply that the number of arguments is known in advance. Instead we either need to spell them out fully, passing `aTHX_` as the first argument (the Perl core tends to do this with functions like Perl_warner), or use a context-free version.

The context-free version of Perl_warner is called Perl_warner_nocontext, and does not take the extra argument. Instead it does dTHX; to get the context from thread-local storage. We `#define warner Perl_warner_nocontext` so that extensions get source compatibility at the expense of performance. (Passing an arg is cheaper than grabbing it from thread-local storage.)

You can ignore [pad]THXx when browsing the Perl headers/sources. Those are strictly for use within the core. Extensions and embedders need only be aware of [pad]THX.

28.9.2 So what happened to dTHR?

`dTHR` was introduced in perl 5.005 to support the older thread model. The older thread model now uses the `THX` mechanism to pass context pointers around, so `dTHR` is not useful any more. Perl 5.6.0 and later still have it for backward source compatibility, but it is defined to be a no-op.

28.9.3 How do I use all this in extensions?

When Perl is built with PERL_IMPLICIT_CONTEXT, extensions that call any functions in the Perl API will need to pass the initial context argument somehow. The kicker is that you will need to write it in such a way that the extension still compiles when Perl hasn't been built with PERL_IMPLICIT_CONTEXT enabled.

There are three ways to do this. First, the easy but inefficient way, which is also the default, in order to maintain source compatibility with extensions: whenever `XSUB.h` is #included, it redefines the aTHX and aTHX_ macros to call a function that will return the context. Thus, something like:

```
sv_setiv(sv, num);
```

in your extension will translate to this when PERL_IMPLICIT_CONTEXT is in effect:

```
Perl_sv_setiv(Perl_get_context(), sv, num);
```

or to this otherwise:

```
Perl_sv_setiv(sv, num);
```

You don't have to do anything new in your extension to get this; since the Perl library provides Perl_get_context(), it will all just work.

The second, more efficient way is to use the following template for your Foo.xs:

```
#define PERL_NO_GET_CONTEXT     /* we want efficiency */
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

STATIC void my_private_function(int arg1, int arg2);
STATIC void
my_private_function(int arg1, int arg2)
{
    dTHX;       /* fetch context */
    ... call many Perl API functions ...
}

[... etc ...]

MODULE = Foo            PACKAGE = Foo

/* typical XSUB */

void
my_xsub(arg)
        int arg
    CODE:
        my_private_function(arg, 10);
```

Note that the only two changes from the normal way of writing an extension is the addition of a `#define PERL_NO_GET_CONTEXT` before including the Perl headers, followed by a `dTHX;` declaration at the start of every function that will call the Perl API. (You'll know which functions need this, because the C compiler will complain that there's an undeclared identifier in those functions.) No changes are needed for the XSUBs themselves, because the XS() macro is correctly defined to pass in the implicit context if needed.

The third, even more efficient way is to ape how it is done within the Perl guts:

```
#define PERL_NO_GET_CONTEXT     /* we want efficiency */
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

/* pTHX_ only needed for functions that call Perl API */
STATIC void my_private_function(pTHX_ int arg1, int arg2);

STATIC void
my_private_function(pTHX_ int arg1, int arg2)
{
    /* dTHX; not needed here, because THX is an argument */
    ... call Perl API functions ...
}

[... etc ...]

MODULE = Foo            PACKAGE = Foo

/* typical XSUB */
```

```
void
my_xsub(arg)
        int arg
    CODE:
        my_private_function(aTHX_ arg, 10);
```

This implementation never has to fetch the context using a function call, since it is always passed as an extra argument. Depending on your needs for simplicity or efficiency, you may mix the previous two approaches freely.

Never add a comma after `pTHX` yourself–always use the form of the macro with the underscore for functions that take explicit arguments, or the form without the argument for functions with no explicit arguments.

If one is compiling Perl with the `-DPERL_GLOBAL_STRUCT` the `dVAR` definition is needed if the Perl global variables (see `perlvars.h` or `globvar.sym`) are accessed in the function and `dTHX` is not used (the `dTHX` includes the `dVAR` if necessary). One notices the need for `dVAR` only with the said compile-time define, because otherwise the Perl global variables are visible as-is.

28.9.4 Should I do anything special if I call perl from multiple threads?

If you create interpreters in one thread and then proceed to call them in another, you need to make sure perl's own Thread Local Storage (TLS) slot is initialized correctly in each of those threads.

The `perl_alloc` and `perl_clone` API functions will automatically set the TLS slot to the interpreter they created, so that there is no need to do anything special if the interpreter is always accessed in the same thread that created it, and that thread did not create or call any other interpreters afterwards. If that is not the case, you have to set the TLS slot of the thread before calling any functions in the Perl API on that particular interpreter. This is done by calling the `PERL_SET_CONTEXT` macro in that thread as the first thing you do:

```
/* do this before doing anything else with some_perl */
PERL_SET_CONTEXT(some_perl);

... other Perl API calls on some_perl go here ...
```

28.9.5 Future Plans and PERL_IMPLICIT_SYS

Just as PERL_IMPLICIT_CONTEXT provides a way to bundle up everything that the interpreter knows about itself and pass it around, so too are there plans to allow the interpreter to bundle up everything it knows about the environment it's running on. This is enabled with the PERL_IMPLICIT_SYS macro. Currently it only works with USE_ITHREADS on Windows.

This allows the ability to provide an extra pointer (called the "host" environment) for all the system calls. This makes it possible for all the system stuff to maintain their own state, broken down into seven C structures. These are thin wrappers around the usual system calls (see `win32/perllib.c`) for the default perl executable, but for a more ambitious host (like the one that would do fork() emulation) all the extra work needed to pretend that different interpreters are actually different "processes", would be done here.

The Perl engine/interpreter and the host are orthogonal entities. There could be one or more interpreters in a process, and one or more "hosts", with free association between them.

28.10 Internal Functions

All of Perl's internal functions which will be exposed to the outside world are prefixed by `Perl_` so that they will not conflict with XS functions or functions used in a program in which Perl is embedded. Similarly, all global variables begin with `PL_`. (By convention, static functions start with `S_`.)

Inside the Perl core (`PERL_CORE` defined), you can get at the functions either with or without the `Perl_` prefix, thanks to a bunch of defines that live in `embed.h`. Note that extension code should *not* set `PERL_CORE`; this exposes the full perl internals, and is likely to cause breakage of the XS in each new perl release.

The file `embed.h` is generated automatically from `embed.pl` and `embed.fnc`. `embed.pl` also creates the prototyping header files for the internal functions, generates the documentation and a lot of other bits and pieces. It's important that when you add a new function to the core or change an existing one, you change the data in the table in `embed.fnc` as well. Here's a sample entry from that table:

```
    Apd |SV**   |av_fetch   |AV* ar|I32 key|I32 lval
```

The second column is the return type, the third column the name. Columns after that are the arguments. The first column is a set of flags:

A

This function is a part of the public API. All such functions should also have 'd', very few do not.

p

This function has a `Perl_` prefix; i.e. it is defined as `Perl_av_fetch`.

d

This function has documentation using the `apidoc` feature which we'll look at in a second. Some functions have 'd' but not 'A'; docs are good.

Other available flags are:

s

This is a static function and is defined as `STATIC S_whatever`, and usually called within the sources as `whatever(...)`.

n

This does not need an interpreter context, so the definition has no `pTHX`, and it follows that callers don't use `aTHX`. (See Section 28.9.1 [Background and PERL_IMPLICIT_CONTEXT], page 547.)

r

This function never returns; `croak`, `exit` and friends.

f

This function takes a variable number of arguments, `printf` style. The argument list should end with `...`, like this:

```
Afprd   |void   |croak          |const char* pat|...
```

M

This function is part of the experimental development API, and may change or disappear without notice.

o

This function should not have a compatibility macro to define, say, `Perl_parse` to `parse`. It must be called as `Perl_parse`.

x

This function isn't exported out of the Perl core.

m

This is implemented as a macro.

X

This function is explicitly exported.

E

This function is visible to extensions included in the Perl core.

b

Binary backward compatibility; this function is a macro but also has a `Perl_` implementation (which is exported).

others

See the comments at the top of `embed.fnc` for others.

If you edit `embed.pl` or `embed.fnc`, you will need to run `make regen_headers` to force a rebuild of `embed.h` and other auto-generated files.

28.10.1 Formatted Printing of IVs, UVs, and NVs

If you are printing IVs, UVs, or NVS instead of the stdio(3) style formatting codes like `%d`, `%ld`, `%f`, you should use the following macros for portability

```
IVdf            IV in decimal
UVuf            UV in decimal
UVof            UV in octal
UVxf            UV in hexadecimal
NVef            NV %e-like
NVff            NV %f-like
NVgf            NV %g-like
```

These will take care of 64-bit integers and long doubles. For example:

```
printf("IV is %"IVdf"\n", iv);
```

The IVdf will expand to whatever is the correct format for the IVs.

Note that there are different "long doubles": Perl will use whatever the compiler has.

If you are printing addresses of pointers, use UVxf combined with PTR2UV(), do not use %lx or %p.

28.10.2 Pointer-To-Integer and Integer-To-Pointer

Because pointer size does not necessarily equal integer size, use the follow macros to do it right.

```
PTR2UV(pointer)
PTR2IV(pointer)
PTR2NV(pointer)
INT2PTR(pointertotype, integer)
```

For example:

```
IV  iv = ...;
SV *sv = INT2PTR(SV*, iv);
```

and

```
AV *av = ...;
UV  uv = PTR2UV(av);
```

28.10.3 Exception Handling

There are a couple of macros to do very basic exception handling in XS modules. You have to define `NO_XSLOCKS` before including `XSUB.h` to be able to use these macros:

```
#define NO_XSLOCKS
#include "XSUB.h"
```

You can use these macros if you call code that may croak, but you need to do some cleanup before giving control back to Perl. For example:

```
dXCPT;    /* set up necessary variables */

XCPT_TRY_START {
  code_that_may_croak();
} XCPT_TRY_END

XCPT_CATCH
{
  /* do cleanup here */
  XCPT_RETHROW;
}
```

Note that you always have to rethrow an exception that has been caught. Using these macros, it is not possible to just catch the exception and ignore it. If you have to ignore the exception, you have to use the `call_*` function.

The advantage of using the above macros is that you don't have to setup an extra function for `call_*`, and that using these macros is faster than using `call_*`.

28.10.4 Source Documentation

There's an effort going on to document the internal functions and automatically produce reference manuals from them – `perlapi` is one such manual which details all the functions which are available to XS writers. `perlintern` is the autogenerated manual for the functions which are not part of the API and are supposedly for internal use only.

Source documentation is created by putting POD comments into the C source, like this:

```
/*
=for apidoc sv_setiv

Copies an integer into the given SV.  Does not handle 'set' magic.  See
C<sv_setiv_mg>.

=cut
*/
```

Please try and supply some documentation if you add functions to the Perl core.

28.10.5 Backwards compatibility

The Perl API changes over time. New functions are added or the interfaces of existing functions are changed. The `Devel::PPPort` module tries to provide compatibility code for some of these changes, so XS writers don't have to code it themselves when supporting multiple versions of Perl.

`Devel::PPPort` generates a C header file `ppport.h` that can also be run as a Perl script. To generate `ppport.h`, run:

```
perl -MDevel::PPPort -eDevel::PPPort::WriteFile
```

Besides checking existing XS code, the script can also be used to retrieve compatibility information for various API calls using the `--api-info` command line switch. For example:

```
% perl ppport.h --api-info=sv_magicext
```

For details, see `perldoc ppport.h`.

28.11 Unicode Support

Perl 5.6.0 introduced Unicode support. It's important for porters and XS writers to understand this support and make sure that the code they write does not corrupt Unicode data.

28.11.1 What is Unicode, anyway?

In the olden, less enlightened times, we all used to use ASCII. Most of us did, anyway. The big problem with ASCII is that it's American. Well, no, that's not actually the problem; the problem is that it's not particularly useful for people who don't use the Roman alphabet. What used to happen was that particular languages would stick their own alphabet in the upper range of the sequence, between 128 and 255. Of course, we then ended up with plenty of variants that weren't quite ASCII, and the whole point of it being a standard was lost.

Worse still, if you've got a language like Chinese or Japanese that has hundreds or thousands of characters, then you really can't fit them into a mere 256, so they had to forget about ASCII altogether, and build their own systems using pairs of numbers to refer to one character.

To fix this, some people formed Unicode, Inc. and produced a new character set containing all the characters you can possibly think of and more. There are several ways of representing these characters, and the one Perl uses is called UTF-8. UTF-8 uses a variable number of bytes to represent a character. You can learn more about Unicode and Perl's Unicode model in Section 81.1 [perlunicode NAME], page 1317.

(On EBCDIC platforms, Perl uses instead UTF-EBCDIC, which is a form of UTF-8 adapted for EBCDIC platforms. Below, we just talk about UTF-8. UTF-EBCDIC is like UTF-8, but the details are different. The macros hide the differences from you, just remember that the particular numbers and bit patterns presented below will differ in UTF-EBCDIC.)

28.11.2 How can I recognise a UTF-8 string?

You can't. This is because UTF-8 data is stored in bytes just like non-UTF-8 data. The Unicode character 200, (`0xC8` for you hex types) capital E with a grave accent, is represented by the two bytes `v196.172`. Unfortunately, the non-Unicode string `chr(196).chr(172)` has that byte sequence as well. So you can't tell just by looking – this is what makes Unicode input an interesting problem.

In general, you either have to know what you're dealing with, or you have to guess. The API function `is_utf8_string` can help; it'll tell you if a string contains only valid UTF-8 characters, and the chances of a non-UTF-8 string looking like valid UTF-8 become very small very quickly with increasing string length. On a character-by-character basis, `isUTF8_CHAR` will tell you whether the current character in a string is valid UTF-8.

28.11.3 How does UTF-8 represent Unicode characters?

As mentioned above, UTF-8 uses a variable number of bytes to store a character. Characters with values 0...127 are stored in one byte, just like good ol' ASCII. Character 128 is stored as `v194.128`; this continues up to character 191, which is `v194.191`. Now we've run out of bits (191 is binary `10111111`) so we move on; character 192 is `v195.128`. And so it goes on, moving to three bytes at character 2048. Section 81.2.10 [perlunicode Unicode Encodings], page 1334 has pictures of how this works.

Assuming you know you're dealing with a UTF-8 string, you can find out how long the first character in it is with the `UTF8SKIP` macro:

```
char *utf = "\305\233\340\240\201";
I32 len;

len = UTF8SKIP(utf); /* len is 2 here */
utf += len;
len = UTF8SKIP(utf); /* len is 3 here */
```

Another way to skip over characters in a UTF-8 string is to use `utf8_hop`, which takes a string and a number of characters to skip over. You're on your own about bounds checking, though, so don't use it lightly.

All bytes in a multi-byte UTF-8 character will have the high bit set, so you can test if you need to do something special with this character like this (the `UTF8_IS_INVARIANT()` is a macro that tests whether the byte is encoded as a single byte even in UTF-8):

```
U8 *utf;
U8 *utf_end; /* 1 beyond buffer pointed to by utf */
UV uv;      /* Note: a UV, not a U8, not a char */
```

```
STRLEN len; /* length of character in bytes */

if (!UTF8_IS_INVARIANT(*utf))
    /* Must treat this as UTF-8 */
    uv = utf8_to_uvchr_buf(utf, utf_end, &len);
else
    /* OK to treat this character as a byte */
    uv = *utf;
```

You can also see in that example that we use `utf8_to_uvchr_buf` to get the value of the character; the inverse function `uvchr_to_utf8` is available for putting a UV into UTF-8:

```
if (!UVCHR_IS_INVARIANT(uv))
    /* Must treat this as UTF8 */
    utf8 = uvchr_to_utf8(utf8, uv);
else
    /* OK to treat this character as a byte */
    *utf8++ = uv;
```

You **must** convert characters to UVs using the above functions if you're ever in a situation where you have to match UTF-8 and non-UTF-8 characters. You may not skip over UTF-8 characters in this case. If you do this, you'll lose the ability to match hi-bit non-UTF-8 characters; for instance, if your UTF-8 string contains `v196.172`, and you skip that character, you can never match a `chr(200)` in a non-UTF-8 string. So don't do that!

(Note that we don't have to test for invariant characters in the examples above. The functions work on any well-formed UTF-8 input. It's just that its faster to avoid the function overhead when it's not needed.)

28.11.4 How does Perl store UTF-8 strings?

Currently, Perl deals with UTF-8 strings and non-UTF-8 strings slightly differently. A flag in the SV, `SVf_UTF8`, indicates that the string is internally encoded as UTF-8. Without it, the byte value is the codepoint number and vice versa. This flag is only meaningful if the SV is `SvPOK` or immediately after stringification via `SvPV` or a similar macro. You can check and manipulate this flag with the following macros:

```
SvUTF8(sv)
SvUTF8_on(sv)
SvUTF8_off(sv)
```

This flag has an important effect on Perl's treatment of the string: if UTF-8 data is not properly distinguished, regular expressions, `length`, `substr` and other string handling operations will have undesirable (wrong) results.

The problem comes when you have, for instance, a string that isn't flagged as UTF-8, and contains a byte sequence that could be UTF-8 – especially when combining non-UTF-8 and UTF-8 strings.

Never forget that the `SVf_UTF8` flag is separate from the PV value; you need to be sure you don't accidentally knock it off while you're manipulating SVs. More specifically, you cannot expect to do this:

```
SV *sv;
SV *nsv;
```

```
STRLEN len;
char *p;

p = SvPV(sv, len);
frobnicate(p);
nsv = newSVpvn(p, len);
```

The `char*` string does not tell you the whole story, and you can't copy or reconstruct an SV just by copying the string value. Check if the old SV has the UTF8 flag set (*after* the `SvPV` call), and act accordingly:

```
p = SvPV(sv, len);
is_utf8 = SvUTF8(sv);
frobnicate(p, is_utf8);
nsv = newSVpvn(p, len);
if (is_utf8)
    SvUTF8_on(nsv);
```

In the above, your `frobnicate` function has been changed to be made aware of whether or not it's dealing with UTF-8 data, so that it can handle the string appropriately.

Since just passing an SV to an XS function and copying the data of the SV is not enough to copy the UTF8 flags, even less right is just passing a `char *` to an XS function.

For full generality, use the Section "DO_UTF8" in `perlapi` macro to see if the string in an SV is to be *treated* as UTF-8. This takes into account if the call to the XS function is being made from within the scope of `bytes`. If so, the underlying bytes that comprise the UTF-8 string are to be exposed, rather than the character they represent. But this pragma should only really be used for debugging and perhaps low-level testing at the byte level. Hence most XS code need not concern itself with this, but various areas of the perl core do need to support it.

And this isn't the whole story. Starting in Perl v5.12, strings that aren't encoded in UTF-8 may also be treated as Unicode under various conditions (see Section 81.2.3 [perlunicode ASCII Rules versus Unicode Rules], page 1319). This is only really a problem for characters whose ordinals are between 128 and 255, and their behavior varies under ASCII versus Unicode rules in ways that your code cares about (see Section 81.2.17 [perlunicode The "Unicode Bug"], page 1340). There is no published API for dealing with this, as it is subject to change, but you can look at the code for `pp_lc` in `pp.c` for an example as to how it's currently done.

28.11.5 How do I convert a string to UTF-8?

If you're mixing UTF-8 and non-UTF-8 strings, it is necessary to upgrade the non-UTF-8 strings to UTF-8. If you've got an SV, the easiest way to do this is:

```
sv_utf8_upgrade(sv);
```

However, you must not do this, for example:

```
if (!SvUTF8(left))
    sv_utf8_upgrade(left);
```

If you do this in a binary operator, you will actually change one of the strings that came into the operator, and, while it shouldn't be noticeable by the end user, it can cause problems in deficient code.

Instead, `bytes_to_utf8` will give you a UTF-8-encoded **copy** of its string argument. This is useful for having the data available for comparisons and so on, without harming the original SV. There's also `utf8_to_bytes` to go the other way, but naturally, this will fail if the string contains any characters above 255 that can't be represented in a single byte.

28.11.6 How do I compare strings?

Section "sv_cmp" in `perlapi` and Section "sv_cmp_flags" in `perlapi` do a lexigraphic comparison of two SV's, and handle UTF-8ness properly. Note, however, that Unicode specifies a much fancier mechanism for collation, available via the `Unicode-Collate` module.

To just compare two strings for equality/non-equality, you can just use Section "memEQ" in `perlapi` and Section "memEQ" in `perlapi` as usual, except the strings must be both UTF-8 or not UTF-8 encoded.

To compare two strings case-insensitively, use Section "foldEQ_utf8" in `perlapi` (the strings don't have to have the same UTF-8ness).

28.11.7 Is there anything else I need to know?

Not really. Just remember these things:

- There's no way to tell if a `char *` or `U8 *` string is UTF-8 or not. But you can tell if an SV is to be treated as UTF-8 by calling `DO_UTF8` on it, after stringifying it with `SvPV` or a similar macro. And, you can tell if SV is actually UTF-8 (even if it is not to be treated as such) by looking at its `SvUTF8` flag (again after stringifying it). Don't forget to set the flag if something should be UTF-8. Treat the flag as part of the PV, even though it's not – if you pass on the PV to somewhere, pass on the flag too.
- If a string is UTF-8, **always** use `utf8_to_uvchr_buf` to get at the value, unless `UTF8_IS_INVARIANT(*s)` in which case you can use `*s`.
- When writing a character UV to a UTF-8 string, **always** use `uvchr_to_utf8`, unless `UVCHR_IS_INVARIANT(uv))` in which case you can use `*s = uv`.
- Mixing UTF-8 and non-UTF-8 strings is tricky. Use `bytes_to_utf8` to get a new string which is UTF-8 encoded, and then combine them.

28.12 Custom Operators

Custom operator support is an experimental feature that allows you to define your own ops. This is primarily to allow the building of interpreters for other languages in the Perl core, but it also allows optimizations through the creation of "macro-ops" (ops which perform the functions of multiple ops which are usually executed together, such as `gvsv, gvsv, add`.)

This feature is implemented as a new op type, `OP_CUSTOM`. The Perl core does not "know" anything special about this op type, and so it will not be involved in any optimizations. This also means that you can define your custom ops to be any op structure – unary, binary, list and so on – you like.

It's important to know what custom operators won't do for you. They won't let you add new syntax to Perl, directly. They won't even let you add new keywords, directly. In fact, they won't change the way Perl compiles a program at all. You have to do those changes yourself, after Perl has compiled the program. You do this either by manipulating the op tree using a `CHECK` block and the `B::Generate` module, or by adding a custom peephole optimizer with the `optimize` module.

When you do this, you replace ordinary Perl ops with custom ops by creating ops with the type `OP_CUSTOM` and the `op_ppaddr` of your own PP function. This should be defined in XS code, and should look like the PP ops in `pp_*.c`. You are responsible for ensuring that your op takes the appropriate number of values from the stack, and you are responsible for adding stack marks if necessary.

You should also "register" your op with the Perl interpreter so that it can produce sensible error and warning messages. Since it is possible to have multiple custom ops within the one "logical" op type `OP_CUSTOM`, Perl uses the value of `o->op_ppaddr` to determine which custom op it is dealing with. You should create an `XOP` structure for each ppaddr you use, set the properties of the custom op with `XopENTRY_set`, and register the structure against the ppaddr using `Perl_custom_op_register`. A trivial example might look like:

```
static XOP my_xop;
static OP *my_pp(pTHX);

BOOT:
    XopENTRY_set(&my_xop, xop_name, "myxop");
    XopENTRY_set(&my_xop, xop_desc, "Useless custom op");
    Perl_custom_op_register(aTHX_ my_pp, &my_xop);
```

The available fields in the structure are:

xop_name

A short name for your op. This will be included in some error messages, and will also be returned as `$op->name` by the `B` module, so it will appear in the output of module like `B-Concise`.

xop_desc

A short description of the function of the op.

xop_class

Which of the various `*OP` structures this op uses. This should be one of the `OA_*` constants from `op.h`, namely

OA_BASEOP
OA_UNOP
OA_BINOP
OA_LOGOP
OA_LISTOP
OA_PMOP
OA_SVOP
OA_PADOP
OA_PVOP_OR_SVOP

This should be interpreted as '`PVOP`' only. The `_OR_SVOP` is because the only core `PVOP`, `OP_TRANS`, can sometimes be a `SVOP` instead.

OA_LOOP
OA_COP

The other `OA_*` constants should not be used.

xop_peep

This member is of type `Perl_cpeep_t`, which expands to `void (*Perl_cpeep_t)(aTHX_ OP *o, OP *oldop)`. If it is set, this function will be called from `Perl_rpeep` when ops of this type are encountered by the peephole optimizer. *o* is the OP that needs optimizing; *oldop* is the previous OP optimized, whose `op_next` points to *o*.

`B::Generate` directly supports the creation of custom ops by name.

28.13 AUTHORS

Until May 1997, this document was maintained by Jeff Okamoto <okamoto@corp.hp.com>. It is now maintained as part of Perl itself by the Perl 5 Porters <perl5-porters@perl.org>.

With lots of help and suggestions from Dean Roehrich, Malcolm Beattie, Andreas Koenig, Paul Hudson, Ilya Zakharevich, Paul Marquess, Neil Bowers, Matthew Green, Tim Bunce, Spider Boardman, Ulrich Pfeifer, Stephen McCamant, and Gurusamy Sarathy.

28.14 SEE ALSO

`perlapi`, `perlintern`, `perlxs`, Section 20.1 [perlembed NAME], page 301

29 perlhack

29.1 NAME

perlhack - How to hack on Perl

29.2 DESCRIPTION

This document explains how Perl development works. It includes details about the Perl 5 Porters email list, the Perl repository, the Perlbug bug tracker, patch guidelines, and commentary on Perl development philosophy.

29.3 SUPER QUICK PATCH GUIDE

If you just want to submit a single small patch like a pod fix, a test for a bug, comment fixes, etc., it's easy! Here's how:

- Check out the source repository

  The perl source is in a git repository. You can clone the repository with the following command:

  ```
  % git clone git://perl5.git.perl.org/perl.git perl
  ```

- Ensure you're following the latest advice

  In case the advice in this guide has been updated recently, read the latest version directly from the perl source:

  ```
  % perldoc pod/perlhack.pod
  ```

- Make your change

  Hack, hack, hack. Keep in mind that Perl runs on many different platforms, with different operating systems that have different capabilities, different filesystem organizations, and even different character sets. Section 30.1 [perlhacktips NAME], page 578 gives advice on this.

- Test your change

  You can run all the tests with the following commands:

  ```
  % ./Configure -des -Dusedevel
  % make test
  ```

  Keep hacking until the tests pass.

- Commit your change

  Committing your work will save the change *on your local system*:

  ```
  % git commit -a -m 'Commit message goes here'
  ```

  Make sure the commit message describes your change in a single sentence. For example, "Fixed spelling errors in perlhack.pod".

- Send your change to perlbug

  The next step is to submit your patch to the Perl core ticket system via email.

  If your changes are in a single git commit, run the following commands to generate the patch file and attach it to your bug report:

```
% git format-patch -1
% ./perl -Ilib utils/perlbug -p 0001-*.patch
```

The perlbug program will ask you a few questions about your email address and the patch you're submitting. Once you've answered them it will submit your patch via email.

If your changes are in multiple commits, generate a patch file for each one and provide them to perlbug's -p option separated by commas:

```
% git format-patch -3
% ./perl -Ilib utils/perlbug -p 0001-fix1.patch,0002-fix2.patch,\
> 0003-fix3.patch
```

When prompted, pick a subject that summarizes your changes.

- Thank you

  The porters appreciate the time you spent helping to make Perl better. Thank you!

- Next time

  The next time you wish to make a patch, you need to start from the latest perl in a pristine state. Check you don't have any local changes or added files in your perl check-out which you wish to keep, then run these commands:

  ```
  % git pull
  % git reset --hard origin/blead
  % git clean -dxf
  ```

29.4 BUG REPORTING

If you want to report a bug in Perl, you must use the perlbug command line tool. This tool will ensure that your bug report includes all the relevant system and configuration information.

To browse existing Perl bugs and patches, you can use the web interface at http://rt.perl.org/.

Please check the archive of the perl5-porters list (see below) and/or the bug tracking system before submitting a bug report. Often, you'll find that the bug has been reported already.

You can log in to the bug tracking system and comment on existing bug reports. If you have additional information regarding an existing bug, please add it. This will help the porters fix the bug.

29.5 PERL 5 PORTERS

The perl5-porters (p5p) mailing list is where the Perl standard distribution is maintained and developed. The people who maintain Perl are also referred to as the "Perl 5 Porters", "p5p" or just the "porters".

A searchable archive of the list is available at http://markmail.org/search/?q=perl5-porters. There is also an archive at http://archive.develooper.com/perl5-porters@perl.org/.

29.5.1 perl-changes mailing list

The perl5-changes mailing list receives a copy of each patch that gets submitted to the maintenance and development branches of the perl repository. See `http://lists.perl.org/list/perl5-changes.html` for subscription and archive information.

29.5.2 #p5p on IRC

Many porters are also active on the `irc://irc.perl.org/#p5p` channel. Feel free to join the channel and ask questions about hacking on the Perl core.

29.6 GETTING THE PERL SOURCE

All of Perl's source code is kept centrally in a Git repository at *perl5.git.perl.org*. The repository contains many Perl revisions from Perl 1 onwards and all the revisions from Perforce, the previous version control system.

For much more detail on using git with the Perl repository, please see Section 26.1 [perlgit NAME], page 492.

29.6.1 Read access via Git

You will need a copy of Git for your computer. You can fetch a copy of the repository using the git protocol:

```
% git clone git://perl5.git.perl.org/perl.git perl
```

This clones the repository and makes a local copy in the `perl` directory.

If you cannot use the git protocol for firewall reasons, you can also clone via http, though this is much slower:

```
% git clone http://perl5.git.perl.org/perl.git perl
```

29.6.2 Read access via the web

You may access the repository over the web. This allows you to browse the tree, see recent commits, subscribe to RSS feeds for the changes, search for particular commits and more. You may access it at `http://perl5.git.perl.org/perl.git`. A mirror of the repository is found at `https://github.com/Perl/perl5`.

29.6.3 Read access via rsync

You can also choose to use rsync to get a copy of the current source tree for the bleadperl branch and all maintenance branches:

```
% rsync -avz rsync://perl5.git.perl.org/perl-current .
% rsync -avz rsync://perl5.git.perl.org/perl-5.12.x .
% rsync -avz rsync://perl5.git.perl.org/perl-5.10.x .
% rsync -avz rsync://perl5.git.perl.org/perl-5.8.x .
% rsync -avz rsync://perl5.git.perl.org/perl-5.6.x .
% rsync -avz rsync://perl5.git.perl.org/perl-5.005xx .
```

(Add the `--delete` option to remove leftover files.)

To get a full list of the available sync points:

```
% rsync perl5.git.perl.org::
```

29.6.4 Write access via git

If you have a commit bit, please see Section 26.1 [perlgit NAME], page 492 for more details on using git.

29.7 PATCHING PERL

If you're planning to do more extensive work than a single small fix, we encourage you to read the documentation below. This will help you focus your work and make your patches easier to incorporate into the Perl source.

29.7.1 Submitting patches

If you have a small patch to submit, please submit it via perlbug. You can also send email directly to perlbug@perl.org. Please note that messages sent to perlbug may be held in a moderation queue, so you won't receive a response immediately.

You'll know your submission has been processed when you receive an email from our ticket tracking system. This email will give you a ticket number. Once your patch has made it to the ticket tracking system, it will also be sent to the perl5-porters@perl.org list.

Patches are reviewed and discussed on the p5p list. Simple, uncontroversial patches will usually be applied without any discussion. When the patch is applied, the ticket will be updated and you will receive email. In addition, an email will be sent to the p5p list.

In other cases, the patch will need more work or discussion. That will happen on the p5p list.

You are encouraged to participate in the discussion and advocate for your patch. Sometimes your patch may get lost in the shuffle. It's appropriate to send a reminder email to p5p if no action has been taken in a month. Please remember that the Perl 5 developers are all volunteers, and be polite.

Changes are always applied directly to the main development branch, called "blead". Some patches may be backported to a maintenance branch. If you think your patch is appropriate for the maintenance branch (see Section 55.6 [perlpolicy MAINTENANCE BRANCHES], page 946), please explain why when you submit it.

29.7.2 Getting your patch accepted

If you are submitting a code patch there are several things that you can do to help the Perl 5 Porters accept your patch.

29.7.2.1 Patch style

If you used git to check out the Perl source, then using `git format-patch` will produce a patch in a style suitable for Perl. The `format-patch` command produces one patch file for each commit you made. If you prefer to send a single patch for all commits, you can use `git diff`.

```
% git checkout blead
% git pull
% git diff blead my-branch-name
```

This produces a patch based on the difference between blead and your current branch. It's important to make sure that blead is up to date before producing the diff, that's why we call `git pull` first.

We strongly recommend that you use git if possible. It will make your life easier, and ours as well.

However, if you're not using git, you can still produce a suitable patch. You'll need a pristine copy of the Perl source to diff against. The porters prefer unified diffs. Using GNU `diff`, you can produce a diff like this:

```
% diff -Npurd perl.pristine perl.mine
```

Make sure that you `make realclean` in your copy of Perl to remove any build artifacts, or you may get a confusing result.

29.7.2.2 Commit message

As you craft each patch you intend to submit to the Perl core, it's important to write a good commit message. This is especially important if your submission will consist of a series of commits.

The first line of the commit message should be a short description without a period. It should be no longer than the subject line of an email, 50 characters being a good rule of thumb.

A lot of Git tools (Gitweb, GitHub, git log –pretty=oneline, ...) will only display the first line (cut off at 50 characters) when presenting commit summaries.

The commit message should include a description of the problem that the patch corrects or new functionality that the patch adds.

As a general rule of thumb, your commit message should help a programmer who knows the Perl core quickly understand what you were trying to do, how you were trying to do it, and why the change matters to Perl.

- Why

  Your commit message should describe why the change you are making is important. When someone looks at your change in six months or six years, your intent should be clear.

  If you're deprecating a feature with the intent of later simplifying another bit of code, say so. If you're fixing a performance problem or adding a new feature to support some other bit of the core, mention that.

- What

  Your commit message should describe what part of the Perl core you're changing and what you expect your patch to do.

- How

  While it's not necessary for documentation changes, new tests or trivial patches, it's often worth explaining how your change works. Even if it's clear to you today, it may not be clear to a porter next month or next year.

A commit message isn't intended to take the place of comments in your code. Commit messages should describe the change you made, while code comments should describe the current state of the code.

If you've just implemented a new feature, complete with doc, tests and well-commented code, a brief commit message will often suffice. If, however, you've just changed a single character deep in the parser or lexer, you might need to write a small novel to ensure that future readers understand what you did and why you did it.

29.7.2.3 Comments, Comments, Comments

Be sure to adequately comment your code. While commenting every line is unnecessary, anything that takes advantage of side effects of operators, that creates changes that will be felt outside of the function being patched, or that others may find confusing should be documented. If you are going to err, it is better to err on the side of adding too many comments than too few.

The best comments explain *why* the code does what it does, not *what it does.*

29.7.2.4 Style

In general, please follow the particular style of the code you are patching.

In particular, follow these general guidelines for patching Perl sources:

- 8-wide tabs (no exceptions!)
- 4-wide indents for code, 2-wide indents for nested CPP #defines
- Try hard not to exceed 79-columns
- ANSI C prototypes
- Uncuddled elses and "K&R" style for indenting control constructs
- No C++ style (//) comments
- Mark places that need to be revisited with XXX (and revisit often!)
- Opening brace lines up with "if" when conditional spans multiple lines; should be at end-of-line otherwise
- In function definitions, name starts in column 0 (return value-type is on previous line)
- Single space after keywords that are followed by parens, no space between function name and following paren
- Avoid assignments in conditionals, but if they're unavoidable, use extra paren, e.g. "if (a && (b = c)) ..."
- "return foo;" rather than "return(foo);"
- "if (!foo) ..." rather than "if (foo == FALSE) ..." etc.
- Do not declare variables using "register". It may be counterproductive with modern compilers, and is deprecated in C++, under which the Perl source is regularly compiled.
- In-line functions that are in headers that are accessible to XS code need to be able to compile without warnings with commonly used extra compilation flags, such as gcc's `-Wswitch-default` which warns whenever a switch statement does not have a "default" case. The use of these extra flags is to catch potential problems in legal C code, and is often used by Perl aggregators, such as Linux distributors.

29.7.2.5 Test suite

If your patch changes code (rather than just changing documentation), you should also include one or more test cases which illustrate the bug you're fixing or validate the new functionality you're adding. In general, you should update an existing test file rather than create a new one.

Your test suite additions should generally follow these guidelines (courtesy of Gurusamy Sarathy <gsar@activestate.com>):

- Know what you're testing. Read the docs, and the source.
- Tend to fail, not succeed.
- Interpret results strictly.
- Use unrelated features (this will flush out bizarre interactions).
- Use non-standard idioms (otherwise you are not testing TIMTOWTDI).
- Avoid using hardcoded test numbers whenever possible (the EXPECTED/GOT found in t/op/tie.t is much more maintainable, and gives better failure reports).
- Give meaningful error messages when a test fails.
- Avoid using qx// and system() unless you are testing for them. If you do use them, make sure that you cover _all_ perl platforms.
- Unlink any temporary files you create.
- Promote unforeseen warnings to errors with $SIG{__WARN__}.
- Be sure to use the libraries and modules shipped with the version being tested, not those that were already installed.
- Add comments to the code explaining what you are testing for.
- Make updating the '1..42' string unnecessary. Or make sure that you update it.
- Test _all_ behaviors of a given operator, library, or function.

  Test all optional arguments.

  Test return values in various contexts (boolean, scalar, list, lvalue).

  Use both global and lexical variables.

  Don't forget the exceptional, pathological cases.

29.7.3 Patching a core module

This works just like patching anything else, with one extra consideration.

Modules in the `cpan/` directory of the source tree are maintained outside of the Perl core. When the author updates the module, the updates are simply copied into the core. See that module's documentation or its listing on `http://search.cpan.org/` for more information on reporting bugs and submitting patches.

In most cases, patches to modules in `cpan/` should be sent upstream and should not be applied to the Perl core individually. If a patch to a file in `cpan/` absolutely cannot wait for the fix to be made upstream, released to CPAN and copied to blead, you must add (or update) a `CUSTOMIZED` entry in the `"Porting/Maintainers.pl"` file to flag that a local modification has been made. See `"Porting/Maintainers.pl"` for more details.

In contrast, modules in the `dist/` directory are maintained in the core.

29.7.4 Updating perldelta

For changes significant enough to warrant a `pod/perldelta.pod` entry, the porters will greatly appreciate it if you submit a delta entry along with your actual change. Significant changes include, but are not limited to:

- Adding, deprecating, or removing core features
- Adding, deprecating, removing, or upgrading core or dual-life modules
- Adding new core tests

- Fixing security issues and user-visible bugs in the core
- Changes that might break existing code, either on the perl or C level
- Significant performance improvements
- Adding, removing, or significantly changing documentation in the `pod/` directory
- Important platform-specific changes

Please make sure you add the perldelta entry to the right section within `pod/perldelta.pod`. More information on how to write good perldelta entries is available in the `Style` section of `Porting/how_to_write_a_perldelta.pod`.

29.7.5 What makes for a good patch?

New features and extensions to the language can be contentious. There is no specific set of criteria which determine what features get added, but here are some questions to consider when developing a patch:

29.7.5.1 Does the concept match the general goals of Perl?

Our goals include, but are not limited to:

1. Keep it fast, simple, and useful.
2. Keep features/concepts as orthogonal as possible.
3. No arbitrary limits (platforms, data sizes, cultures).
4. Keep it open and exciting to use/patch/advocate Perl everywhere.
5. Either assimilate new technologies, or build bridges to them.

29.7.5.2 Where is the implementation?

All the talk in the world is useless without an implementation. In almost every case, the person or people who argue for a new feature will be expected to be the ones who implement it. Porters capable of coding new features have their own agendas, and are not available to implement your (possibly good) idea.

29.7.5.3 Backwards compatibility

It's a cardinal sin to break existing Perl programs. New warnings can be contentious–some say that a program that emits warnings is not broken, while others say it is. Adding keywords has the potential to break programs, changing the meaning of existing token sequences or functions might break programs.

The Perl 5 core includes mechanisms to help porters make backwards incompatible changes more compatible such as the `feature` and `deprecate` modules. Please use them when appropriate.

29.7.5.4 Could it be a module instead?

Perl 5 has extension mechanisms, modules and XS, specifically to avoid the need to keep changing the Perl interpreter. You can write modules that export functions, you can give those functions prototypes so they can be called like built-in functions, you can even write XS code to mess with the runtime data structures of the Perl interpreter if you want to implement really complicated things.

Whenever possible, new features should be prototyped in a CPAN module before they will be considered for the core.

29.7.5.5 Is the feature generic enough?

Is this something that only the submitter wants added to the language, or is it broadly useful? Sometimes, instead of adding a feature with a tight focus, the porters might decide to wait until someone implements the more generalized feature.

29.7.5.6 Does it potentially introduce new bugs?

Radical rewrites of large chunks of the Perl interpreter have the potential to introduce new bugs.

29.7.5.7 How big is it?

The smaller and more localized the change, the better. Similarly, a series of small patches is greatly preferred over a single large patch.

29.7.5.8 Does it preclude other desirable features?

A patch is likely to be rejected if it closes off future avenues of development. For instance, a patch that placed a true and final interpretation on prototypes is likely to be rejected because there are still options for the future of prototypes that haven't been addressed.

29.7.5.9 Is the implementation robust?

Good patches (tight code, complete, correct) stand more chance of going in. Sloppy or incorrect patches might be placed on the back burner until the pumpking has time to fix, or might be discarded altogether without further notice.

29.7.5.10 Is the implementation generic enough to be portable?

The worst patches make use of system-specific features. It's highly unlikely that non-portable additions to the Perl language will be accepted.

29.7.5.11 Is the implementation tested?

Patches which change behaviour (fixing bugs or introducing new features) must include regression tests to verify that everything works as expected.

Without tests provided by the original author, how can anyone else changing perl in the future be sure that they haven't unwittingly broken the behaviour the patch implements? And without tests, how can the patch's author be confident that his/her hard work put into the patch won't be accidentally thrown away by someone in the future?

29.7.5.12 Is there enough documentation?

Patches without documentation are probably ill-thought out or incomplete. No features can be added or changed without documentation, so submitting a patch for the appropriate pod docs as well as the source code is important.

29.7.5.13 Is there another way to do it?

Larry said "Although the Perl Slogan is *There's More Than One Way to Do It*, I hesitate to make 10 ways to do something". This is a tricky heuristic to navigate, though–one man's essential addition is another man's pointless cruft.

29.7.5.14 Does it create too much work?

Work for the pumpking, work for Perl programmers, work for module authors, ... Perl is supposed to be easy.

29.7.5.15 Patches speak louder than words

Working code is always preferred to pie-in-the-sky ideas. A patch to add a feature stands a much higher chance of making it to the language than does a random feature request, no matter how fervently argued the request might be. This ties into "Will it be useful?", as the fact that someone took the time to make the patch demonstrates a strong desire for the feature.

29.8 TESTING

The core uses the same testing style as the rest of Perl, a simple "ok/not ok" run through Test::Harness, but there are a few special considerations.

There are three ways to write a test in the core: `Test-More`, `t/test.pl` and ad hoc `print $test ? "ok 42\n" : "not ok 42\n"`. The decision of which to use depends on what part of the test suite you're working on. This is a measure to prevent a high-level failure (such as Config.pm breaking) from causing basic functionality tests to fail.

The `t/test.pl` library provides some of the features of `Test-More`, but avoids loading most modules and uses as few core features as possible.

If you write your own test, use the Test Anything Protocol (`http://testanything.org`).

- `t/base`, `t/comp` and `t/opbasic`

  Since we don't know if `require` works, or even subroutines, use ad hoc tests for these three. Step carefully to avoid using the feature being tested. Tests in `t/opbasic`, for instance, have been placed there rather than in `t/op` because they test functionality which `t/test.pl` presumes has already been demonstrated to work.
- `t/cmd`, `t/run`, `t/io` and `t/op`

  Now that basic require() and subroutines are tested, you can use the `t/test.pl` library.

  You can also use certain libraries like Config conditionally, but be sure to skip the test gracefully if it's not there.
- Everything else

  Now that the core of Perl is tested, `Test-More` can and should be used. You can also use the full suite of core modules in the tests.

When you say "make test", Perl uses the `t/TEST` program to run the test suite (except under Win32 where it uses `t/harness` instead). All tests are run from the `t/` directory, **not** the directory which contains the test. This causes some problems with the tests in `lib/`, so here's some opportunity for some patching.

You must be triply conscious of cross-platform concerns. This usually boils down to using `File-Spec`, avoiding things like `fork()` and `system()` unless absolutely necessary, and not assuming that a given character has a particular ordinal value (code point) or that its UTF-8 representation is composed of particular bytes.

There are several functions available to specify characters and code points portably in tests. The always-preloaded functions `utf8::unicode_to_native()` and its inverse `utf8::native_to_unicode()` take code points and translate appropriately. The file `t/charset_tools.pl` has several functions that can be useful. It has versions of the previous two functions that take strings as inputs – not single numeric code points: `uni_to_native()` and `native_to_uni()`. If you must look at the individual bytes comprising a UTF-8 encoded string, `byte_utf8a_to_utf8n()` takes as input a string of those bytes encoded for an ASCII platform, and returns the equivalent string in the native platform. For example, `byte_utf8a_to_utf8n("\xC2\xA0")` returns the byte sequence on the current platform that form the UTF-8 for `U+00A0`, since `"\xC2\xA0"` are the UTF-8 bytes on an ASCII platform for that code point. This function returns `"\xC2\xA0"` on an ASCII platform, and `"\x80\x41"` on an EBCDIC 1047 one.

But easiest is, if the character is specifiable as a literal, like `"A"` or `"%"`, to use that; if not so specificable, you can use use `\N{}` , if the side effects aren't troublesome. Simply specify all your characters in hex, using `\N{U+ZZ}` instead of `\xZZ`. `\N{}` is the Unicode name, and so it always gives you the Unicode character. `\N{U+41}` is the character whose Unicode code point is `0x41`, hence is `'A'` on all platforms. The side effects are:

1)

> These select Unicode rules. That means that in double-quotish strings, the string is always converted to UTF-8 to force a Unicode interpretation (you can `utf8::downgrade()` afterwards to convert back to non-UTF8, if possible). In regular expression patterns, the conversion isn't done, but if the character set modifier would otherwise be `/d`, it is changed to `/u`.

2)

> If you use the form `\N{`*`character name`*`}`, the `charnames` module gets automatically loaded. This may not be suitable for the test level you are doing.

If you are testing locales (see Section 38.1 [perllocale NAME], page 701), there are helper functions in `t/loc_tools.pl` to enable you to see what locales there are on the current platform.

29.8.1 Special `make test` targets

There are various special make targets that can be used to test Perl slightly differently than the standard "test" target. Not all them are expected to give a 100% success rate. Many of them have several aliases, and many of them are not available on certain operating systems.

- test_porting

  This runs some basic sanity tests on the source tree and helps catch basic errors before you submit a patch.

- minitest

  Run `miniperl` on `t/base`, `t/comp`, `t/cmd`, `t/run`, `t/io`, `t/op`, `t/uni` and `t/mro` tests.

- test.valgrind check.valgrind

  (Only in Linux) Run all the tests using the memory leak + naughty memory access tool "valgrind". The log files will be named `testname.valgrind`.

- test_harness

  Run the test suite with the `t/harness` controlling program, instead of `t/TEST`. `t/harness` is more sophisticated, and uses the `Test-Harness` module, thus using this test target supposes that perl mostly works. The main advantage for our purposes is that it prints a detailed summary of failed tests at the end. Also, unlike `t/TEST`, it doesn't redirect stderr to stdout.

  Note that under Win32 `t/harness` is always used instead of `t/TEST`, so there is no special "test_harness" target.

  Under Win32's "test" target you may use the TEST_SWITCHES and TEST_FILES environment variables to control the behaviour of `t/harness`. This means you can say

  ```
  nmake test TEST_FILES="op/*.t"
  nmake test TEST_SWITCHES="-torture" TEST_FILES="op/*.t"
  ```

- test-notty test_notty

  Sets PERL_SKIP_TTY_TEST to true before running normal test.

29.8.2 Parallel tests

The core distribution can now run its regression tests in parallel on Unix-like platforms. Instead of running `make test`, set `TEST_JOBS` in your environment to the number of tests to run in parallel, and run `make test_harness`. On a Bourne-like shell, this can be done as

```
TEST_JOBS=3 make test_harness  # Run 3 tests in parallel
```

An environment variable is used, rather than parallel make itself, because `TAP-Harness` needs to be able to schedule individual non-conflicting test scripts itself, and there is no standard interface to `make` utilities to interact with their job schedulers.

Note that currently some test scripts may fail when run in parallel (most notably `ext/IO/t/io_dir.t`). If necessary, run just the failing scripts again sequentially and see if the failures go away.

29.8.3 Running tests by hand

You can run part of the test suite by hand by using one of the following commands from the `t/` directory:

```
./perl -I../lib TEST list-of-.t-files
```

or

```
./perl -I../lib harness list-of-.t-files
```

(If you don't specify test scripts, the whole test suite will be run.)

29.8.4 Using t/harness for testing

If you use `harness` for testing, you have several command line options available to you. The arguments are as follows, and are in the order that they must appear if used together.

```
harness -v -torture -re=pattern LIST OF FILES TO TEST
harness -v -torture -re LIST OF PATTERNS TO MATCH
```

If `LIST OF FILES TO TEST` is omitted, the file list is obtained from the manifest. The file list may include shell wildcards which will be expanded out.

- -v

  Run the tests under verbose mode so you can see what tests were run, and debug output.

- -torture

  Run the torture tests as well as the normal set.

- -re=PATTERN

  Filter the file list so that all the test files run match PATTERN. Note that this form is distinct from the **-re LIST OF PATTERNS** form below in that it allows the file list to be provided as well.

- -re LIST OF PATTERNS

  Filter the file list so that all the test files run match /(LIST`|`OF`|`PATTERNS)/. Note that with this form the patterns are joined by '`|`' and you cannot supply a list of files, instead the test files are obtained from the MANIFEST.

You can run an individual test by a command similar to

```
./perl -I../lib path/to/foo.t
```

except that the harnesses set up some environment variables that may affect the execution of the test:

- PERL_CORE=1

  indicates that we're running this test as part of the perl core test suite. This is useful for modules that have a dual life on CPAN.

- PERL_DESTRUCT_LEVEL=2

  is set to 2 if it isn't set already (see Section 30.8.1 [perlhacktips PERL_DESTRUCT_LEVEL], page 596).

- PERL

  (used only by `t/TEST`) if set, overrides the path to the perl executable that should be used to run the tests (the default being `./perl`).

- PERL_SKIP_TTY_TEST

  if set, tells to skip the tests that need a terminal. It's actually set automatically by the Makefile, but can also be forced artificially by running 'make test_notty'.

29.8.4.1 Other environment variables that may influence tests

- PERL_TEST_Net_Ping

  Setting this variable runs all the Net::Ping modules tests, otherwise some tests that interact with the outside world are skipped. See `perl58delta`.

- PERL_TEST_NOVREXX

  Setting this variable skips the vrexx.t tests for OS2::REXX.

- PERL_TEST_NUMCONVERTS

  This sets a variable in op/numconvert.t.

- PERL_TEST_MEMORY

  Setting this variable includes the tests in `t/bigmem/`. This should be set to the number of gigabytes of memory available for testing, eg. `PERL_TEST_MEMORY=4` indicates that tests that require 4GiB of available memory can be run safely.

See also the documentation for the Test and Test::Harness modules, for more environment variables that affect testing.

29.8.5 Performance testing

The file `t/perf/benchmarks` contains snippets of perl code which are intended to be benchmarked across a range of perls by the `Porting/bench.pl` tool. If you fix or enhance a performance issue, you may want to add a representative code sample to the file, then run `bench.pl` against the previous and current perls to see what difference it has made, and whether anything else has slowed down as a consequence.

The file `t/perf/opcount.t` is designed to test whether a particular code snippet has been compiled into an optree containing specified numbers of particular op types. This is good for testing whether optimisations which alter ops, such as converting an `aelem` op into an `aelemfast` op, are really doing that.

The files `t/perf/speed.t` and `t/re/speed.t` are designed to test things that run thousands of times slower if a particular optimisation is broken (for example, the utf8 length cache on long utf8 strings). Add a test that will take a fraction of a second normally, and minutes otherwise, causing the test file to time out on failure.

29.9 MORE READING FOR GUTS HACKERS

To hack on the Perl guts, you'll need to read the following things:

- Section 71.1 [perlsource NAME], page 1208

  An overview of the Perl source tree. This will help you find the files you're looking for.

- Section 33.1 [perlinterp NAME], page 627

  An overview of the Perl interpreter source code and some details on how Perl does what it does.

- Section 31.1 [perlhacktut NAME], page 601

  This document walks through the creation of a small patch to Perl's C code. If you're just getting started with Perl core hacking, this will help you understand how it works.

- Section 30.1 [perlhacktips NAME], page 578

  More details on hacking the Perl core. This document focuses on lower level details such as how to write tests, compilation issues, portability, debugging, etc.

  If you plan on doing serious C hacking, make sure to read this.

- Section 28.1 [perlguts NAME], page 512

  This is of paramount importance, since it's the documentation of what goes where in the Perl source. Read it over a couple of times and it might start to make sense - don't worry if it doesn't yet, because the best way to study it is to read it in conjunction with poking at Perl source, and we'll do that later on.

  Gisle Aas's "illustrated perlguts", also known as *illguts*, has very helpful pictures:

  `http://search.cpan.org/dist/illguts/`

- `perlxstut` and `perlxs`

  A working knowledge of XSUB programming is incredibly useful for core hacking; XSUBs use techniques drawn from the PP code, the portion of the guts that actually executes a Perl program. It's a lot gentler to learn those techniques from simple examples and explanation than from the core itself.

- `perlapi`

  The documentation for the Perl API explains what some of the internal functions do, as well as the many macros used in the source.

- `Porting/pumpkin.pod`

  This is a collection of words of wisdom for a Perl porter; some of it is only useful to the pumpkin holder, but most of it applies to anyone wanting to go about Perl development.

29.10 CPAN TESTERS AND PERL SMOKERS

The CPAN testers (http://testers.cpan.org/) are a group of volunteers who test CPAN modules on a variety of platforms.

Perl Smokers (http://www.nntp.perl.org/group/perl.daily-build/ and http://www.nntp.perl.org/group/perl.daily-build.reports/) automatically test Perl source releases on platforms with various configurations.

Both efforts welcome volunteers. In order to get involved in smoke testing of the perl itself visit `http://search.cpan.org/dist/Test-Smoke/`. In order to start smoke testing CPAN modules visit `http://search.cpan.org/dist/CPANPLUS-YACSmoke/` or `http://search.cpan.org/dist/minismokebox/` or `http://search.cpan.org/dist/CPAN-Reporter/`.

29.11 WHAT NEXT?

If you've read all the documentation in the document and the ones listed above, you're more than ready to hack on Perl.

Here's some more recommendations

- Subscribe to perl5-porters, follow the patches and try and understand them; don't be afraid to ask if there's a portion you're not clear on - who knows, you may unearth a bug in the patch...
- Do read the README associated with your operating system, e.g. README.aix on the IBM AIX OS. Don't hesitate to supply patches to that README if you find anything missing or changed over a new OS release.
- Find an area of Perl that seems interesting to you, and see if you can work out how it works. Scan through the source, and step over it in the debugger. Play, poke, investigate, fiddle! You'll probably get to understand not just your chosen area but a much wider range of `perl`'s activity as well, and probably sooner than you'd think.

29.11.1 "The Road goes ever on and on, down from the door where it began."

If you can do these things, you've started on the long road to Perl porting. Thanks for wanting to help make Perl better - and happy hacking!

29.11.2 Metaphoric Quotations

If you recognized the quote about the Road above, you're in luck.

Most software projects begin each file with a literal description of each file's purpose. Perl instead begins each with a literary allusion to that file's purpose.

Like chapters in many books, all top-level Perl source files (along with a few others here and there) begin with an epigrammatic inscription that alludes, indirectly and metaphorically, to the material you're about to read.

Quotations are taken from writings of J.R.R. Tolkien pertaining to his Legendarium, almost always from *The Lord of the Rings*. Chapters and page numbers are given using the following editions:

- *The Hobbit*, by J.R.R. Tolkien. The hardcover, 70th-anniversary edition of 2007 was used, published in the UK by Harper Collins Publishers and in the US by the Houghton Mifflin Company.
- *The Lord of the Rings*, by J.R.R. Tolkien. The hardcover, 50th-anniversary edition of 2004 was used, published in the UK by Harper Collins Publishers and in the US by the Houghton Mifflin Company.
- *The Lays of Beleriand*, by J.R.R. Tolkien and published posthumously by his son and literary executor, C.J.R. Tolkien, being the 3rd of the 12 volumes in Christopher's mammoth *History of Middle Earth*. Page numbers derive from the hardcover edition, first published in 1983 by George Allen & Unwin; no page numbers changed for the special 3-volume omnibus edition of 2002 or the various trade-paper editions, all again now by Harper Collins or Houghton Mifflin.

Other JRRT books fair game for quotes would thus include *The Adventures of Tom Bombadil*, *The Silmarillion*, *Unfinished Tales*, and *The Tale of the Children of Hurin*, all but the first posthumously assembled by CJRT. But *The Lord of the Rings* itself is perfectly fine and probably best to quote from, provided you can find a suitable quote there.

So if you were to supply a new, complete, top-level source file to add to Perl, you should conform to this peculiar practice by yourself selecting an appropriate quotation from Tolkien, retaining the original spelling and punctuation and using the same format the rest of the quotes are in. Indirect and oblique is just fine; remember, it's a metaphor, so being meta is, after all, what it's for.

29.12 AUTHOR

This document was originally written by Nathan Torkington, and is maintained by the perl5-porters mailing list.

30 perlhacktips

30.1 NAME

perlhacktips - Tips for Perl core C code hacking

30.2 DESCRIPTION

This document will help you learn the best way to go about hacking on the Perl core C code. It covers common problems, debugging, profiling, and more.

If you haven't read Section 29.1 [perlhack NAME], page 562 and Section 31.1 [perlhacktut NAME], page 601 yet, you might want to do that first.

30.3 COMMON PROBLEMS

Perl source plays by ANSI C89 rules: no C99 (or C++) extensions. In some cases we have to take pre-ANSI requirements into consideration. You don't care about some particular platform having broken Perl? I hear there is still a strong demand for J2EE programmers.

30.3.1 Perl environment problems

- Not compiling with threading

  Compiling with threading (-Duseithreads) completely rewrites the function prototypes of Perl. You better try your changes with that. Related to this is the difference between "Perl_-less" and "Perl_-ly" APIs, for example:

  ```
  Perl_sv_setiv(aTHX_ ...);
  sv_setiv(...);
  ```

  The first one explicitly passes in the context, which is needed for e.g. threaded builds. The second one does that implicitly; do not get them mixed. If you are not passing in a aTHX_, you will need to do a dTHX (or a dVAR) as the first thing in the function.

  See Section 28.9 [perlguts How multiple interpreters and concurrency are supported], page 547 for further discussion about context.

- Not compiling with -DDEBUGGING

  The DEBUGGING define exposes more code to the compiler, therefore more ways for things to go wrong. You should try it.

- Introducing (non-read-only) globals

  Do not introduce any modifiable globals, truly global or file static. They are bad form and complicate multithreading and other forms of concurrency. The right way is to introduce them as new interpreter variables, see `intrpvar.h` (at the very end for binary compatibility).

  Introducing read-only (const) globals is okay, as long as you verify with e.g. `nm libperl.a|egrep -v ' [TURtr] '` (if your `nm` has BSD-style output) that the data you added really is read-only. (If it is, it shouldn't show up in the output of that command.)

  If you want to have static strings, make them constant:

```
static const char etc[] = "...";
```

If you want to have arrays of constant strings, note carefully the right combination of `const`s:

```
static const char * const yippee[] =
    {"hi", "ho", "silver"};
```

There is a way to completely hide any modifiable globals (they are all moved to heap), the compilation setting `-DPERL_GLOBAL_STRUCT_PRIVATE`. It is not normally used, but can be used for testing, read more about it in Section 28.9.1 [perlguts Background and PERL_IMPLICIT_CONTEXT], page 547.

- Not exporting your new function

  Some platforms (Win32, AIX, VMS, OS/2, to name a few) require any function that is part of the public API (the shared Perl library) to be explicitly marked as exported. See the discussion about `embed.pl` in Section 28.1 [perlguts NAME], page 512.

- Exporting your new function

  The new shiny result of either genuine new functionality or your arduous refactoring is now ready and correctly exported. So what could possibly go wrong?

  Maybe simply that your function did not need to be exported in the first place. Perl has a long and not so glorious history of exporting functions that it should not have.

  If the function is used only inside one source code file, make it static. See the discussion about `embed.pl` in Section 28.1 [perlguts NAME], page 512.

  If the function is used across several files, but intended only for Perl's internal use (and this should be the common case), do not export it to the public API. See the discussion about `embed.pl` in Section 28.1 [perlguts NAME], page 512.

30.3.2 Portability problems

The following are common causes of compilation and/or execution failures, not common to Perl as such. The C FAQ is good bedtime reading. Please test your changes with as many C compilers and platforms as possible; we will, anyway, and it's nice to save oneself from public embarrassment.

If using gcc, you can add the `-std=c89` option which will hopefully catch most of these unportabilities. (However it might also catch incompatibilities in your system's header files.)

Use the Configure `-Dgccansipedantic` flag to enable the gcc `-ansi -pedantic` flags which enforce stricter ANSI rules.

If using the `gcc -Wall` note that not all the possible warnings (like `-Wunitialized`) are given unless you also compile with `-O`.

Note that if using gcc, starting from Perl 5.9.5 the Perl core source code files (the ones at the top level of the source code distribution, but not e.g. the extensions under ext/) are automatically compiled with as many as possible of the `-std=c89`, `-ansi`, `-pedantic`, and a selection of `-W` flags (see cflags.SH).

Also study Section 56.1 [perlport NAME], page 951 carefully to avoid any bad assumptions about the operating system, filesystems, character set, and so forth.

You may once in a while try a "make microperl" to see whether we can still compile Perl with just the bare minimum of interfaces. (See README.micro.)

Do not assume an operating system indicates a certain compiler.

- Casting pointers to integers or casting integers to pointers

  ```
      void castaway(U8* p)
      {
        IV i = p;
  ```

  or

  ```
      void castaway(U8* p)
      {
        IV i = (IV)p;
  ```

  Both are bad, and broken, and unportable. Use the PTR2IV() macro that does it right. (Likewise, there are PTR2UV(), PTR2NV(), INT2PTR(), and NUM2PTR().)

- Casting between function pointers and data pointers

  Technically speaking casting between function pointers and data pointers is unportable and undefined, but practically speaking it seems to work, but you should use the FPTR2DPTR() and DPTR2FPTR() macros. Sometimes you can also play games with unions.

- Assuming sizeof(int) == sizeof(long)

  There are platforms where longs are 64 bits, and platforms where ints are 64 bits, and while we are out to shock you, even platforms where shorts are 64 bits. This is all legal according to the C standard. (In other words, "long long" is not a portable way to specify 64 bits, and "long long" is not even guaranteed to be any wider than "long".)

  Instead, use the definitions IV, UV, IVSIZE, I32SIZE, and so forth. Avoid things like I32 because they are **not** guaranteed to be *exactly* 32 bits, they are *at least* 32 bits, nor are they guaranteed to be **int** or **long**. If you really explicitly need 64-bit variables, use I64 and U64, but only if guarded by HAS_QUAD.

- Assuming one can dereference any type of pointer for any type of data

  ```
    char *p = ...;
    long pony = *p;    /* BAD */
  ```

  Many platforms, quite rightly so, will give you a core dump instead of a pony if the p happens not to be correctly aligned.

- Lvalue casts

  ```
    (int)*p = ...;    /* BAD */
  ```

  Simply not portable. Get your lvalue to be of the right type, or maybe use temporary variables, or dirty tricks with unions.

- Assume **anything** about structs (especially the ones you don't control, like the ones coming from the system headers)

  - That a certain field exists in a struct
  - That no other fields exist besides the ones you know of
  - That a field is of certain signedness, sizeof, or type
  - That the fields are in a certain order
    - While C guarantees the ordering specified in the struct definition, between different platforms the definitions might differ

  - That the sizeof(struct) or the alignments are the same everywhere
    - There might be padding bytes between the fields to align the fields - the bytes can be anything
    - Structs are required to be aligned to the maximum alignment required by the fields - which for native types is for usually equivalent to sizeof() of the field

- Assuming the character set is ASCIIish

  Perl can compile and run under EBCDIC platforms. See Section 19.1 [perlebcdic NAME], page 266. This is transparent for the most part, but because the character sets differ, you shouldn't use numeric (decimal, octal, nor hex) constants to refer to characters. You can safely say `'A'`, but not `0x41`. You can safely say `'\n'`, but not `\012`. However, you can use macros defined in `utf8.h` to specify any code point portably. `LATIN1_TO_NATIVE(0xDF)` is going to be the code point that means LATIN SMALL LETTER SHARP S on whatever platform you are running on (on ASCII platforms it compiles without adding any extra code, so there is zero performance hit on those). The acceptable inputs to `LATIN1_TO_NATIVE` are from `0x00` through `0xFF`. If your input isn't guaranteed to be in that range, use `UNICODE_TO_NATIVE` instead. `NATIVE_TO_LATIN1` and `NATIVE_TO_UNICODE` translate the opposite direction.

  If you need the string representation of a character that doesn't have a mnemonic name in C, you should add it to the list in `regen/unicode_constants.pl`, and have Perl create `#define`'s for you, based on the current platform.

  Note that the `is`*`FOO`* and `to`*`FOO`* macros in `handy.h` work properly on native code points and strings.

  Also, the range 'A' - 'Z' in ASCII is an unbroken sequence of 26 upper case alphabetic characters. That is not true in EBCDIC. Nor for 'a' to 'z'. But '0' - '9' is an unbroken range in both systems. Don't assume anything about other ranges. (Note that special handling of ranges in regular expression patterns makes it appear to Perl code that the aforementioned ranges are all unbroken.)

  Many of the comments in the existing code ignore the possibility of EBCDIC, and may be wrong therefore, even if the code works. This is actually a tribute to the successful transparent insertion of being able to handle EBCDIC without having to change pre-existing code.

  UTF-8 and UTF-EBCDIC are two different encodings used to represent Unicode code points as sequences of bytes. Macros with the same names (but different definitions) in `utf8.h` and `utfebcdic.h` are used to allow the calling code to think that there is only one such encoding. This is almost always referred to as `utf8`, but it means the EBCDIC version as well. Again, comments in the code may well be wrong even if the code itself is right. For example, the concept of UTF-8 `invariant characters` differs between ASCII and EBCDIC. On ASCII platforms, only characters that do not have the high-order bit set (i.e. whose ordinals are strict ASCII, 0 - 127) are invariant, and the documentation and comments in the code may assume that, often referring to something like, say, `hibit`. The situation differs and is not so simple on EBCDIC machines, but as long as the code itself uses the `NATIVE_IS_INVARIANT()` macro appropriately, it works, even if the comments are wrong.

  As noted in Section 29.8 [perlhack TESTING], page 571, when writing test scripts, the file `t/charset_tools.pl` contains some helpful functions for writing tests valid on both ASCII and EBCDIC platforms. Sometimes, though, a test can't use a function and it's inconvenient to have different test versions depending on the platform. There are 20 code points that are the same in all 4 character sets currently recognized by Perl (the 3 EBCDIC code pages plus ISO 8859-1 (ASCII/Latin1)). These can be used in such tests, though there is a small possibility that Perl will become available in yet another character set, breaking your test. All but one of these code points are C0 control characters. The most significant controls that are the same are `\0`, `\r`, and `\N{VT}` (also specifiable as `\cK`, `\x0B`, `\N{U+0B}`, or `\013`). The single non-control is U+00B6 PILCROW SIGN. The controls that are the same have the same bit pattern in all 4 character sets, regardless of the UTF8ness of the string containing them. The bit pattern for U+B6 is the same in all 4 for non-UTF8 strings, but differs in each when its containing string is UTF-8 encoded. The only other code points that have some sort of sameness across all 4 character sets are the pair 0xDC and 0xFC. Together these represent upper- and lowercase LATIN LETTER U WITH DIAERESIS, but which is upper and which is lower may be reversed: 0xDC is the capital in Latin1 and 0xFC is the small letter, while 0xFC is the capital in EBCDIC and 0xDC is the small one. This factoid may be exploited in writing case insensitive tests that are the same across all 4 character sets.

- Assuming the character set is just ASCII

  ASCII is a 7 bit encoding, but bytes have 8 bits in them. The 128 extra characters have different meanings depending on the locale. Absent a locale, currently these extra characters are generally considered to be unassigned, and this has presented some problems. This has being changed starting in 5.12 so that these characters can be considered to be Latin-1 (ISO-8859-1).

- Mixing #define and #ifdef

  ```
  #define BURGLE(x) ... \
  #ifdef BURGLE_OLD_STYLE        /* BAD */
  ... do it the old way ... \
  #else
  ... do it the new way ... \
  #endif
  ```

  You cannot portably "stack" cpp directives. For example in the above you need two separate BURGLE() #defines, one for each #ifdef branch.

- Adding non-comment stuff after #endif or #else

  ```
  #ifdef SNOSH
  ...
  #else !SNOSH    /* BAD */
  ...
  #endif SNOSH    /* BAD */
  ```

  The #endif and #else cannot portably have anything non-comment after them. If you want to document what is going (which is a good idea especially if the branches are long), use (C) comments:

  ```
  #ifdef SNOSH
  ...
  #else /* !SNOSH */
  ```

```
...
#endif /* SNOSH */
```

The gcc option `-Wendif-labels` warns about the bad variant (by default on starting from Perl 5.9.4).

- Having a comma after the last element of an enum list

  ```
  enum color {
    CERULEAN,
    CHARTREUSE,
    CINNABAR,     /* BAD */
  };
  ```

  is not portable. Leave out the last comma.

  Also note that whether enums are implicitly morphable to ints varies between compilers, you might need to (int).

- Using //-comments

  ```
  // This function bamfoodles the zorklator.   /* BAD */
  ```

  That is C99 or C++. Perl is C89. Using the //-comments is silently allowed by many C compilers but cranking up the ANSI C89 strictness (which we like to do) causes the compilation to fail.

- Mixing declarations and code

  ```
  void zorklator()
  {
    int n = 3;
    set_zorkmids(n);    /* BAD */
    int q = 4;
  ```

  That is C99 or C++. Some C compilers allow that, but you shouldn't.

  The gcc option `-Wdeclaration-after-statements` scans for such problems (by default on starting from Perl 5.9.4).

- Introducing variables inside for()

  ```
  for(int i = ...; ...; ...) {    /* BAD */
  ```

  That is C99 or C++. While it would indeed be awfully nice to have that also in C89, to limit the scope of the loop variable, alas, we cannot.

- Mixing signed char pointers with unsigned char pointers

  ```
  int foo(char *s) { ... }
  ...
  unsigned char *t = ...; /* Or U8* t = ... */
  foo(t);   /* BAD */
  ```

  While this is legal practice, it is certainly dubious, and downright fatal in at least one platform: for example VMS cc considers this a fatal error. One cause for people often making this mistake is that a "naked char" and therefore dereferencing a "naked char pointer" have an undefined signedness: it depends on the compiler and the flags of the compiler and the underlying platform whether the result is signed or unsigned. For this very same reason using a 'char' as an array index is bad.

- Macros that have string constants and their arguments as substrings of the string constants

  ```
  #define FOO(n) printf("number = %d\n", n)    /* BAD */
  FOO(10);
  ```

  Pre-ANSI semantics for that was equivalent to

  ```
  printf("10umber = %d\10");
  ```

  which is probably not what you were expecting. Unfortunately at least one reasonably common and modern C compiler does "real backward compatibility" here, in AIX that is what still happens even though the rest of the AIX compiler is very happily C89.

- Using printf formats for non-basic C types

  ```
  IV i = ...;
  printf("i = %d\n", i);    /* BAD */
  ```

  While this might by accident work in some platform (where IV happens to be an `int`), in general it cannot. IV might be something larger. Even worse the situation is with more specific types (defined by Perl's configuration step in `config.h`):

  ```
  Uid_t who = ...;
  printf("who = %d\n", who);    /* BAD */
  ```

  The problem here is that Uid_t might be not only not `int`-wide but it might also be unsigned, in which case large uids would be printed as negative values.

  There is no simple solution to this because of printf()'s limited intelligence, but for many types the right format is available as with either 'f' or '_f' suffix, for example:

  ```
  IVdf /* IV in decimal */
  UVxf /* UV is hexadecimal */

  printf("i = %"IVdf"\n", i); /* The IVdf is a string constant. */

  Uid_t_f /* Uid_t in decimal */

  printf("who = %"Uid_t_f"\n", who);
  ```

  Or you can try casting to a "wide enough" type:

  ```
  printf("i = %"IVdf"\n", (IV)something_very_small_and_signed);
  ```

  Also remember that the `%p` format really does require a void pointer:

  ```
  U8* p = ...;
  printf("p = %p\n", (void*)p);
  ```

  The gcc option `-Wformat` scans for such problems.

- Blindly using variadic macros

  gcc has had them for a while with its own syntax, and C99 brought them with a standardized syntax. Don't use the former, and use the latter only if the HAS_C99_VARIADIC_MACROS is defined.

- Blindly passing va_list

  Not all platforms support passing va_list to further varargs (stdarg) functions. The right thing to do is to copy the va_list using the Perl_va_copy() if the NEED_VA_COPY is defined.

- Using gcc statement expressions

  ```
  val = ({...;...;...});    /* BAD */
  ```

  While a nice extension, it's not portable. The Perl code does admittedly use them if available to gain some extra speed (essentially as a funky form of inlining), but you shouldn't.

- Binding together several statements in a macro

  Use the macros STMT_START and STMT_END.

  ```
  STMT_START {
      ...
  } STMT_END
  ```

- Testing for operating systems or versions when should be testing for features

  ```
  #ifdef __FOONIX__    /* BAD */
  foo = quux();
  #endif
  ```

  Unless you know with 100% certainty that quux() is only ever available for the "Foonix" operating system **and** that is available **and** correctly working for **all** past, present, **and** future versions of "Foonix", the above is very wrong. This is more correct (though still not perfect, because the below is a compile-time check):

  ```
  #ifdef HAS_QUUX
  foo = quux();
  #endif
  ```

  How does the HAS_QUUX become defined where it needs to be? Well, if Foonix happens to be Unixy enough to be able to run the Configure script, and Configure has been taught about detecting and testing quux(), the HAS_QUUX will be correctly defined. In other platforms, the corresponding configuration step will hopefully do the same.

  In a pinch, if you cannot wait for Configure to be educated, or if you have a good hunch of where quux() might be available, you can temporarily try the following:

  ```
  #if (defined(__FOONIX__) || defined(__BARNIX__))
  # define HAS_QUUX
  #endif

  ...

  #ifdef HAS_QUUX
  foo = quux();
  #endif
  ```

  But in any case, try to keep the features and operating systems separate.

- Assuming the contents of static memory pointed to by the return values of Perl wrappers for C library functions doesn't change. Many C library functions return pointers to static storage that can be overwritten by subsequent calls to the same or related functions. Perl has light-weight wrappers for some of these functions, and which don't make copies of the static memory. A good example is the interface to the environment variables that are in effect for the program. Perl has `PerlEnv_getenv` to get values from the environment. But the return is a pointer to static memory in the C library. If you are using the value to immediately test for something, that's fine, but if you save the value and expect it to be unchanged by later processing, you would be wrong, but perhaps you wouldn't know it because different C library implementations behave differently, and the one on the platform you're testing on might work for your situation. But on some platforms, a subsequent call to `PerlEnv_getenv` or related function WILL overwrite the memory that your first call points to. This has led to some hard-to-debug problems. Do a Section "savepv" in `perlapi` to make a copy, thus avoiding these problems. You will have to free the copy when you're done to avoid memory leaks. If you don't have control over when it gets freed, you'll need to make the copy in a mortal scalar, like so:

```
if ((s = PerlEnv_getenv("foo") == NULL) {
   ... /* handle NULL case */
}
else {
    s = SvPVX(sv_2mortal(newSVpv(s, 0)));
}
```

The above example works only if `"s"` is `NUL`-terminated; otherwise you have to pass its length to `newSVpv`.

30.3.3 Problematic System Interfaces

- malloc(0), realloc(0), calloc(0, 0) are non-portable. To be portable allocate at least one byte. (In general you should rarely need to work at this low level, but instead use the various malloc wrappers.)
- snprintf() - the return type is unportable. Use my_snprintf() instead.

30.3.4 Security problems

Last but not least, here are various tips for safer coding. See also Section 9.1 [perlclib NAME], page 62 for libc/stdio replacements one should use.

- Do not use gets()

  Or we will publicly ridicule you. Seriously.

- Do not use tmpfile()

  Use mkstemp() instead.

- Do not use strcpy() or strcat() or strncpy() or strncat()

  Use my_strlcpy() and my_strlcat() instead: they either use the native implementation, or Perl's own implementation (borrowed from the public domain implementation of INN).

- Do not use sprintf() or vsprintf()

  If you really want just plain byte strings, use my_snprintf() and my_vsnprintf() instead, which will try to use snprintf() and vsnprintf() if those safer APIs are available. If you want something fancier than a plain byte string, use Section "form" in `perlapi` or SVs and Section "sv_catpvf" in `perlapi`.

  Note that glibc `printf()`, `sprintf()`, etc. are buggy before glibc version 2.17. They won't allow a `%.s` format with a precision to create a string that isn't valid UTF-8 if the current underlying locale of the program is UTF-8. What happens is that the `%s` and its operand are simply skipped without any notice. `https://sourceware.org/bugzilla/show_bug.cgi?id=6530`.

- Do not use atoi()

  Use grok_atoUV() instead. atoi() has ill-defined behavior on overflows, and cannot be used for incremental parsing. It is also affected by locale, which is bad.

- Do not use strtol() or strtoul()

  Use grok_atoUV() instead. strtol() or strtoul() (or their IV/UV-friendly macro disguises, Strtol() and Strtoul(), or Atol() and Atoul() are affected by locale, which is bad.

30.4 DEBUGGING

You can compile a special debugging version of Perl, which allows you to use the `-D` option of Perl to tell more about what Perl is doing. But sometimes there is no alternative than to dive in with a debugger, either to see the stack trace of a core dump (very useful in a bug report), or trying to figure out what went wrong before the core dump happened, or how did we end up having wrong or unexpected results.

30.4.1 Poking at Perl

To really poke around with Perl, you'll probably want to build Perl for debugging, like this:

```
./Configure -d -D optimize=-g
make
```

`-g` is a flag to the C compiler to have it produce debugging information which will allow us to step through a running program, and to see in which C function we are at (without the debugging information we might see only the numerical addresses of the functions, which is not very helpful).

`Configure` will also turn on the `DEBUGGING` compilation symbol which enables all the internal debugging code in Perl. There are a whole bunch of things you can debug with this: Section 69.1 [perlrun NAME], page 1176 lists them all, and the best way to find out about them is to play about with them. The most useful options are probably

```
l  Context (loop) stack processing
t  Trace execution
o  Method and overloading resolution
c  String/numeric conversions
```

Some of the functionality of the debugging code can be achieved using XS modules.

```
-Dr => use re 'debug'
-Dx => use O 'Debug'
```

30.4.2 Using a source-level debugger

If the debugging output of `-D` doesn't help you, it's time to step through perl's execution with a source-level debugger.

- We'll use `gdb` for our examples here; the principles will apply to any debugger (many vendors call their debugger `dbx`), but check the manual of the one you're using.

To fire up the debugger, type

```
gdb ./perl
```

Or if you have a core dump:

```
gdb ./perl core
```

You'll want to do that in your Perl source tree so the debugger can read the source code. You should see the copyright message, followed by the prompt.

```
(gdb)
```

`help` will get you into the documentation, but here are the most useful commands:

- run [args]

  Run the program with the given arguments.

- break function_name
- break source.c:xxx

  Tells the debugger that we'll want to pause execution when we reach either the named function (but see Section 28.10 [perlguts Internal Functions], page 552!) or the given line in the named source file.

- step

  Steps through the program a line at a time.

- next

  Steps through the program a line at a time, without descending into functions.

- continue

  Run until the next breakpoint.

- finish

  Run until the end of the current function, then stop again.

- 'enter'

  Just pressing Enter will do the most recent operation again - it's a blessing when stepping through miles of source code.

- ptype

  Prints the C definition of the argument given.

  ```
  (gdb) ptype PL_op
  type = struct op {
      OP *op_next;
      OP *op_sibparent;
      OP *(*op_ppaddr)(void);
      PADOFFSET op_targ;
      unsigned int op_type : 9;
      unsigned int op_opt : 1;
      unsigned int op_slabbed : 1;
      unsigned int op_savefree : 1;
      unsigned int op_static : 1;
      unsigned int op_folded : 1;
      unsigned int op_spare : 2;
  ```

```
        U8 op_flags;
        U8 op_private;
    } *
```

- print

  Execute the given C code and print its results. **WARNING**: Perl makes heavy use of macros, and gdb does not necessarily support macros (see later Section 30.4.3 [gdb macro support], page 589). You'll have to substitute them yourself, or to invoke cpp on the source code files (see Section 30.8.8 [The .i Targets], page 600) So, for instance, you can't say

  ```
  print SvPV_nolen(sv)
  ```

  but you have to say

  ```
  print Perl_sv_2pv_nolen(sv)
  ```

You may find it helpful to have a "macro dictionary", which you can produce by saying `cpp -dM perl.c | sort`. Even then, cpp won't recursively apply those macros for you.

30.4.3 gdb macro support

Recent versions of gdb have fairly good macro support, but in order to use it you'll need to compile perl with macro definitions included in the debugging information. Using gcc version 3.1, this means configuring with `-Doptimize=-g3`. Other compilers might use a different switch (if they support debugging macros at all).

30.4.4 Dumping Perl Data Structures

One way to get around this macro hell is to use the dumping functions in `dump.c`; these work a little like an internal `Devel-Peek`, but they also cover OPs and other structures that you can't get at from Perl. Let's take an example. We'll use the `$a = $b + $c` we used before, but give it a bit of context: `$b = "6XXXX"; $c = 2.3;`. Where's a good place to stop and poke around?

What about `pp_add`, the function we examined earlier to implement the `+` operator:

```
(gdb) break Perl_pp_add
Breakpoint 1 at 0x46249f: file pp_hot.c, line 309.
```

Notice we use `Perl_pp_add` and not `pp_add` - see Section 28.10 [perlguts Internal Functions], page 552. With the breakpoint in place, we can run our program:

```
(gdb) run -e '$b = "6XXXX"; $c = 2.3; $a = $b + $c'
```

Lots of junk will go past as gdb reads in the relevant source files and libraries, and then:

```
Breakpoint 1, Perl_pp_add () at pp_hot.c:309
309         dSP; dATARGET; tryAMAGICbin(add,opASSIGN);
(gdb) step
311           dPOPTOPnnrl_ul;
(gdb)
```

We looked at this bit of code before, and we said that `dPOPTOPnnrl_ul` arranges for two `NV`s to be placed into `left` and `right` - let's slightly expand it:

```
#define dPOPTOPnnrl_ul  NV right = POPn; \
                        SV *leftsv = TOPs; \
```

```
                         NV left = USE_LEFT(leftsv) ? SvNV(leftsv) : 0.0
```

POPn takes the SV from the top of the stack and obtains its NV either directly (if SvNOK is set) or by calling the sv_2nv function. TOPs takes the next SV from the top of the stack - yes, POPn uses TOPs - but doesn't remove it. We then use SvNV to get the NV from leftsv in the same way as before - yes, POPn uses SvNV.

Since we don't have an NV for $b, we'll have to use sv_2nv to convert it. If we step again, we'll find ourselves there:

```
(gdb) step
Perl_sv_2nv (sv=0xa0675d0) at sv.c:1669
1669        if (!sv)
(gdb)
```

We can now use Perl_sv_dump to investigate the SV:

```
(gdb) print Perl_sv_dump(sv)
SV = PV(0xa057cc0) at 0xa0675d0
REFCNT = 1
FLAGS = (POK,pPOK)
PV = 0xa06a510 "6XXXX"\0
CUR = 5
LEN = 6
$1 = void
```

We know we're going to get 6 from this, so let's finish the subroutine:

```
(gdb) finish
Run till exit from #0  Perl_sv_2nv (sv=0xa0675d0) at sv.c:1671
0x462669 in Perl_pp_add () at pp_hot.c:311
311           dPOPTOPnnrl_ul;
```

We can also dump out this op: the current op is always stored in PL_op, and we can dump it with Perl_op_dump. This'll give us similar output to B-Debug.

```
(gdb) print Perl_op_dump(PL_op)
{
13  TYPE = add  ===> 14
    TARG = 1
    FLAGS = (SCALAR,KIDS)
    {
        TYPE = null  ===> (12)
          (was rv2sv)
        FLAGS = (SCALAR,KIDS)
        {
11          TYPE = gvsv  ===> 12
            FLAGS = (SCALAR)
            GV = main::b
        }
    }
```

# finish this later #

30.4.5 Using gdb to look at specific parts of a program

With the example above, you knew to look for `Perl_pp_add`, but what if there were multiple calls to it all over the place, or you didn't know what the op was you were looking for?

One way to do this is to inject a rare call somewhere near what you're looking for. For example, you could add `study` before your method:

```
study;
```

And in gdb do:

```
(gdb) break Perl_pp_study
```

And then step until you hit what you're looking for. This works well in a loop if you want to only break at certain iterations:

```
for my $c (1..100) {
    study if $c == 50;
}
```

30.4.6 Using gdb to look at what the parser/lexer are doing

If you want to see what perl is doing when parsing/lexing your code, you can use `BEGIN {}`:

```
print "Before\n";
BEGIN { study; }
print "After\n";
```

And in gdb:

```
(gdb) break Perl_pp_study
```

If you want to see what the parser/lexer is doing inside of `if` blocks and the like you need to be a little trickier:

```
if ($a && $b && do { BEGIN { study } 1 } && $c) { ... }
```

30.5 SOURCE CODE STATIC ANALYSIS

Various tools exist for analysing C source code **statically**, as opposed to **dynamically**, that is, without executing the code. It is possible to detect resource leaks, undefined behaviour, type mismatches, portability problems, code paths that would cause illegal memory accesses, and other similar problems by just parsing the C code and looking at the resulting graph, what does it tell about the execution and data flows. As a matter of fact, this is exactly how C compilers know to give warnings about dubious code.

30.5.1 lint, splint

The good old C code quality inspector, `lint`, is available in several platforms, but please be aware that there are several different implementations of it by different vendors, which means that the flags are not identical across different platforms.

There is a lint variant called `splint` (Secure Programming Lint) available from http://www.splint.org/ that should compile on any Unix-like platform.

There are `lint` and <splint> targets in Makefile, but you may have to diddle with the flags (see above).

30.5.2 Coverity

Coverity (http://www.coverity.com/) is a product similar to lint and as a testbed for their product they periodically check several open source projects, and they give out accounts to open source developers to the defect databases.

30.5.3 cpd (cut-and-paste detector)

The cpd tool detects cut-and-paste coding. If one instance of the cut-and-pasted code changes, all the other spots should probably be changed, too. Therefore such code should probably be turned into a subroutine or a macro.

cpd (http://pmd.sourceforge.net/cpd.html) is part of the pmd project (http://pmd.sourceforge.net/). pmd was originally written for static analysis of Java code, but later the cpd part of it was extended to parse also C and C++.

Download the pmd-bin-X.Y.zip () from the SourceForge site, extract the pmd-X.Y.jar from it, and then run that on source code thusly:

```
java -cp pmd-X.Y.jar net.sourceforge.pmd.cpd.CPD \
 --minimum-tokens 100 --files /some/where/src --language c > cpd.txt
```

You may run into memory limits, in which case you should use the -Xmx option:

```
java -Xmx512M ...
```

30.5.4 gcc warnings

Though much can be written about the inconsistency and coverage problems of gcc warnings (like `-Wall` not meaning "all the warnings", or some common portability problems not being covered by `-Wall`, or `-ansi` and `-pedantic` both being a poorly defined collection of warnings, and so forth), gcc is still a useful tool in keeping our coding nose clean.

The `-Wall` is by default on.

The `-ansi` (and its sidekick, `-pedantic`) would be nice to be on always, but unfortunately they are not safe on all platforms, they can for example cause fatal conflicts with the system headers (Solaris being a prime example). If Configure `-Dgccansipedantic` is used, the `cflags` frontend selects `-ansi -pedantic` for the platforms where they are known to be safe.

Starting from Perl 5.9.4 the following extra flags are added:

- `-Wendif-labels`
- `-Wextra`
- `-Wdeclaration-after-statement`

The following flags would be nice to have but they would first need their own Augean stablemaster:

- `-Wpointer-arith`
- `-Wshadow`
- `-Wstrict-prototypes`

The `-Wtraditional` is another example of the annoying tendency of gcc to bundle a lot of warnings under one switch (it would be impossible to deploy in practice because it would complain a lot) but it does contain some warnings that would be beneficial to have available on their own, such as the warning about string constants inside macros containing the macro arguments: this behaved differently pre-ANSI than it does in ANSI, and some C compilers are still in transition, AIX being an example.

30.5.5 Warnings of other C compilers

Other C compilers (yes, there **are** other C compilers than gcc) often have their "strict ANSI" or "strict ANSI with some portability extensions" modes on, like for example the Sun Workshop has its `-Xa` mode on (though implicitly), or the DEC (these days, HP...) has its `-std1` mode on.

30.6 MEMORY DEBUGGERS

NOTE 1: Running under older memory debuggers such as Purify, valgrind or Third Degree greatly slows down the execution: seconds become minutes, minutes become hours. For example as of Perl 5.8.1, the ext/Encode/t/Unicode.t takes extraordinarily long to complete under e.g. Purify, Third Degree, and valgrind. Under valgrind it takes more than six hours, even on a snappy computer. The said test must be doing something that is quite unfriendly for memory debuggers. If you don't feel like waiting, that you can simply kill away the perl process. Roughly valgrind slows down execution by factor 10, AddressSanitizer by factor 2.

NOTE 2: To minimize the number of memory leak false alarms (see Section 30.8.1 [PERL_DESTRUCT_LEVEL], page 596 for more information), you have to set the environment variable PERL_DESTRUCT_LEVEL to 2. For example, like this:

```
env PERL_DESTRUCT_LEVEL=2 valgrind ./perl -Ilib ...
```

NOTE 3: There are known memory leaks when there are compile-time errors within eval or require, seeing `S_doeval` in the call stack is a good sign of these. Fixing these leaks is non-trivial, unfortunately, but they must be fixed eventually.

NOTE 4: `DynaLoader` will not clean up after itself completely unless Perl is built with the Configure option `-Accflags=-DDL_UNLOAD_ALL_AT_EXIT`.

30.6.1 valgrind

The valgrind tool can be used to find out both memory leaks and illegal heap memory accesses. As of version 3.3.0, Valgrind only supports Linux on x86, x86-64 and PowerPC and Darwin (OS X) on x86 and x86-64). The special "test.valgrind" target can be used to run the tests under valgrind. Found errors and memory leaks are logged in files named `testfile.valgrind` and by default output is displayed inline.

Example usage:

```
make test.valgrind
```

Since valgrind adds significant overhead, tests will take much longer to run. The valgrind tests support being run in parallel to help with this:

```
TEST_JOBS=9 make test.valgrind
```

Note that the above two invocations will be very verbose as reachable memory and leak-checking is enabled by default. If you want to just see pure errors, try:

```
VG_OPTS='-q --leak-check=no --show-reachable=no' TEST_JOBS=9 \
    make test.valgrind
```

Valgrind also provides a cachegrind tool, invoked on perl as:

```
VG_OPTS=--tool=cachegrind make test.valgrind
```

As system libraries (most notably glibc) are also triggering errors, valgrind allows to suppress such errors using suppression files. The default suppression file that comes with valgrind already catches a lot of them. Some additional suppressions are defined in `t/perl.supp`.

To get valgrind and for more information see

```
http://valgrind.org/
```

30.6.2 AddressSanitizer

AddressSanitizer is a clang and gcc extension, included in clang since v3.1 and gcc since v4.8. It checks illegal heap pointers, global pointers, stack pointers and use after free errors, and is fast enough that you can easily compile your debugging or optimized perl with it. It does not check memory leaks though. AddressSanitizer is available for Linux, Mac OS X and soon on Windows.

To build perl with AddressSanitizer, your Configure invocation should look like:

```
sh Configure -des -Dcc=clang \
   -Accflags=-faddress-sanitizer -Aldflags=-faddress-sanitizer \
   -Alddlflags=-shared\ -faddress-sanitizer
```

where these arguments mean:

- -Dcc=clang

  This should be replaced by the full path to your clang executable if it is not in your path.

- -Accflags=-faddress-sanitizer

  Compile perl and extensions sources with AddressSanitizer.

- -Aldflags=-faddress-sanitizer

  Link the perl executable with AddressSanitizer.

- -Alddlflags=-shared\ -faddress-sanitizer

  Link dynamic extensions with AddressSanitizer. You must manually specify `-shared` because using `-Alddlflags=-shared` will prevent Configure from setting a default value for `lddlflags`, which usually contains `-shared` (at least on Linux).

See also `http://code.google.com/p/address-sanitizer/wiki/AddressSanitizer`.

30.7 PROFILING

Depending on your platform there are various ways of profiling Perl.

There are two commonly used techniques of profiling executables: *statistical time-sampling* and *basic-block counting*.

The first method takes periodically samples of the CPU program counter, and since the program counter can be correlated with the code generated for functions, we get a statistical view of in which functions the program is spending its time. The caveats are that very small/fast functions have lower probability of showing up in the profile, and that periodically interrupting the program (this is usually done rather frequently, in the scale of milliseconds) imposes an additional overhead that may skew the results. The first problem can be alleviated by running the code for longer (in general this is a good idea for profiling), the second problem is usually kept in guard by the profiling tools themselves.

The second method divides up the generated code into *basic blocks*. Basic blocks are sections of code that are entered only in the beginning and exited only at the end. For example, a conditional jump starts a basic block. Basic block profiling usually works by *instrumenting* the code by adding *enter basic block #nnnn* book-keeping code to the generated code. During the execution of the code the basic block counters are then updated appropriately. The caveat is that the added extra code can skew the results: again, the profiling tools usually try to factor their own effects out of the results.

30.7.1 Gprof Profiling

gprof is a profiling tool available in many Unix platforms which uses *statistical time-sampling*. You can build a profiled version of `perl` by compiling using gcc with the flag `-pg`. Either edit `config.sh` or re-run `Configure`. Running the profiled version of Perl will create an output file called `gmon.out` which contains the profiling data collected during the execution.

quick hint:

```
$ sh Configure -des -Dusedevel -Accflags='-pg' \
    -Aldflags='-pg' -Alddlflags='-pg -shared' \
    && make perl
$ ./perl ... # creates gmon.out in current directory
$ gprof ./perl > out
$ less out
```

(you probably need to add `-shared` to the <-Alddlflags> line until RT #118199 is resolved)

The `gprof` tool can then display the collected data in various ways. Usually `gprof` understands the following options:

- -a

  Suppress statically defined functions from the profile.
- -b

  Suppress the verbose descriptions in the profile.
- -e routine

  Exclude the given routine and its descendants from the profile.
- -f routine

  Display only the given routine and its descendants in the profile.
- -s

  Generate a summary file called `gmon.sum` which then may be given to subsequent gprof runs to accumulate data over several runs.
- -z

  Display routines that have zero usage.

For more detailed explanation of the available commands and output formats, see your own local documentation of `gprof`.

30.7.2 GCC gcov Profiling

basic block profiling is officially available in gcc 3.0 and later. You can build a profiled version of `perl` by compiling using gcc with the flags `-fprofile-arcs -ftest-coverage`. Either edit `config.sh` or re-run `Configure`.

quick hint:

```
$ sh Configure -des -Dusedevel -Doptimize='-g' \
    -Accflags='-fprofile-arcs -ftest-coverage' \
    -Aldflags='-fprofile-arcs -ftest-coverage' \
    -Alddlflags='-fprofile-arcs -ftest-coverage -shared' \
    && make perl
$ rm -f regexec.c.gcov regexec.gcda
$ ./perl ...
$ gcov regexec.c
$ less regexec.c.gcov
```

(you probably need to add `-shared` to the <-Alddlflags> line until RT #118199 is resolved)

Running the profiled version of Perl will cause profile output to be generated. For each source file an accompanying `.gcda` file will be created.

To display the results you use the *gcov* utility (which should be installed if you have gcc 3.0 or newer installed). `gcov` is run on source code files, like this

```
gcov sv.c
```

which will cause `sv.c.gcov` to be created. The `.gcov` files contain the source code annotated with relative frequencies of execution indicated by "#" markers. If you want to generate `.gcov` files for all profiled object files, you can run something like this:

```
for file in `find . -name \*.gcno`
do sh -c "cd `dirname $file` && gcov `basename $file .gcno`"
done
```

Useful options of `gcov` include `-b` which will summarise the basic block, branch, and function call coverage, and `-c` which instead of relative frequencies will use the actual counts. For more information on the use of `gcov` and basic block profiling with gcc, see the latest GNU CC manual. As of gcc 4.8, this is at `http://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html#Gcov-Intro`

30.8 MISCELLANEOUS TRICKS

30.8.1 PERL_DESTRUCT_LEVEL

If you want to run any of the tests yourself manually using e.g. valgrind, please note that by default perl **does not** explicitly cleanup all the memory it has allocated (such as global memory arenas) but instead lets the exit() of the whole program "take care" of such allocations, also known as "global destruction of objects".

There is a way to tell perl to do complete cleanup: set the environment variable PERL_DESTRUCT_LEVEL to a non-zero value. The t/TEST wrapper does set this to 2, and this is what you need to do too, if you don't want to see the "global leaks": For example, for running under valgrind

```
        env PERL_DESTRUCT_LEVEL=2 valgrind ./perl -Ilib t/foo/bar.t
```

(Note: the mod_perl apache module uses also this environment variable for its own purposes and extended its semantics. Refer to the mod_perl documentation for more information. Also, spawned threads do the equivalent of setting this variable to the value 1.)

If, at the end of a run you get the message *N scalars leaked*, you can recompile with `-DDEBUG_LEAKING_SCALARS`, which will cause the addresses of all those leaked SVs to be dumped along with details as to where each SV was originally allocated. This information is also displayed by Devel::Peek. Note that the extra details recorded with each SV increases memory usage, so it shouldn't be used in production environments. It also converts `new_SV()` from a macro into a real function, so you can use your favourite debugger to discover where those pesky SVs were allocated.

If you see that you're leaking memory at runtime, but neither valgrind nor `-DDEBUG_LEAKING_SCALARS` will find anything, you're probably leaking SVs that are still reachable and will be properly cleaned up during destruction of the interpreter. In such cases, using the `-Dm` switch can point you to the source of the leak. If the executable was built with `-DDEBUG_LEAKING_SCALARS`, `-Dm` will output SV allocations in addition to memory allocations. Each SV allocation has a distinct serial number that will be written on creation and destruction of the SV. So if you're executing the leaking code in a loop, you need to look for SVs that are created, but never destroyed between each cycle. If such an SV is found, set a conditional breakpoint within `new_SV()` and make it break only when `PL_sv_serial` is equal to the serial number of the leaking SV. Then you will catch the interpreter in exactly the state where the leaking SV is allocated, which is sufficient in many cases to find the source of the leak.

As `-Dm` is using the PerlIO layer for output, it will by itself allocate quite a bunch of SVs, which are hidden to avoid recursion. You can bypass the PerlIO layer if you use the SV logging provided by `-DPERL_MEM_LOG` instead.

30.8.2 PERL_MEM_LOG

If compiled with `-DPERL_MEM_LOG`, both memory and SV allocations go through logging functions, which is handy for breakpoint setting.

Unless `-DPERL_MEM_LOG_NOIMPL` is also compiled, the logging functions read $ENV{PERL_MEM_LOG} to determine whether to log the event, and if so how:

```
    $ENV{PERL_MEM_LOG} =~ /m/           Log all memory ops
    $ENV{PERL_MEM_LOG} =~ /s/           Log all SV ops
    $ENV{PERL_MEM_LOG} =~ /t/           include timestamp in Log
    $ENV{PERL_MEM_LOG} =~ /^(\d+)/      write to FD given (default is 2)
```

Memory logging is somewhat similar to `-Dm` but is independent of `-DDEBUGGING`, and at a higher level; all uses of Newx(), Renew(), and Safefree() are logged with the caller's source code file and line number (and C function name, if supported by the C compiler). In contrast, `-Dm` is directly at the point of `malloc()`. SV logging is similar.

Since the logging doesn't use PerlIO, all SV allocations are logged and no extra SV allocations are introduced by enabling the logging. If compiled with `-DDEBUG_LEAKING_SCALARS`, the serial number for each SV allocation is also logged.

30.8.3 DDD over gdb

Those debugging perl with the DDD frontend over gdb may find the following useful:

You can extend the data conversion shortcuts menu, so for example you can display an SV's IV value with one click, without doing any typing. To do that simply edit ~/.ddd/init file and add after:

```
! Display shortcuts.
Ddd*gdbDisplayShortcuts: \
/t ()   // Convert to Bin\n\
/d ()   // Convert to Dec\n\
/x ()   // Convert to Hex\n\
/o ()   // Convert to Oct(\n\
```

the following two lines:

```
((XPV*) (())->sv_any )->xpv_pv  // 2pvx\n\
((XPVIV*) (())->sv_any )->xiv_iv // 2ivx
```

so now you can do ivx and pvx lookups or you can plug there the sv_peek "conversion":

```
Perl_sv_peek(my_perl, (SV*)()) // sv_peek
```

(The my_perl is for threaded builds.) Just remember that every line, but the last one, should end with \n\

Alternatively edit the init file interactively via: 3rd mouse button -> New Display -> Edit Menu

Note: you can define up to 20 conversion shortcuts in the gdb section.

30.8.4 C backtrace

On some platforms Perl supports retrieving the C level backtrace (similar to what symbolic debuggers like gdb do).

The backtrace returns the stack trace of the C call frames, with the symbol names (function names), the object names (like "perl"), and if it can, also the source code locations (file:line).

The supported platforms are Linux, and OS X (some *BSD might work at least partly, but they have not yet been tested).

This feature hasn't been tested with multiple threads, but it will only show the backtrace of the thread doing the backtracing.

The feature needs to be enabled with `Configure -Dusecbacktrace`.

The `-Dusecbacktrace` also enables keeping the debug information when compiling/linking (often: `-g`). Many compilers/linkers do support having both optimization and keeping the debug information. The debug information is needed for the symbol names and the source locations.

Static functions might not be visible for the backtrace.

Source code locations, even if available, can often be missing or misleading if the compiler has e.g. inlined code. Optimizer can make matching the source code and the object code quite challenging.

Linux

You **must** have the BFD (-lbfd) library installed, otherwise `perl` will fail to link. The BFD is usually distributed as part of the GNU binutils.

Summary: `Configure ... -Dusecbacktrace` and you need `-lbfd`.

OS X

The source code locations are supported **only** if you have the Developer Tools installed. (BFD is **not** needed.)

Summary: `Configure ... -Dusecbacktrace` and installing the Developer Tools would be good.

Optionally, for trying out the feature, you may want to enable automatic dumping of the backtrace just before a warning or croak (die) message is emitted, by adding `-Accflags=-DUSE_C_BACKTRACE_ON_ERROR` for Configure.

Unless the above additional feature is enabled, nothing about the backtrace functionality is visible, except for the Perl/XS level.

Furthermore, even if you have enabled this feature to be compiled, you need to enable it in runtime with an environment variable: `PERL_C_BACKTRACE_ON_ERROR=10`. It must be an integer higher than zero, telling the desired frame count.

Retrieving the backtrace from Perl level (using for example an XS extension) would be much less exciting than one would hope: normally you would see `runops`, `entersub`, and not much else. This API is intended to be called **from within** the Perl implementation, not from Perl level execution.

The C API for the backtrace is as follows:

get_c_backtrace
free_c_backtrace
get_c_backtrace_dump
dump_c_backtrace

30.8.5 Poison

If you see in a debugger a memory area mysteriously full of 0xABABABAB or 0xEFEFEFEF, you may be seeing the effect of the Poison() macros, see Section 9.1 [perlclib NAME], page 62.

30.8.6 Read-only optrees

Under ithreads the optree is read only. If you want to enforce this, to check for write accesses from buggy code, compile with `-Accflags=-DPERL_DEBUG_READONLY_OPS` to enable code that allocates op memory via `mmap`, and sets it read-only when it is attached to a subroutine. Any write access to an op results in a `SIGBUS` and abort.

This code is intended for development only, and may not be portable even to all Unix variants. Also, it is an 80% solution, in that it isn't able to make all ops read only. Specifically it does not apply to op slabs belonging to `BEGIN` blocks.

However, as an 80% solution it is still effective, as it has caught bugs in the past.

30.8.7 When is a bool not a bool?

On pre-C99 compilers, `bool` is defined as equivalent to `char`. Consequently assignment of any larger type to a `bool` is unsafe and may be truncated. The `cBOOL` macro exists to cast it correctly.

On those platforms and compilers where `bool` really is a boolean (C++, C99), it is easy to forget the cast. You can force `bool` to be a `char` by compiling with `-Accflags=-DPERL_BOOL_AS_CHAR`. You may also wish to run `Configure` with something like

```
-Accflags='-Wconversion -Wno-sign-conversion -Wno-shorten-64-to-32'
```

or your compiler's equivalent to make it easier to spot any unsafe truncations that show up.

30.8.8 The .i Targets

You can expand the macros in a `foo.c` file by saying

```
make foo.i
```

which will expand the macros using cpp. Don't be scared by the results.

30.9 AUTHOR

This document was originally written by Nathan Torkington, and is maintained by the perl5-porters mailing list.

31 perlhacktut

31.1 NAME

perlhacktut - Walk through the creation of a simple C code patch

31.2 DESCRIPTION

This document takes you through a simple patch example.

If you haven't read Section 29.1 [perlhack NAME], page 562 yet, go do that first! You might also want to read through Section 71.1 [perlsource NAME], page 1208 too.

Once you're done here, check out Section 30.1 [perlhacktips NAME], page 578 next.

31.3 EXAMPLE OF A SIMPLE PATCH

Let's take a simple patch from start to finish.

Here's something Larry suggested: if a `U` is the first active format during a `pack`, (for example, `pack "U3C8", @stuff`) then the resulting string should be treated as UTF-8 encoded.

If you are working with a git clone of the Perl repository, you will want to create a branch for your changes. This will make creating a proper patch much simpler. See the Section 26.1 [perlgit NAME], page 492 for details on how to do this.

31.3.1 Writing the patch

How do we prepare to fix this up? First we locate the code in question - the `pack` happens at runtime, so it's going to be in one of the `pp` files. Sure enough, `pp_pack` is in `pp.c`. Since we're going to be altering this file, let's copy it to `pp.c~`.

[Well, it was in `pp.c` when this tutorial was written. It has now been split off with `pp_unpack` to its own file, `pp_pack.c`]

Now let's look over `pp_pack`: we take a pattern into `pat`, and then loop over the pattern, taking each format character in turn into `datum_type`. Then for each possible format character, we swallow up the other arguments in the pattern (a field width, an asterisk, and so on) and convert the next chunk input into the specified format, adding it onto the output SV `cat`.

How do we know if the `U` is the first format in the `pat`? Well, if we have a pointer to the start of `pat` then, if we see a `U` we can test whether we're still at the start of the string. So, here's where `pat` is set up:

```
STRLEN fromlen;
char *pat = SvPVx(*++MARK, fromlen);
char *patend = pat + fromlen;
I32 len;
I32 datumtype;
SV *fromstr;
```

We'll have another string pointer in there:

```
    STRLEN fromlen;
    char *pat = SvPVx(*++MARK, fromlen);
    char *patend = pat + fromlen;
+   char *patcopy;
    I32 len;
    I32 datumtype;
    SV *fromstr;
```

And just before we start the loop, we'll set `patcopy` to be the start of `pat`:

```
    items = SP - MARK;
    MARK++;
    sv_setpvn(cat, "", 0);
+   patcopy = pat;
    while (pat < patend) {
```

Now if we see a `U` which was at the start of the string, we turn on the `UTF8` flag for the output SV, `cat`:

```
+   if (datumtype == 'U' && pat==patcopy+1)
+       SvUTF8_on(cat);
    if (datumtype == '#') {
        while (pat < patend && *pat != '\n')
            pat++;
```

Remember that it has to be `patcopy+1` because the first character of the string is the `U` which has been swallowed into `datumtype!`

Oops, we forgot one thing: what if there are spaces at the start of the pattern? `pack(" U*", @stuff)` will have `U` as the first active character, even though it's not the first thing in the pattern. In this case, we have to advance `patcopy` along with `pat` when we see spaces:

```
    if (isSPACE(datumtype))
        continue;
```

needs to become

```
    if (isSPACE(datumtype)) {
        patcopy++;
        continue;
    }
```

OK. That's the C part done. Now we must do two additional things before this patch is ready to go: we've changed the behaviour of Perl, and so we must document that change. We must also provide some more regression tests to make sure our patch works and doesn't create a bug somewhere else along the line.

31.3.2 Testing the patch

The regression tests for each operator live in `t/op/`, and so we make a copy of `t/op/pack.t` to `t/op/pack.t~`. Now we can add our tests to the end. First, we'll test that the `U` does indeed create Unicode strings.

t/op/pack.t has a sensible ok() function, but if it didn't we could use the one from t/test.pl.

```
require './test.pl';
plan( tests => 159 );
```

so instead of this:

```
print 'not ' unless "1.20.300.4000" eq sprintf "%vd",
                                              pack("U*",1,20,300,4000);
print "ok $test\n"; $test++;
```

we can write the more sensible (see `Test-More` for a full explanation of is() and other testing functions).

```
is( "1.20.300.4000", sprintf "%vd", pack("U*",1,20,300,4000),
                                      "U* produces Unicode" );
```

Now we'll test that we got that space-at-the-beginning business right:

```
is( "1.20.300.4000", sprintf "%vd", pack("  U*",1,20,300,4000),
                                    "  with spaces at the beginning" );
```

And finally we'll test that we don't make Unicode strings if `U` is **not** the first active format:

```
isnt( v1.20.300.4000, sprintf "%vd", pack("C0U*",1,20,300,4000),
                                      "U* not first isn't Unicode" );
```

Mustn't forget to change the number of tests which appears at the top, or else the automated tester will get confused. This will either look like this:

```
print "1..156\n";
```

or this:

```
plan( tests => 156 );
```

We now compile up Perl, and run it through the test suite. Our new tests pass, hooray!

31.3.3 Documenting the patch

Finally, the documentation. The job is never done until the paperwork is over, so let's describe the change we've just made. The relevant place is `pod/perlfunc.pod`; again, we make a copy, and then we'll insert this text in the description of `pack`:

```
=item *

If the pattern begins with a C<U>, the resulting string will be treated
as UTF-8-encoded Unicode. You can force UTF-8 encoding on in a string
with an initial C<U0>, and the bytes that follow will be interpreted as
Unicode characters. If you don't want this to happen, you can begin
your pattern with C<C0> (or anything else) to force Perl not to UTF-8
encode your string, and then follow this with a C<U*> somewhere in your
pattern.
```

31.3.4 Submit

See Section 29.1 [perlhack NAME], page 562 for details on how to submit this patch.

31.4 AUTHOR

This document was originally written by Nathan Torkington, and is maintained by the perl5-porters mailing list.

32 perlhist

32.1 NAME

perlhist - the Perl history records

32.2 DESCRIPTION

This document aims to record the Perl source code releases.

32.3 INTRODUCTION

Perl history in brief, by Larry Wall:

```
Perl 0 introduced Perl to my officemates.
Perl 1 introduced Perl to the world, and changed /\(...\|...\)/ to
    /(...|...)/.  \(Dan Faigin still hasn't forgiven me. :-\)
Perl 2 introduced Henry Spencer's regular expression package.
Perl 3 introduced the ability to handle binary data (embedded nulls).
Perl 4 introduced the first Camel book.  Really.  We mostly just
    switched version numbers so the book could refer to 4.000.
Perl 5 introduced everything else, including the ability to
    introduce everything else.
```

32.4 THE KEEPERS OF THE PUMPKIN

Larry Wall, Andy Dougherty, Tom Christiansen, Charles Bailey, Nick Ing-Simmons, Chip Salzenberg, Tim Bunce, Malcolm Beattie, Gurusamy Sarathy, Graham Barr, Jarkko Hietaniemi, Hugo van der Sanden, Michael Schwern, Rafael Garcia-Suarez, Nicholas Clark, Richard Clamp, Leon Brocard, Dave Mitchell, Jesse Vincent, Ricardo Signes, Steve Hay, Matt S Trout, David Golden, Florian Ragwitz, Tatsuhiko Miyagawa, Chris `BinGOs` Williams, Zefram, Ævar Arnfjörð Bjarmason, Stevan Little, Dave Rolsky, Max Maischein, Abigail, Jesse Luehrs, Tony Cook, Dominic Hargreaves, Aaron Crane, Aristotle Pagaltzis, Matthew Horsfall, Peter Martini, and Sawyer X.

32.4.1 PUMPKIN?

[from Porting/pumpkin.pod in the Perl source code distribution]

Chip Salzenberg gets credit for that, with a nod to his cow orker, David Croy. We had passed around various names (baton, token, hot potato) but none caught on. Then, Chip asked:

[begin quote]

```
Who has the patch pumpkin?
```

To explain: David Croy once told me that at a previous job, there was one tape drive and multiple systems that used it for backups. But instead of some high-tech exclusion software, they used a low-tech method to prevent multiple simultaneous backups: a stuffed pumpkin. No one was allowed to make backups unless they had the "backup pumpkin".

[end quote]

The name has stuck. The holder of the pumpkin is sometimes called the pumpking (keeping the source afloat?) or the pumpkineer (pulling the strings?).

32.5 THE RECORDS

Pump- king	Release	Date	Notes (by no means comprehensive, see Changes* for details)
Larry	0	Classified.	Don’t ask.
Larry	1.000	1987-Dec-18	
	1.001..10	1988-Jan-30	
	1.011..14	1988-Feb-02	
Schwern	1.0.15	2002-Dec-18	Modernization
Richard	1.0_16	2003-Dec-18	
Larry	2.000	1988-Jun-05	
	2.001	1988-Jun-28	
Larry	3.000	1989-Oct-18	
	3.001	1989-Oct-26	
	3.002..4	1989-Nov-11	
	3.005	1989-Nov-18	
	3.006..8	1989-Dec-22	
	3.009..13	1990-Mar-02	
	3.014	1990-Mar-13	
	3.015	1990-Mar-14	
	3.016..18	1990-Mar-28	
	3.019..27	1990-Aug-10	User subs.
	3.028	1990-Aug-14	
	3.029..36	1990-Oct-17	
	3.037	1990-Oct-20	
	3.040	1990-Nov-10	
	3.041	1990-Nov-13	
	3.042..43	1991-Jan-??	
	3.044	1991-Jan-12	
Larry	4.000	1991-Mar-21	
	4.001..3	1991-Apr-12	

	4.004..9	1991-Jun-07	
	4.010	1991-Jun-10	
	4.011..18	1991-Nov-05	
	4.019	1991-Nov-11	Stable.
	4.020..33	1992-Jun-08	
	4.034	1992-Jun-11	
	4.035	1992-Jun-23	
Larry	4.036	1993-Feb-05	Very stable.
	5.000alpha1	1993-Jul-31	
	5.000alpha2	1993-Aug-16	
	5.000alpha3	1993-Oct-10	
	5.000alpha4	1993-???-??	
	5.000alpha5	1993-???-??	
	5.000alpha6	1994-Mar-18	
	5.000alpha7	1994-Mar-25	
Andy	5.000alpha8	1994-Apr-04	
Larry	5.000alpha9	1994-May-05	ext appears.
	5.000alpha10	1994-Jun-11	
	5.000alpha11	1994-Jul-01	
Andy	5.000a11a	1994-Jul-07	To fit 14.
	5.000a11b	1994-Jul-14	
	5.000a11c	1994-Jul-19	
	5.000a11d	1994-Jul-22	
Larry	5.000alpha12	1994-Aug-04	
Andy	5.000a12a	1994-Aug-08	
	5.000a12b	1994-Aug-15	
	5.000a12c	1994-Aug-22	
	5.000a12d	1994-Aug-22	
	5.000a12e	1994-Aug-22	
	5.000a12f	1994-Aug-24	
	5.000a12g	1994-Aug-24	
	5.000a12h	1994-Aug-24	
Larry	5.000beta1	1994-Aug-30	
Andy	5.000b1a	1994-Sep-06	
Larry	5.000beta2	1994-Sep-14	Core slushified.
Andy	5.000b2a	1994-Sep-14	
	5.000b2b	1994-Sep-17	
	5.000b2c	1994-Sep-17	
Larry	5.000beta3	1994-Sep-??	
Andy	5.000b3a	1994-Sep-18	
	5.000b3b	1994-Sep-22	
	5.000b3c	1994-Sep-23	
	5.000b3d	1994-Sep-27	
	5.000b3e	1994-Sep-28	
	5.000b3f	1994-Sep-30	
	5.000b3g	1994-Oct-04	

```
Andy     5.000b3h       1994-Oct-07
Larry?   5.000gamma     1994-Oct-13?

Larry   5.000           1994-Oct-17

Andy     5.000a         1994-Dec-19
         5.000b         1995-Jan-18
         5.000c         1995-Jan-18
         5.000d         1995-Jan-18
         5.000e         1995-Jan-18
         5.000f         1995-Jan-18
         5.000g         1995-Jan-18
         5.000h         1995-Jan-18
         5.000i         1995-Jan-26
         5.000j         1995-Feb-07
         5.000k         1995-Feb-11
         5.000l         1995-Feb-21
         5.000m         1995-Feb-28
         5.000n         1995-Mar-07
         5.000o         1995-Mar-13?

Larry   5.001           1995-Mar-13

Andy     5.001a         1995-Mar-15
         5.001b         1995-Mar-31
         5.001c         1995-Apr-07
         5.001d         1995-Apr-14
         5.001e         1995-Apr-18     Stable.
         5.001f         1995-May-31
         5.001g         1995-May-25
         5.001h         1995-May-25
         5.001i         1995-May-30
         5.001j         1995-Jun-05
         5.001k         1995-Jun-06
         5.001l         1995-Jun-06     Stable.
         5.001m         1995-Jul-02     Very stable.
         5.001n         1995-Oct-31     Very unstable.
         5.002beta1     1995-Nov-21
         5.002b1a       1995-Dec-04
         5.002b1b       1995-Dec-04
         5.002b1c       1995-Dec-04
         5.002b1d       1995-Dec-04
         5.002b1e       1995-Dec-08
         5.002b1f       1995-Dec-08
Tom      5.002b1g       1995-Dec-21     Doc release.
Andy     5.002b1h       1996-Jan-05
         5.002b2        1996-Jan-14
```

```
Larry    5.002b3       1996-Feb-02
Andy     5.002gamma    1996-Feb-11
Larry    5.002delta    1996-Feb-27

Larry   5.002          1996-Feb-29     Prototypes.

Charles  5.002_01      1996-Mar-25

        5.003          1996-Jun-25     Security release.

         5.003_01      1996-Jul-31
Nick     5.003_02      1996-Aug-10
Andy     5.003_03      1996-Aug-28
         5.003_04      1996-Sep-02
         5.003_05      1996-Sep-12
         5.003_06      1996-Oct-07
         5.003_07      1996-Oct-10
Chip     5.003_08      1996-Nov-19
         5.003_09      1996-Nov-26
         5.003_10      1996-Nov-29
         5.003_11      1996-Dec-06
         5.003_12      1996-Dec-19
         5.003_13      1996-Dec-20
         5.003_14      1996-Dec-23
         5.003_15      1996-Dec-23
         5.003_16      1996-Dec-24
         5.003_17      1996-Dec-27
         5.003_18      1996-Dec-31
         5.003_19      1997-Jan-04
         5.003_20      1997-Jan-07
         5.003_21      1997-Jan-15
         5.003_22      1997-Jan-16
         5.003_23      1997-Jan-25
         5.003_24      1997-Jan-29
         5.003_25      1997-Feb-04
         5.003_26      1997-Feb-10
         5.003_27      1997-Feb-18
         5.003_28      1997-Feb-21
         5.003_90      1997-Feb-25     Ramping up to the 5.004 release.
         5.003_91      1997-Mar-01
         5.003_92      1997-Mar-06
         5.003_93      1997-Mar-10
         5.003_94      1997-Mar-22
         5.003_95      1997-Mar-25
         5.003_96      1997-Apr-01
         5.003_97      1997-Apr-03     Fairly widely used.
         5.003_97a     1997-Apr-05
         5.003_97b     1997-Apr-08
         5.003_97c     1997-Apr-10
         5.003_97d     1997-Apr-13
         5.003_97e     1997-Apr-15
         5.003_97f     1997-Apr-17
         5.003_97g     1997-Apr-18
         5.003_97h     1997-Apr-24
         5.003_97i     1997-Apr-25
         5.003_97j     1997-Apr-28
         5.003_98      1997-Apr-30
         5.003_99      1997-May-01
         5.003_99a     1997-May-09
         p54rc1        1997-May-12     Release Candidates.
         p54rc2        1997-May-14

Chip    5.004          1997-May-15     A major maintenance release.

Tim      5.004_01-t1   1997-???-??     The 5.004 maintenance track.
         5.004_01-t2   1997-Jun-11     aka perl5.004m1t2
         5.004_01      1997-Jun-13
         5.004_01_01   1997-Jul-29     aka perl5.004m2t1
         5.004_01_02   1997-Aug-01     aka perl5.004m2t2
         5.004_01_03   1997-Aug-05     aka perl5.004m2t3
         5.004_02      1997-Aug-07
         5.004_02_01   1997-Aug-12     aka perl5.004m3t1
         5.004_03-t2   1997-Aug-13     aka perl5.004m3t2
         5.004_03      1997-Sep-05
         5.004_04-t1   1997-Sep-19     aka perl5.004m4t1
         5.004_04-t2   1997-Sep-23     aka perl5.004m4t2
         5.004_04-t3   1997-Oct-10     aka perl5.004m4t3
         5.004_04-t4   1997-Oct-14     aka perl5.004m4t4
         5.004_04      1997-Oct-15
         5.004_04-m1   1998-Mar-04     (5.004m5t1) Maint. trials for 5.004_05.
         5.004_04-m2   1998-May-01
         5.004_04-m3   1998-May-15
         5.004_04-m4   1998-May-19
         5.004_05-MT5  1998-Jul-21
         5.004_05-MT6  1998-Oct-09
         5.004_05-MT7  1998-Nov-22
         5.004_05-MT8  1998-Dec-03
Chip     5.004_05-MT9  1999-Apr-26
         5.004_05      1999-Apr-29

Malcolm  5.004_50      1997-Sep-09     The 5.005 development track.
         5.004_51      1997-Oct-02
         5.004_52      1997-Oct-15
         5.004_53      1997-Oct-16
```

	5.004_54	1997-Nov-14	
	5.004_55	1997-Nov-25	
	5.004_56	1997-Dec-18	
	5.004_57	1998-Feb-03	
	5.004_58	1998-Feb-06	
	5.004_59	1998-Feb-13	
	5.004_60	1998-Feb-20	
	5.004_61	1998-Feb-27	
	5.004_62	1998-Mar-06	
	5.004_63	1998-Mar-17	
	5.004_64	1998-Apr-03	
	5.004_65	1998-May-15	
	5.004_66	1998-May-29	
Sarathy	5.004_67	1998-Jun-15	
	5.004_68	1998-Jun-23	
	5.004_69	1998-Jun-29	
	5.004_70	1998-Jul-06	
	5.004_71	1998-Jul-09	
	5.004_72	1998-Jul-12	
	5.004_73	1998-Jul-13	
	5.004_74	1998-Jul-14	5.005 beta candidate.
	5.004_75	1998-Jul-15	5.005 beta1.
	5.004_76	1998-Jul-21	5.005 beta2.
Sarathy	5.005	1998-Jul-22	Oneperl.
Sarathy	5.005_01	1998-Jul-27	The 5.005 maintenance track.
	5.005_02-T1	1998-Aug-02	
	5.005_02-T2	1998-Aug-05	
	5.005_02	1998-Aug-08	
Graham	5.005_03-MT1	1998-Nov-30	
	5.005_03-MT2	1999-Jan-04	
	5.005_03-MT3	1999-Jan-17	
	5.005_03-MT4	1999-Jan-26	
	5.005_03-MT5	1999-Jan-28	
	5.005_03-MT6	1999-Mar-05	
	5.005_03	1999-Mar-28	
Leon	5.005_04-RC1	2004-Feb-05	
	5.005_04-RC2	2004-Feb-18	
	5.005_04	2004-Feb-23	
	5.005_05-RC1	2009-Feb-16	
Sarathy	5.005_50	1998-Jul-26	The 5.6 development track.
	5.005_51	1998-Aug-10	
	5.005_52	1998-Sep-25	
	5.005_53	1998-Oct-31	
	5.005_54	1998-Nov-30	

```
          5.005_55      1999-Feb-16
          5.005_56      1999-Mar-01
          5.005_57      1999-May-25
          5.005_58      1999-Jul-27
          5.005_59      1999-Aug-02
          5.005_60      1999-Aug-02
          5.005_61      1999-Aug-20
          5.005_62      1999-Oct-15
          5.005_63      1999-Dec-09
          5.5.640       2000-Feb-02
          5.5.650       2000-Feb-08     beta1
          5.5.660       2000-Feb-22     beta2
          5.5.670       2000-Feb-29     beta3
          5.6.0-RC1     2000-Mar-09     Release candidate 1.
          5.6.0-RC2     2000-Mar-14     Release candidate 2.
          5.6.0-RC3     2000-Mar-21     Release candidate 3.

 Sarathy  5.6.0         2000-Mar-22

 Sarathy  5.6.1-TRIAL1  2000-Dec-18     The 5.6 maintenance track.
          5.6.1-TRIAL2  2001-Jan-31
          5.6.1-TRIAL3  2001-Mar-19
          5.6.1-foolish 2001-Apr-01     The "fools-gold" release.
          5.6.1         2001-Apr-08
 Rafael   5.6.2-RC1     2003-Nov-08
          5.6.2         2003-Nov-15     Fix new build issues

 Jarkko   5.7.0         2000-Sep-02     The 5.7 track: Development.
          5.7.1         2001-Apr-09
          5.7.2         2001-Jul-13     Virtual release candidate 0.
          5.7.3         2002-Mar-05
          5.8.0-RC1     2002-Jun-01
          5.8.0-RC2     2002-Jun-21
          5.8.0-RC3     2002-Jul-13

 Jarkko   5.8.0         2002-Jul-18

 Jarkko   5.8.1-RC1     2003-Jul-10     The 5.8 maintenance track
          5.8.1-RC2     2003-Jul-11
          5.8.1-RC3     2003-Jul-30
          5.8.1-RC4     2003-Aug-01
          5.8.1-RC5     2003-Sep-22
          5.8.1         2003-Sep-25
 Nicholas 5.8.2-RC1     2003-Oct-27
          5.8.2-RC2     2003-Nov-03
          5.8.2         2003-Nov-05
          5.8.3-RC1     2004-Jan-07
```

```
         5.8.3          2004-Jan-14
         5.8.4-RC1      2004-Apr-05
         5.8.4-RC2      2004-Apr-15
         5.8.4          2004-Apr-21
         5.8.5-RC1      2004-Jul-06
         5.8.5-RC2      2004-Jul-08
         5.8.5          2004-Jul-19
         5.8.6-RC1      2004-Nov-11
         5.8.6          2004-Nov-27
         5.8.7-RC1      2005-May-18
         5.8.7          2005-May-30
         5.8.8-RC1      2006-Jan-20
         5.8.8          2006-Jan-31
         5.8.9-RC1      2008-Nov-10
         5.8.9-RC2      2008-Dec-06
         5.8.9          2008-Dec-14

Hugo     5.9.0          2003-Oct-27     The 5.9 development track
Rafael   5.9.1          2004-Mar-16
         5.9.2          2005-Apr-01
         5.9.3          2006-Jan-28
         5.9.4          2006-Aug-15
         5.9.5          2007-Jul-07
         5.10.0-RC1     2007-Nov-17
         5.10.0-RC2     2007-Nov-25

Rafael   5.10.0         2007-Dec-18

David M  5.10.1-RC1     2009-Aug-06     The 5.10 maintenance track
         5.10.1-RC2     2009-Aug-18
         5.10.1         2009-Aug-22

Jesse    5.11.0         2009-Oct-02     The 5.11 development track
         5.11.1         2009-Oct-20
Leon     5.11.2         2009-Nov-20
Jesse    5.11.3         2009-Dec-20
Ricardo  5.11.4         2010-Jan-20
Steve    5.11.5         2010-Feb-20
Jesse    5.12.0-RC0     2010-Mar-21
         5.12.0-RC1     2010-Mar-29
         5.12.0-RC2     2010-Apr-01
         5.12.0-RC3     2010-Apr-02
         5.12.0-RC4     2010-Apr-06
         5.12.0-RC5     2010-Apr-09

Jesse    5.12.0         2010-Apr-12
Jesse    5.12.1-RC2    2010-May-13     The 5.12 maintenance track
         5.12.1-RC1    2010-May-09
         5.12.1        2010-May-16
         5.12.2-RC2    2010-Aug-31
         5.12.2        2010-Sep-06
Ricardo  5.12.3-RC1    2011-Jan-09
Ricardo  5.12.3-RC2    2011-Jan-14
Ricardo  5.12.3-RC3    2011-Jan-17
Ricardo  5.12.3        2011-Jan-21
Leon     5.12.4-RC1    2011-Jun-08
Leon     5.12.4        2011-Jun-20
Dominic  5.12.5        2012-Nov-10

Leon     5.13.0        2010-Apr-20     The 5.13 development track
Ricardo  5.13.1        2010-May-20
Matt     5.13.2        2010-Jun-22
David G  5.13.3        2010-Jul-20
Florian  5.13.4        2010-Aug-20
Steve    5.13.5        2010-Sep-19
Miyagawa 5.13.6        2010-Oct-20
BinGOs   5.13.7        2010-Nov-20
Zefram   5.13.8        2010-Dec-20
Jesse    5.13.9        2011-Jan-20
Ævar     5.13.10       2011-Feb-20
Florian  5.13.11       2011-Mar-20
Jesse    5.14.0RC1     2011-Apr-20
Jesse    5.14.0RC2     2011-May-04
Jesse    5.14.0RC3     2011-May-11

Jesse    5.14.0        2011-May-14     The 5.14 maintenance track
Jesse    5.14.1        2011-Jun-16
Florian  5.14.2-RC1    2011-Sep-19
         5.14.2        2011-Sep-26
Dominic  5.14.3        2012-Oct-12
David M  5.14.4-RC1    2013-Mar-05
David M  5.14.4-RC2    2013-Mar-07
David M  5.14.4        2013-Mar-10

David G  5.15.0        2011-Jun-20     The 5.15 development track
Zefram   5.15.1        2011-Jul-20
Ricardo  5.15.2        2011-Aug-20
Stevan   5.15.3        2011-Sep-20
Florian  5.15.4        2011-Oct-20
Steve    5.15.5        2011-Nov-20
Dave R   5.15.6        2011-Dec-20
BinGOs   5.15.7        2012-Jan-20
Max M    5.15.8        2012-Feb-20
```

```
Abigail  5.15.9        2012-Mar-20
Ricardo  5.16.0-RC0    2012-May-10
Ricardo  5.16.0-RC1    2012-May-14
Ricardo  5.16.0-RC2    2012-May-15

Ricardo  5.16.0        2012-May-20     The 5.16 maintenance track
Ricardo  5.16.1        2012-Aug-08
Ricardo  5.16.2        2012-Nov-01
Ricardo  5.16.3-RC1    2013-Mar-06
Ricardo  5.16.3        2013-Mar-11

Zefram   5.17.0        2012-May-26     The 5.17 development track
Jesse L  5.17.1        2012-Jun-20
TonyC    5.17.2        2012-Jul-20
Steve    5.17.3        2012-Aug-20
Florian  5.17.4        2012-Sep-20
Florian  5.17.5        2012-Oct-20
Ricardo  5.17.6        2012-Nov-20
Dave R   5.17.7        2012-Dec-18
Aaron    5.17.8        2013-Jan-20
BinGOs   5.17.9        2013-Feb-20
Max M    5.17.10       2013-Mar-21
Ricardo  5.17.11       2013-Apr-20

Ricardo  5.18.0-RC1    2013-May-11     The 5.18 maintenance track
Ricardo  5.18.0-RC2    2013-May-12
Ricardo  5.18.0-RC3    2013-May-13
Ricardo  5.18.0-RC4    2013-May-15
Ricardo  5.18.0        2013-May-18
Ricardo  5.18.1-RC1    2013-Aug-01
Ricardo  5.18.1-RC2    2013-Aug-03
Ricardo  5.18.1-RC3    2013-Aug-08
Ricardo  5.18.1        2013-Aug-12
Ricardo  5.18.2        2014-Jan-06
Ricardo  5.18.3-RC1    2014-Sep-17
Ricardo  5.18.3-RC2    2014-Sep-27
Ricardo  5.18.3        2014-Oct-01
Ricardo  5.18.4        2014-Oct-01

Ricardo   5.19.0       2013-May-20     The 5.19 development track
David G   5.19.1       2013-Jun-21
Aristotle 5.19.2       2013-Jul-22
Steve     5.19.3       2013-Aug-20
Steve     5.19.4       2013-Sep-20
Steve     5.19.5       2013-Oct-20
BinGOs    5.19.6       2013-Nov-20
Abigail   5.19.7       2013-Dec-20
```

```
Ricardo   5.19.8       2014-Jan-20
TonyC     5.19.9       2014-Feb-20
Aaron     5.19.10      2014-Mar-20
Steve     5.19.11      2014-Apr-20

Ricardo   5.20.0-RC1   2014-May-16     The 5.20 maintenance track
Ricardo   5.20.0       2014-May-27
Steve     5.20.1-RC1   2014-Aug-25
Steve     5.20.1-RC2   2014-Sep-07
Steve     5.20.1       2014-Sep-14
Steve     5.20.2-RC1   2015-Jan-31
Steve     5.20.2       2015-Feb-14

Ricardo   5.21.0       2014-May-27     The 5.21 development track
Matthew H 5.21.1       2014-Jun-20
Abigail   5.21.2       2014-Jul-20
Peter     5.21.3       2014-Aug-20
Steve     5.21.4       2014-Sep-20
Abigail   5.21.5       2014-Oct-20
BinGOs    5.21.6       2014-Nov-20
Max M     5.21.7       2014-Dec-20
Matthew H 5.21.8       2015-Jan-20
Sawyer X  5.21.9       2015-Feb-20
Steve     5.21.10      2015-Mar-20
Steve     5.21.11      2015-Apr-20

Ricardo   5.22.0-RC1   2015-May-19     The 5.22 maintenance track
Ricardo   5.22.0-RC2   2015-May-21
Ricardo   5.22.0       2015-Jun-01
```

32.5.1 SELECTED RELEASE SIZES

For example the notation "core: 212 29" in the release 1.000 means that it had in the core 212 kilobytes, in 29 files. The "core".."doc" are explained below.

```
release         core       lib         ext        t         doc
======================================================================

1.000           212  29      -   -      -   -     38   51     62   3
1.014           219  29      -   -      -   -     39   52     68   4
2.000           309  31      2   3      -   -     55   57     92   4
2.001           312  31      2   3      -   -     55   57     94   4
3.000           508  36     24  11      -   -     79   73    156   5
3.044           645  37     61  20      -   -     90   74    190   6
4.000           635  37     59  20      -   -     91   75    198   4
4.019           680  37     85  29      -   -     98   76    199   4
4.036           709  37     89  30      -   -     98   76    208   5
5.000alpha2     785  50    114  32      -   -    112   86    209   5
```

5.000alpha3	801	50	117	33	-	-	121	87	209	5
5.000alpha9	1022	56	149	43	116	29	125	90	217	6
5.000a12h	978	49	140	49	205	46	152	97	228	9
5.000b3h	1035	53	232	70	216	38	162	94	218	21
5.000	1038	53	250	76	216	38	154	92	536	62
5.001m	1071	54	388	82	240	38	159	95	544	29
5.002	1121	54	661	101	287	43	155	94	847	35
5.003	1129	54	680	102	291	43	166	100	853	35
5.003_07	1231	60	748	106	396	53	213	137	976	39
5.004	1351	60	1230	136	408	51	355	161	1587	55
5.004_01	1356	60	1258	138	410	51	358	161	1587	55
5.004_04	1375	60	1294	139	413	51	394	162	1629	55
5.004_05	1463	60	1435	150	394	50	445	175	1855	59
5.004_51	1401	61	1260	140	413	53	358	162	1594	56
5.004_53	1422	62	1295	141	438	70	394	162	1637	56
5.004_56	1501	66	1301	140	447	74	408	165	1648	57
5.004_59	1555	72	1317	142	448	74	424	171	1678	58
5.004_62	1602	77	1327	144	629	92	428	173	1674	58
5.004_65	1626	77	1358	146	615	92	446	179	1698	60
5.004_68	1856	74	1382	152	619	92	463	187	1784	60
5.004_70	1863	75	1456	154	675	92	494	194	1809	60
5.004_73	1874	76	1467	152	762	102	506	196	1883	61
5.004_75	1877	76	1467	152	770	103	508	196	1896	62
5.005	1896	76	1469	152	795	103	509	197	1945	63
5.005_03	1936	77	1541	153	813	104	551	201	2176	72
5.005_50	1969	78	1842	301	795	103	514	198	1948	63
5.005_53	1999	79	1885	303	806	104	602	224	2002	67
5.005_56	2086	79	1970	307	866	113	672	238	2221	75
5.6.0	2820	79	2626	364	1096	129	863	280	2840	93
5.6.1	2946	78	2921	430	1171	132	1024	304	3330	102
5.6.2	2947	78	3143	451	1247	127	1303	387	3406	102
5.7.0	2977	80	2801	425	1250	132	975	307	3206	100
5.7.1	3351	84	3442	455	1944	167	1334	357	3698	124
5.7.2	3491	87	4858	618	3290	298	1598	449	3910	139
5.7.3	3299	85	4295	537	2196	300	2176	626	4171	120
5.8.0	3489	87	4533	585	2437	331	2588	726	4368	125
5.8.1	3674	90	5104	623	2604	353	2983	836	4625	134
5.8.2	3633	90	5111	623	2623	357	3019	848	4634	135
5.8.3	3625	90	5141	624	2660	363	3083	869	4669	136
5.8.4	3653	90	5170	634	2684	368	3148	885	4689	137
5.8.5	3664	90	4260	303	2707	369	3208	898	4689	138
5.8.6	3690	90	4271	303	3141	396	3411	925	4709	139
5.8.7	3788	90	4322	307	3297	401	3485	964	4744	141
5.8.8	3895	90	4357	314	3409	431	3622	1017	4979	144
5.8.9	4132	93	5508	330	3826	529	4364	1234	5348	152
5.9.0	3657	90	4951	626	2603	354	3011	841	4609	135
5.9.1	3580	90	5196	634	2665	367	3186	889	4725	138

```
5.9.2          3863  90   4654 312   3283  403   3551  973   4800 142
5.9.3          4096  91   5318 381   4806  597   4272 1214   5139 147
5.9.4          4393  94   5718 415   4578  642   4646 1310   5335 153
5.9.5          4681  96   6849 479   4827  671   5155 1490   5572 159
5.10.0         4710  97   7050 486   4899  673   5275 1503   5673 160
5.10.1         4858  98   7440 519   6195  921   6147 1751   5151 163
5.12.0         4999 100   1146 121  15227 2176   6400 1843   5342 168
5.12.1         5000 100   1146 121  15283 2178   6407 1846   5354 169
5.12.2         5003 100   1146 121  15404 2178   6413 1846   5376 170
5.12.3         5004 100   1146 121  15529 2180   6417 1848   5391 171
5.14.0         5328 104   1100 114  17779 2479   7697 2130   5871 188
5.16.0         5562 109   1077  80  20504 2702   8750 2375   4815 152
5.18.0         5892 113   1088  79  20077 2760   9365 2439   4943 154
5.20.0         6243 115   1187  75  19499 2701   9620 2457   5145 159
5.22.0         7819 115   1284  77  19121 2635   9772 2434   5615 176
```

The "core"..."doc" mean the following files from the Perl source code distribution. The glob notation ** means recursively, (.) means regular files.

```
core   *.[hcy]
lib    lib/**/*.p[ml]
ext    ext/**/*.{[hcyt],xs,pm} (for -5.10.1) or
       {dist,ext,cpan}/**/*.{[hcyt],xs,pm} (for 5.12.0-)
t      t/**/*(.) (for 1-5.005_56) or **/*.t (for 5.6.0-5.7.3)
doc    {README*,INSTALL,*[_.]man{,.?},pod/**/*.pod}
```

Here are some statistics for the other subdirectories and one file in the Perl source distribution for somewhat more selected releases.

```
======================================================================
  Legend:  kB   #

                 1.014      2.001      3.044

Configure      31    1    37    1    62    1
eg              -    -    34   28    47   39
h2pl            -    -     -    -    12   12
msdos           -    -     -    -    41   13
os2             -    -     -    -    63   22
usub            -    -     -    -    21   16
x2p           103   17   104   17   137   17

======================================================================

                 4.000      4.019      4.036

atarist         -    -     -    -   113   31
Configure      73    1    83    1    86    1
eg             47   39    47   39    47   39
emacs          67    4    67    4    67    4
```

```
h2pl           12   12    12   12    12   12
hints           -    -     5   42    11   56
msdos          57   15    58   15    60   15
os2            81   29    81   29   113   31
usub           25    7    43    8    43    8
x2p           147   18   152   19   154   19

======================================================================

              5.000a2  5.000a12h   5.000b3h      5.000     5.001m

apollo          8    3     8    3     8    3     8    3     8    3
atarist       113   31   113   31     -    -     -    -     -    -
bench           -    -     0    1     -    -     -    -     -    -
Bugs            2    5    26    1     -    -     -    -     -    -
dlperl         40    5     -    -     -    -     -    -     -    -
do            127   71     -    -     -    -     -    -     -    -
Configure       -    -   153    1   159    1   160    1   180    1
Doc             -    -    26    1    75    7    11    1    11    1
eg             79   58    53   44    51   43    54   44    54   44
emacs          67    4   104    6   104    6   104    1   104    6
h2pl           12   12    12   12    12   12    12   12    12   12
hints          11   56    12   46    18   48    18   48    44   56
msdos          60   15    60   15     -    -     -    -     -    -
os2           113   31   113   31     -    -     -    -     -    -
U               -    -    62    8   112   42     -    -     -    -
usub           43    8     -    -     -    -     -    -     -    -
vms             -    -    80    7   123    9   184   15   304   20
x2p           171   22   171   21   162   20   162   20   279   20

======================================================================

                5.002      5.003   5.003_07

Configure     201    1   201    1   217    1
eg             54   44    54   44    54   44
emacs         108    1   108    1   143    1
h2pl           12   12    12   12    12   12
hints          73   59    77   60    90   62
os2            84   17    56   10   117   42
plan9           -    -     -    -    79   15
Porting         -    -     -    -    51    1
utils          87    7    88    7    97    7
vms           500   24   475   26   505   27
x2p           280   20   280   20   280   19

======================================================================
                 5.004   5.004_04   5.004_62   5.004_65   5.004_68

beos            -    -     -    -     -    -      1   1      1   1
Configure     225    1   225    1   240    1    248   1    256   1
cygwin32       23    5    23    5    23    5     24   5     24   5
djgpp           -    -     -    -    14    5     14   5     14   5
eg             81   62    81   62    81   62     81  62     81  62
emacs         194    1   204    1   212    2    212   2    212   2
h2pl           12   12    12   12    12   12     12  12     12  12
hints         129   69   132   71   144   72    151  74    155  74
os2           121   42   127   42   127   44    129  44    129  44
plan9          82   15    82   15    82   15     82  15     82  15
Porting        94    2   109    4   203    6    234   8    241   9
qnx             1    2     1    2     1    2      1   2      1   2
utils         112    8   118    8   124    8    156   9    159   9
vms           518   34   524   34   538   34    569  34    569  34
win32         285   33   378   36   470   39    493  39    575  41
x2p           281   19   281   19   281   19    282  19    281  19

======================================================================

              5.004_70   5.004_73   5.004_75      5.005   5.005_03

apollo          -    -     -    -     -    -     -    -      0   1
beos            1    1     1    1     1    1     1    1      1   1
Configure     256    1   256    1   264    1   264    1    270   1
cygwin32       24    5    24    5    24    5    24    5     24   5
djgpp          14    5    14    5    14    5    14    5     15   5
eg             86   65    86   65    86   65    86   65     86  65
emacs         262    2   262    2   262    2   262    2    274   2
h2pl           12   12    12   12    12   12    12   12     12  12
hints         157   74   157   74   159   74   160   74    179  77
mint            -    -     -    -     -    -     -    -      4   7
mpeix           -    -     -    -     5    3     5    3      5   3
os2           129   44   139   44   142   44   143   44    148  44
plan9          82   15    82   15    82   15    82   15     82  15
Porting       241    9   253    9   259   10   264   12    272  13
qnx             1    2     1    2     1    2     1    2      1   2
utils         160    9   160    9   160    9   160    9    164   9
vms           570   34   572   34   573   34   575   34    583  34
vos             -    -     -    -     -    -     -    -    156  10
win32         577   41   585   41   585   41   587   41    600  42
x2p           281   19   281   19   281   19   281   19    281  19

======================================================================
                 5.6.0      5.6.1      5.6.2      5.7.3

apollo          8    3     8    3     8    3     8    3
beos            5    2     5    2     5    2     6    4
Configure     346    1   361    1   363    1   394    1
Cross           -    -     -    -     -    -     4    2
djgpp          19    6    19    6    19    6    21    7
eg            112   71   112   71   112   71     -    -
emacs         303    4   319    4   319    4   319    4
epoc           29    8    35    8    35    8    36    8
h2pl           24   15    24   15    24   15    24   15
hints         242   83   250   84   321   89   272   87
mint           11    9    11    9    11    9    11    9
mpeix           9    4     9    4     9    4     9    4
NetWare         -    -     -    -     -    -   423   57
os2           214   59   224   60   224   60   357   66
plan9          92   17    92   17    92   17    85   15
Porting       361   15   390   16   390   16   425   21
qnx             5    3     5    3     5    3     5    3
utils         228   12   221   11   222   11   267   13
uts             -    -     -    -     -    -    12    3
vmesa          25    4    25    4    25    4    25    4
vms           686   38   627   38   627   38   649   36
vos           227   12   249   15   248   15   281   17
win32         755   41   782   42   801   42  1006   50
x2p           307   20   307   20   307   20   345   20

======================================================================

                 5.8.0      5.8.1      5.8.2      5.8.3      5.8.4

apollo          8    3     8    3     8    3     8    3     8    3
beos            6    4     6    4     6    4     6    4     6    4
Configure     472    1   493    1   493    1   493    1   494    1
Cross           4    2    45   10    45   10    45   10    45   10
djgpp          21    7    21    7    21    7    21    7    21    7
emacs         319    4   329    4   329    4   329    4   329    4
epoc           33    8    33    8    33    8    33    8    33    8
h2pl           24   15    24   15    24   15    24   15    24   15
hints         294   88   321   89   321   89   321   89   348   91
mint           11    9    11    9    11    9    11    9    11    9
mpeix          24    5    25    5    25    5    25    5    25    5
NetWare       488   61   490   61   490   61   490   61   488   61
os2           361   66   445   67   450   67   488   67   488   67
plan9          85   15   325   17   325   17   325   17   321   17
Porting       479   22   537   32   538   32   539   32   538   33
qnx             5    3     5    3     5    3     5    3     5    3
```

```
utils         275   15   258   16   258   16   263   19   263   19
uts            12    3    12    3    12    3    12    3    12    3
vmesa          25    4    25    4    25    4    25    4    25    4
vms           648   36   654   36   654   36   656   36   656   36
vos           330   20   335   20   335   20   335   20   335   20
win32        1062   49  1125   49  1127   49  1126   49  1181   56
x2p           347   20   348   20   348   20   348   20   348   20

======================================================================

                 5.8.5       5.8.6       5.8.7       5.8.8       5.8.9

apollo          8    3     8    3     8    3     8    3     8    3
beos            6    4     6    4     8    4     8    4     8    4
Configure     494    1   494    1   495    1   506    1   520    1
Cross          45   10    45   10    45   10    45   10    46   10
djgpp          21    7    21    7    21    7    21    7    21    7
emacs         329    4   329    4   329    4   329    4   406    4
epoc           33    8    33    8    33    8    34    8    35    8
h2pl           24   15    24   15    24   15    24   15    24   15
hints         350   91   352   91   355   94   360   94   387   99
mint           11    9    11    9    11    9    11    9    11    9
mpeix          25    5    25    5    25    5    49    6    49    6
NetWare       488   61   488   61   488   61   490   61   491   61
os2           488   67   488   67   488   67   488   67   552   70
plan9         321   17   321   17   321   17   322   17   324   17
Porting       538   34   548   35   549   35   564   37   625   41
qnx             5    3     5    3     5    3     5    3     5    3
utils         265   19   265   19   266   19   267   19   281   21
uts            12    3    12    3    12    3    12    3    12    3
vmesa          25    4    25    4    25    4    25    4    25    4
vms           657   36   658   36   662   36   664   36   716   35
vos           335   20   335   20   335   20   336   21   345   22
win32        1183   56  1190   56  1199   56  1219   56  1484   68
x2p           349   20   349   20   349   20   349   19   350   19

======================================================================

                 5.9.0       5.9.1       5.9.2       5.9.3       5.9.4

apollo          8    3     8    3     8    3     8    3     8    3
beos            6    4     6    4     8    4     8    4     8    4
Configure     493    1   493    1   495    1   508    1   512    1
Cross          45   10    45   10    45   10    45   10    46   10
djgpp          21    7    21    7    21    7    21    7    21    7
emacs         329    4   329    4   329    4   329    4   329    4
epoc           33    8    33    8    33    8    34    8    34    8
```

```
h2pl           24   15    24   15    24   15    24   15    24   15
hints         321   89   346   91   355   94   359   94   366   96
mad             -    -     -    -     -    -     -    -   174    6
mint           11    9    11    9    11    9    11    9    11    9
mpeix          25    5    25    5    25    5    49    6    49    6
NetWare       489   61   487   61   487   61   489   61   489   61
os2           444   67   488   67   488   67   488   67   488   67
plan9         325   17   321   17   321   17   322   17   323   17
Porting       537   32   536   33   549   36   564   38   576   38
qnx             5    3     5    3     5    3     5    3     5    3
symbian         -    -     -    -     -    -   293   53   293   53
utils         258   16   263   19   268   20   273   23   275   24
uts            12    3    12    3    12    3    12    3    12    3
vmesa          25    4    25    4    25    4    25    4    25    4
vms           660   36   547   33   553   33   661   33   696   33
vos            11    7    11    7    11    7    11    7    11    7
win32        1120   49  1124   51  1191   56  1209   56  1719   90
x2p           348   20   348   20   349   20   349   19   349   19

======================================================================

                 5.9.5     5.10.0     5.10.1     5.12.0     5.12.1

apollo          8    3     8    3     0    3     0    3     0    3
beos            8    4     8    4     4    4     4    4     4    4
Configure     518    1   518    1   533    1   536    1   536    1
Cross         122   15   122   15   119   15   118   15   118   15
djgpp          21    7    21    7    17    7    17    7    17    7
emacs         329    4   406    4   402    4   402    4   402    4
epoc           34    8    35    8    31    8    31    8    31    8
h2pl           24   15    24   15    12   15    12   15    12   15
hints         377   98   381   98   385  100   368   97   368   97
mad           182    8   182    8   174    8   174    8   174    8
mint           11    9    11    9     3    9     -    -     -    -
mpeix          49    6    49    6    45    6    45    6    45    6
NetWare       489   61   489   61   465   61   466   61   466   61
os2           552   70   552   70   507   70   507   70   507   70
plan9         324   17   324   17   316   17   316   17   316   17
Porting       627   40   632   40   933   53   749   54   749   54
qnx             5    3     5    4     1    4     1    4     1    4
symbian       300   54   300   54   290   54   288   54   288   54
utils         260   26   264   27   268   27   269   27   269   27
uts            12    3    12    3     8    3     8    3     8    3
vmesa          25    4    25    4    21    4    21    4    21    4
vms           690   32   722   32   693   30   645   18   645   18
vos            19    8    19    8    16    8    16    8    16    8
win32        1482   68  1485   68  1497   70  1841   73  1841   73
```

```
x2p           349   19   349   19   345   19   345   19   345   19

======================================================================

                5.12.2     5.12.3      5.14.0     5.16.0       5.18.0

apollo          0    3     0    3      -    -     -    -      -     -
beos            4    4     4    4      5    4     5    4      -     -
Configure     536    1   536    1    539    1   547    1    550     1
Cross         118   15   118   15    118   15   118   15    118    15
djgpp          17    7    17    7     18    7    18    7     18     7
emacs         402    4   402    4      -    -     -    -      -     -
epoc           31    8    31    8     32    8    30    8      -     -
h2pl           12   15    12   15     15   15    15   15     13    15
hints         368   97   368   97    370   96   371   96    354    91
mad           174    8   174    8    176    8   176    8    174     8
mpeix          45    6    45    6     46    6    46    6      -     -
NetWare       466   61   466   61    473   61   472   61    469    61
os2           507   70   507   70    518   70   519   70    510    70
plan9         316   17   316   17    319   17   319   17    318    17
Porting       750   54   750   54    855   60  1093   69   1149    70
qnx             1    4     1    4      2    4     2    4      1     4
symbian       288   54   288   54    292   54   292   54    290    54
utils         269   27   269   27    249   29   245   30    246    31
uts             8    3     8    3      9    3     9    3      -     -
vmesa          21    4    21    4     22    4    22    4      -     -
vms           646   18   644   18    639   17   571   15    564    15
vos            16    8    16    8     17    8     9    7      8     7
win32        1841   73  1841   73   1833   72  1655   67   1157    62
x2p           345   19   345   19    346   19   345   19    344    20

======================================================================

                 5.20.0           5.22.0

Configure    552      1       570      1
Cross        118     15       118     15
djgpp         18      7        17      7
h2pl          13     15        13     15
hints        355     90       356     87
mad          174      8         -      -
NetWare      467     61       466     61
os2          510     70       510     70
plan9        316     17       317     17
Porting     1204     68      1393     71
qnx            1      4         1      4
symbian      290     54       291     54
```

utils	241	27	242	27
vms	538	12	532	12
vos	8	7	8	7
win32	1183	64	1201	64
x2p	341	19	-	-

32.5.2 SELECTED PATCH SIZES

The "diff lines kB" means that for example the patch 5.003_08, to be applied on top of the 5.003_07 (or whatever was before the 5.003_08) added lines for 110 kilobytes, it removed lines for 19 kilobytes, and changed lines for 424 kilobytes. Just the lines themselves are counted, not their context. The "+ - !" become from the diff(1) context diff output format.

Pump-king	Release	Date	diff lines kB		
			+	-	!
Chip	5.003_08	1996-Nov-19	110	19	424
	5.003_09	1996-Nov-26	38	9	248
	5.003_10	1996-Nov-29	29	2	27
	5.003_11	1996-Dec-06	73	12	165
	5.003_12	1996-Dec-19	275	6	436
	5.003_13	1996-Dec-20	95	1	56
	5.003_14	1996-Dec-23	23	7	333
	5.003_15	1996-Dec-23	0	0	1
	5.003_16	1996-Dec-24	12	3	50
	5.003_17	1996-Dec-27	19	1	14
	5.003_18	1996-Dec-31	21	1	32
	5.003_19	1997-Jan-04	80	3	85
	5.003_20	1997-Jan-07	18	1	146
	5.003_21	1997-Jan-15	38	10	221
	5.003_22	1997-Jan-16	4	0	18
	5.003_23	1997-Jan-25	71	15	119
	5.003_24	1997-Jan-29	426	1	20
	5.003_25	1997-Feb-04	21	8	169
	5.003_26	1997-Feb-10	16	1	15
	5.003_27	1997-Feb-18	32	10	38
	5.003_28	1997-Feb-21	58	4	66
	5.003_90	1997-Feb-25	22	2	34
	5.003_91	1997-Mar-01	37	1	39
	5.003_92	1997-Mar-06	16	3	69
	5.003_93	1997-Mar-10	12	3	15
	5.003_94	1997-Mar-22	407	7	200
	5.003_95	1997-Mar-25	41	1	37
	5.003_96	1997-Apr-01	283	5	261
	5.003_97	1997-Apr-03	13	2	34
	5.003_97a	1997-Apr-05	57	1	27

	5.003_97b	1997-Apr-08	14	1	20
	5.003_97c	1997-Apr-10	20	1	16
	5.003_97d	1997-Apr-13	8	0	16
	5.003_97e	1997-Apr-15	15	4	46
	5.003_97f	1997-Apr-17	7	1	33
	5.003_97g	1997-Apr-18	6	1	42
	5.003_97h	1997-Apr-24	23	3	68
	5.003_97i	1997-Apr-25	23	1	31
	5.003_97j	1997-Apr-28	36	1	49
	5.003_98	1997-Apr-30	171	12	539
	5.003_99	1997-May-01	6	0	7
	5.003_99a	1997-May-09	36	2	61
	p54rc1	1997-May-12	8	1	11
	p54rc2	1997-May-14	6	0	40
	5.004	1997-May-15	4	0	4
Tim	5.004_01	1997-Jun-13	222	14	57
	5.004_02	1997-Aug-07	112	16	119
	5.004_03	1997-Sep-05	109	0	17
	5.004_04	1997-Oct-15	66	8	173

32.5.2.1 The patch-free era

In more modern times, named releases don't come as often, and as progress can be followed (nearly) instantly (with rsync, and since late 2008, git) patches between versions are no longer provided. However, that doesn't keep us from calculating how large a patch could have been. Which is shown in the table below. Unless noted otherwise, the size mentioned is the patch to bring version x.y.z to x.y.z+1.

Sarathy	5.6.1	2001-Apr-08	531	44	651	
Rafael	5.6.2	2003-Nov-15	20	11	1819	
Jarkko	5.8.0	2002-Jul-18	1205	31	471	From 5.7.3
	5.8.1	2003-Sep-25	243	102	6162	
Nicholas	5.8.2	2003-Nov-05	10	50	788	
	5.8.3	2004-Jan-14	31	13	360	
	5.8.4	2004-Apr-21	33	8	299	
	5.8.5	2004-Jul-19	11	19	255	
	5.8.6	2004-Nov-27	35	3	192	
	5.8.7	2005-May-30	75	34	778	
	5.8.8	2006-Jan-31	131	42	1251	
	5.8.9	2008-Dec-14	340	132	12988	
Hugo	5.9.0	2003-Oct-27	281	168	7132	From 5.8.0
Rafael	5.9.1	2004-Mar-16	57	250	2107	
	5.9.2	2005-Apr-01	720	57	858	

```
5.9.3         2006-Jan-28    1124   102  1906
5.9.4         2006-Aug-15     896    60   862
5.9.5         2007-Jul-07    1149   128  1062

5.10.0        2007-Dec-18      50    31 13111   From 5.9.5
```

32.6 THE KEEPERS OF THE RECORDS

Jarkko Hietaniemi <jhi@iki.fi>.

Thanks to the collective memory of the Perlfolk. In addition to the Keepers of the Pumpkin also Alan Champion, Mark Dominus, Andreas König, John Macdonald, Matthias Neeracher, Jeff Okamoto, Michael Peppler, Randal Schwartz, and Paul D. Smith sent corrections and additions. Abigail added file and patch size data for the 5.6.0 - 5.10 era.

33 perlinterp

33.1 NAME

perlinterp - An overview of the Perl interpreter

33.2 DESCRIPTION

This document provides an overview of how the Perl interpreter works at the level of C code, along with pointers to the relevant C source code files.

33.3 ELEMENTS OF THE INTERPRETER

The work of the interpreter has two main stages: compiling the code into the internal representation, or bytecode, and then executing it. Section 28.7 [perlguts Compiled code], page 541 explains exactly how the compilation stage happens.

Here is a short breakdown of perl's operation:

33.3.1 Startup

The action begins in `perlmain.c`. (or `miniperlmain.c` for miniperl) This is very high-level code, enough to fit on a single screen, and it resembles the code found in Section 20.1 [perlembed NAME], page 301; most of the real action takes place in `perl.c`

`perlmain.c` is generated by `ExtUtils::Miniperl` from `miniperlmain.c` at make time, so you should make perl to follow this along.

First, `perlmain.c` allocates some memory and constructs a Perl interpreter, along these lines:

```
1 PERL_SYS_INIT3(&argc,&argv,&env);
2
3 if (!PL_do_undump) {
4     my_perl = perl_alloc();
5     if (!my_perl)
6         exit(1);
7     perl_construct(my_perl);
8     PL_perl_destruct_level = 0;
9 }
```

Line 1 is a macro, and its definition is dependent on your operating system. Line 3 references `PL_do_undump`, a global variable - all global variables in Perl start with `PL_`. This tells you whether the current running program was created with the `-u` flag to perl and then `undump`, which means it's going to be false in any sane context.

Line 4 calls a function in `perl.c` to allocate memory for a Perl interpreter. It's quite a simple function, and the guts of it looks like this:

```
my_perl = (PerlInterpreter*)PerlMem_malloc(sizeof(PerlInterpreter));
```

Here you see an example of Perl's system abstraction, which we'll see later: `PerlMem_malloc` is either your system's `malloc`, or Perl's own `malloc` as defined in `malloc.c` if you selected that option at configure time.

Next, in line 7, we construct the interpreter using perl_construct, also in perl.c; this sets up all the special variables that Perl needs, the stacks, and so on.

Now we pass Perl the command line options, and tell it to go:

```
exitstatus = perl_parse(my_perl, xs_init, argc, argv, (char **)NULL);
if (!exitstatus)
    perl_run(my_perl);

exitstatus = perl_destruct(my_perl);

perl_free(my_perl);
```

perl_parse is actually a wrapper around S_parse_body, as defined in perl.c, which processes the command line options, sets up any statically linked XS modules, opens the program and calls yyparse to parse it.

33.3.2 Parsing

The aim of this stage is to take the Perl source, and turn it into an op tree. We'll see what one of those looks like later. Strictly speaking, there's three things going on here.

yyparse, the parser, lives in perly.c, although you're better off reading the original YACC input in perly.y. (Yes, Virginia, there **is** a YACC grammar for Perl!) The job of the parser is to take your code and "understand" it, splitting it into sentences, deciding which operands go with which operators and so on.

The parser is nobly assisted by the lexer, which chunks up your input into tokens, and decides what type of thing each token is: a variable name, an operator, a bareword, a subroutine, a core function, and so on. The main point of entry to the lexer is yylex, and that and its associated routines can be found in toke.c. Perl isn't much like other computer languages; it's highly context sensitive at times, it can be tricky to work out what sort of token something is, or where a token ends. As such, there's a lot of interplay between the tokeniser and the parser, which can get pretty frightening if you're not used to it.

As the parser understands a Perl program, it builds up a tree of operations for the interpreter to perform during execution. The routines which construct and link together the various operations are to be found in op.c, and will be examined later.

33.3.3 Optimization

Now the parsing stage is complete, and the finished tree represents the operations that the Perl interpreter needs to perform to execute our program. Next, Perl does a dry run over the tree looking for optimisations: constant expressions such as 3 + 4 will be computed now, and the optimizer will also see if any multiple operations can be replaced with a single one. For instance, to fetch the variable $foo, instead of grabbing the glob *foo and looking at the scalar component, the optimizer fiddles the op tree to use a function which directly looks up the scalar in question. The main optimizer is peep in op.c, and many ops have their own optimizing functions.

33.3.4 Running

Now we're finally ready to go: we have compiled Perl byte code, and all that's left to do is run it. The actual execution is done by the runops_standard function in run.c; more specifically, it's done by these three innocent looking lines:

```
while ((PL_op = PL_op->op_ppaddr(aTHX))) {
    PERL_ASYNC_CHECK();
}
```

You may be more comfortable with the Perl version of that:

```
PERL_ASYNC_CHECK() while $Perl::op = &{$Perl::op->{function}};
```

Well, maybe not. Anyway, each op contains a function pointer, which stipulates the function which will actually carry out the operation. This function will return the next op in the sequence - this allows for things like `if` which choose the next op dynamically at run time. The `PERL_ASYNC_CHECK` makes sure that things like signals interrupt execution if required.

The actual functions called are known as PP code, and they're spread between four files: `pp_hot.c` contains the "hot" code, which is most often used and highly optimized, `pp_sys.c` contains all the system-specific functions, `pp_ctl.c` contains the functions which implement control structures (`if`, `while` and the like) and `pp.c` contains everything else. These are, if you like, the C code for Perl's built-in functions and operators.

Note that each `pp_` function is expected to return a pointer to the next op. Calls to perl subs (and eval blocks) are handled within the same runops loop, and do not consume extra space on the C stack. For example, `pp_entersub` and `pp_entertry` just push a `CxSUB` or `CxEVAL` block struct onto the context stack which contain the address of the op following the sub call or eval. They then return the first op of that sub or eval block, and so execution continues of that sub or block. Later, a `pp_leavesub` or `pp_leavetry` op pops the `CxSUB` or `CxEVAL`, retrieves the return op from it, and returns it.

33.3.5 Exception handing

Perl's exception handing (i.e. `die` etc.) is built on top of the low-level `setjmp()`/`longjmp()` C-library functions. These basically provide a way to capture the current PC and SP registers and later restore them; i.e. a `longjmp()` continues at the point in code where a previous `setjmp()` was done, with anything further up on the C stack being lost. This is why code should always save values using `SAVE_FOO` rather than in auto variables.

The perl core wraps `setjmp()` etc in the macros `JMPENV_PUSH` and `JMPENV_JUMP`. The basic rule of perl exceptions is that `exit`, and `die` (in the absence of `eval`) perform a `JMPENV_JUMP(2)`, while `die` within `eval` does a `JMPENV_JUMP(3)`.

At entry points to perl, such as `perl_parse()`, `perl_run()` and `call_sv(cv, G_EVAL)` each does a `JMPENV_PUSH`, then enter a runops loop or whatever, and handle possible exception returns. For a 2 return, final cleanup is performed, such as popping stacks and calling `CHECK` or `END` blocks. Amongst other things, this is how scope cleanup still occurs during an `exit`.

If a `die` can find a `CxEVAL` block on the context stack, then the stack is popped to that level and the return op in that block is assigned to `PL_restartop`; then a `JMPENV_JUMP(3)` is performed. This normally passes control back to the guard. In the case of `perl_run` and `call_sv`, a non-null `PL_restartop` triggers re-entry to the runops loop. The is the normal way that `die` or `croak` is handled within an `eval`.

Sometimes ops are executed within an inner runops loop, such as tie, sort or overload code. In this case, something like

```
sub FETCH { eval { die } }
```

would cause a longjmp right back to the guard in `perl_run`, popping both runops loops, which is clearly incorrect. One way to avoid this is for the tie code to do a `JMPENV_PUSH` before executing `FETCH` in the inner runops loop, but for efficiency reasons, perl in fact just sets a flag, using `CATCH_SET(TRUE)`. The `pp_require`, `pp_entereval` and `pp_entertry` ops check this flag, and if true, they call `docatch`, which does a `JMPENV_PUSH` and starts a new runops level to execute the code, rather than doing it on the current loop.

As a further optimisation, on exit from the eval block in the `FETCH`, execution of the code following the block is still carried on in the inner loop. When an exception is raised, `docatch` compares the `JMPENV` level of the `CxEVAL` with `PL_top_env` and if they differ, just re-throws the exception. In this way any inner loops get popped.

Here's an example.

```
1: eval { tie @a, ’A’ };
2: sub A::TIEARRAY {
3:     eval { die };
4:     die;
5: }
```

To run this code, `perl_run` is called, which does a `JMPENV_PUSH` then enters a runops loop. This loop executes the eval and tie ops on line 1, with the eval pushing a `CxEVAL` onto the context stack.

The `pp_tie` does a `CATCH_SET(TRUE)`, then starts a second runops loop to execute the body of `TIEARRAY`. When it executes the entertry op on line 3, `CATCH_GET` is true, so `pp_entertry` calls `docatch` which does a `JMPENV_PUSH` and starts a third runops loop, which then executes the die op. At this point the C call stack looks like this:

```
Perl_pp_die
Perl_runops      # third loop
S_docatch_body
S_docatch
Perl_pp_entertry
Perl_runops      # second loop
S_call_body
Perl_call_sv
Perl_pp_tie
Perl_runops      # first loop
S_run_body
perl_run
main
```

and the context and data stacks, as shown by `-Dstv`, look like:

```
STACK 0: MAIN
  CX 0: BLOCK  =>
  CX 1: EVAL   => AV()  PV("A"\0)
  retop=leave
STACK 1: MAGIC
  CX 0: SUB    =>
  retop=(null)
```

```
      CX 1: EVAL   => *
    retop=nextstate
```

The die pops the first CxEVAL off the context stack, sets PL_restartop from it, does a JMPENV_JUMP(3), and control returns to the top docatch. This then starts another third-level runops level, which executes the nextstate, pushmark and die ops on line 4. At the point that the second pp_die is called, the C call stack looks exactly like that above, even though we are no longer within an inner eval; this is because of the optimization mentioned earlier. However, the context stack now looks like this, ie with the top CxEVAL popped:

```
STACK 0: MAIN
  CX 0: BLOCK  =>
  CX 1: EVAL   => AV()  PV("A"\0)
  retop=leave
STACK 1: MAGIC
  CX 0: SUB    =>
  retop=(null)
```

The die on line 4 pops the context stack back down to the CxEVAL, leaving it as:

```
STACK 0: MAIN
  CX 0: BLOCK  =>
```

As usual, PL_restartop is extracted from the CxEVAL, and a JMPENV_JUMP(3) done, which pops the C stack back to the docatch:

```
S_docatch
Perl_pp_entertry
Perl_runops      # second loop
S_call_body
Perl_call_sv
Perl_pp_tie
Perl_runops      # first loop
S_run_body
perl_run
main
```

In this case, because the JMPENV level recorded in the CxEVAL differs from the current one, docatch just does a JMPENV_JUMP(3) and the C stack unwinds to:

```
perl_run
main
```

Because PL_restartop is non-null, run_body starts a new runops loop and execution continues.

33.3.6 INTERNAL VARIABLE TYPES

You should by now have had a look at Section 28.1 [perlguts NAME], page 512, which tells you about Perl's internal variable types: SVs, HVs, AVs and the rest. If not, do that now.

These variables are used not only to represent Perl-space variables, but also any constants in the code, as well as some structures completely internal to Perl. The symbol table, for instance, is an ordinary Perl hash. Your code is represented by an SV as it's read into the parser; any program files you call are opened via ordinary Perl filehandles, and so on.

The core Devel-Peek module lets us examine SVs from a Perl program. Let's see, for instance, how Perl treats the constant "hello".

```
   % perl -MDevel::Peek -e 'Dump("hello")'
 1 SV = PV(0xa041450) at 0xa04ecbc
 2   REFCNT = 1
 3   FLAGS = (POK,READONLY,pPOK)
 4   PV = 0xa0484e0 "hello"\0
 5   CUR = 5
 6   LEN = 6
```

Reading Devel::Peek output takes a bit of practise, so let's go through it line by line.

Line 1 tells us we're looking at an SV which lives at 0xa04ecbc in memory. SVs themselves are very simple structures, but they contain a pointer to a more complex structure. In this case, it's a PV, a structure which holds a string value, at location 0xa041450. Line 2 is the reference count; there are no other references to this data, so it's 1.

Line 3 are the flags for this SV - it's OK to use it as a PV, it's a read-only SV (because it's a constant) and the data is a PV internally. Next we've got the contents of the string, starting at location 0xa0484e0.

Line 5 gives us the current length of the string - note that this does **not** include the null terminator. Line 6 is not the length of the string, but the length of the currently allocated buffer; as the string grows, Perl automatically extends the available storage via a routine called SvGROW.

You can get at any of these quantities from C very easily; just add Sv to the name of the field shown in the snippet, and you've got a macro which will return the value: SvCUR(sv) returns the current length of the string, SvREFCOUNT(sv) returns the reference count, SvPV(sv, len) returns the string itself with its length, and so on. More macros to manipulate these properties can be found in Section 28.1 [perlguts NAME], page 512.

Let's take an example of manipulating a PV, from sv_catpvn, in sv.c

```
  1  void
  2  Perl_sv_catpvn(pTHX_ SV *sv, const char *ptr, STRLEN len)
  3  {
  4      STRLEN tlen;
  5      char *junk;

  6      junk = SvPV_force(sv, tlen);
  7      SvGROW(sv, tlen + len + 1);
  8      if (ptr == junk)
  9          ptr = SvPVX(sv);
 10      Move(ptr,SvPVX(sv)+tlen,len,char);
 11      SvCUR(sv) += len;
 12      *SvEND(sv) = '\0';
 13      (void)SvPOK_only_UTF8(sv);          /* validate pointer */
 14      SvTAINT(sv);
 15  }
```

This is a function which adds a string, ptr, of length len onto the end of the PV stored in sv. The first thing we do in line 6 is make sure that the SV **has** a valid PV, by calling the `SvPV_force` macro to force a PV. As a side effect, `tlen` gets set to the current value of the PV, and the PV itself is returned to `junk`.

In line 7, we make sure that the SV will have enough room to accommodate the old string, the new string and the null terminator. If `LEN` isn't big enough, `SvGROW` will reallocate space for us.

Now, if `junk` is the same as the string we're trying to add, we can grab the string directly from the SV; `SvPVX` is the address of the PV in the SV.

Line 10 does the actual catenation: the `Move` macro moves a chunk of memory around: we move the string `ptr` to the end of the PV - that's the start of the PV plus its current length. We're moving `len` bytes of type `char`. After doing so, we need to tell Perl we've extended the string, by altering `CUR` to reflect the new length. `SvEND` is a macro which gives us the end of the string, so that needs to be a `"\0"`.

Line 13 manipulates the flags; since we've changed the PV, any IV or NV values will no longer be valid: if we have `$a=10; $a.="6";` we don't want to use the old IV of 10. `SvPOK_only_utf8` is a special UTF-8-aware version of `SvPOK_only`, a macro which turns off the IOK and NOK flags and turns on POK. The final `SvTAINT` is a macro which launders tainted data if taint mode is turned on.

AVs and HVs are more complicated, but SVs are by far the most common variable type being thrown around. Having seen something of how we manipulate these, let's go on and look at how the op tree is constructed.

33.4 OP TREES

First, what is the op tree, anyway? The op tree is the parsed representation of your program, as we saw in our section on parsing, and it's the sequence of operations that Perl goes through to execute your program, as we saw in Section 33.3.4 [Running], page 628.

An op is a fundamental operation that Perl can perform: all the built-in functions and operators are ops, and there are a series of ops which deal with concepts the interpreter needs internally - entering and leaving a block, ending a statement, fetching a variable, and so on.

The op tree is connected in two ways: you can imagine that there are two "routes" through it, two orders in which you can traverse the tree. First, parse order reflects how the parser understood the code, and secondly, execution order tells perl what order to perform the operations in.

The easiest way to examine the op tree is to stop Perl after it has finished parsing, and get it to dump out the tree. This is exactly what the compiler backends `B-Terse`, `B-Concise` and `B-Debug` do.

Let's have a look at how Perl sees `$a = $b + $c`:

```
% perl -MO=Terse -e '$a=$b+$c'
1  LISTOP (0x8179888) leave
2      OP (0x81798b0) enter
3      COP (0x8179850) nextstate
4      BINOP (0x8179828) sassign
5          BINOP (0x8179800) add [1]
6              UNOP (0x81796e0) null [15]
```

```
  7                 SVOP (0x80fafe0) gvsv  GV (0x80fa4cc) *b
  8             UNOP (0x81797e0) null [15]
  9                 SVOP (0x8179700) gvsv  GV (0x80efeb0) *c
 10         UNOP (0x816b4f0) null [15]
 11             SVOP (0x816dcf0) gvsv  GV (0x80fa460) *a
```

Let's start in the middle, at line 4. This is a BINOP, a binary operator, which is at location `0x8179828`. The specific operator in question is `sassign` - scalar assignment - and you can find the code which implements it in the function `pp_sassign` in `pp_hot.c`. As a binary operator, it has two children: the add operator, providing the result of `$b+$c`, is uppermost on line 5, and the left hand side is on line 10.

Line 10 is the null op: this does exactly nothing. What is that doing there? If you see the null op, it's a sign that something has been optimized away after parsing. As we mentioned in Section 33.3.3 [Optimization], page 628, the optimization stage sometimes converts two operations into one, for example when fetching a scalar variable. When this happens, instead of rewriting the op tree and cleaning up the dangling pointers, it's easier just to replace the redundant operation with the null op. Originally, the tree would have looked like this:

```
 10         SVOP (0x816b4f0) rv2sv [15]
 11             SVOP (0x816dcf0) gv  GV (0x80fa460) *a
```

That is, fetch the `a` entry from the main symbol table, and then look at the scalar component of it: `gvsv` (`pp_gvsv` into `pp_hot.c`) happens to do both these things.

The right hand side, starting at line 5 is similar to what we've just seen: we have the `add` op (`pp_add` also in `pp_hot.c`) add together two `gvsv`s.

Now, what's this about?

```
  1  LISTOP (0x8179888) leave
  2      OP (0x81798b0) enter
  3      COP (0x8179850) nextstate
```

`enter` and `leave` are scoping ops, and their job is to perform any housekeeping every time you enter and leave a block: lexical variables are tidied up, unreferenced variables are destroyed, and so on. Every program will have those first three lines: `leave` is a list, and its children are all the statements in the block. Statements are delimited by `nextstate`, so a block is a collection of `nextstate` ops, with the ops to be performed for each statement being the children of `nextstate`. `enter` is a single op which functions as a marker.

That's how Perl parsed the program, from top to bottom:

```
                        Program
                           |
                       Statement
                           |
                           =
                          / \
                         /   \
                        $a   +
                            / \
                          $b   $c
```

However, it's impossible to **perform** the operations in this order: you have to find the values of `$b` and `$c` before you add them together, for instance. So, the other thread that runs through the op tree is the execution order: each op has a field `op_next` which points to the next op to be run, so following these pointers tells us how perl executes the code. We can traverse the tree in this order using the `exec` option to `B::Terse`:

```
% perl -MO=Terse,exec -e '$a=$b+$c'
1  OP (0x8179928) enter
2  COP (0x81798c8) nextstate
3  SVOP (0x81796c8) gvsv  GV (0x80fa4d4) *b
4  SVOP (0x8179798) gvsv  GV (0x80efeb0) *c
5  BINOP (0x8179878) add [1]
6  SVOP (0x816dd38) gvsv  GV (0x80fa468) *a
7  BINOP (0x81798a0) sassign
8  LISTOP (0x8179900) leave
```

This probably makes more sense for a human: enter a block, start a statement. Get the values of `$b` and `$c`, and add them together. Find `$a`, and assign one to the other. Then leave.

The way Perl builds up these op trees in the parsing process can be unravelled by examining `perly.y`, the YACC grammar. Let's take the piece we need to construct the tree for `$a = $b + $c`

```
1 term    :   term ASSIGNOP term
2                { $$ = newASSIGNOP(OPf_STACKED, $1, $2, $3); }
3         |   term ADDOP term
4                { $$ = newBINOP($2, 0, scalar($1), scalar($3)); }
```

If you're not used to reading BNF grammars, this is how it works: You're fed certain things by the tokeniser, which generally end up in upper case. Here, `ADDOP`, is provided when the tokeniser sees `+` in your code. `ASSIGNOP` is provided when `=` is used for assigning. These are "terminal symbols", because you can't get any simpler than them.

The grammar, lines one and three of the snippet above, tells you how to build up more complex forms. These complex forms, "non-terminal symbols" are generally placed in lower case. `term` here is a non-terminal symbol, representing a single expression.

The grammar gives you the following rule: you can make the thing on the left of the colon if you see all the things on the right in sequence. This is called a "reduction", and the aim of parsing is to completely reduce the input. There are several different ways you can perform a reduction, separated by vertical bars: so, `term` followed by `=` followed by `term` makes a `term`, and `term` followed by `+` followed by `term` can also make a `term`.

So, if you see two terms with an `=` or `+`, between them, you can turn them into a single expression. When you do this, you execute the code in the block on the next line: if you see `=`, you'll do the code in line 2. If you see `+`, you'll do the code in line 4. It's this code which contributes to the op tree.

```
|   term ADDOP term
{ $$ = newBINOP($2, 0, scalar($1), scalar($3)); }
```

What this does is creates a new binary op, and feeds it a number of variables. The variables refer to the tokens: `$1` is the first token in the input, `$2` the second, and so on - think regular expression backreferences. `$$` is the op returned from this reduction. So, we call newBINOP to create a new binary operator. The first parameter to newBINOP, a function in op.c, is the op type. It's an addition operator, so we want the type to be ADDOP. We could specify this directly, but it's right there as the second token in the input, so we use $2. The second parameter is the op's flags: 0 means "nothing special". Then the things to add: the left and right hand side of our expression, in scalar context.

33.5 STACKS

When perl executes something like addop, how does it pass on its results to the next op? The answer is, through the use of stacks. Perl has a number of stacks to store things it's currently working on, and we'll look at the three most important ones here.

33.5.1 Argument stack

Arguments are passed to PP code and returned from PP code using the argument stack, ST. The typical way to handle arguments is to pop them off the stack, deal with them how you wish, and then push the result back onto the stack. This is how, for instance, the cosine operator works:

```
NV value;
value = POPn;
value = Perl_cos(value);
XPUSHn(value);
```

We'll see a more tricky example of this when we consider Perl's macros below. POPn gives you the NV (floating point value) of the top SV on the stack: the $x in cos($x). Then we compute the cosine, and push the result back as an NV. The X in XPUSHn means that the stack should be extended if necessary - it can't be necessary here, because we know there's room for one more item on the stack, since we've just removed one! The XPUSH* macros at least guarantee safety.

Alternatively, you can fiddle with the stack directly: SP gives you the first element in your portion of the stack, and TOP* gives you the top SV/IV/NV/etc. on the stack. So, for instance, to do unary negation of an integer:

```
SETi(-TOPi);
```

Just set the integer value of the top stack entry to its negation.

Argument stack manipulation in the core is exactly the same as it is in XSUBs - see perlxstut, perlxs and Section 28.1 [perlguts NAME], page 512 for a longer description of the macros used in stack manipulation.

33.5.2 Mark stack

I say "your portion of the stack" above because PP code doesn't necessarily get the whole stack to itself: if your function calls another function, you'll only want to expose the arguments aimed for the called function, and not (necessarily) let it get at your own data. The way we do this is to have a "virtual" bottom-of-stack, exposed to each function. The mark stack keeps bookmarks to locations in the argument stack usable by each function. For instance, when dealing with a tied variable, (internally, something with "P" magic) Perl has to call methods for accesses to the tied variables. However, we need to separate the arguments exposed to the method to the argument exposed to the original function - the store or fetch or whatever it may be. Here's roughly how the tied `push` is implemented; see `av_push` in `av.c`:

```
PUSHMARK(SP);
EXTEND(SP,2);
PUSHs(SvTIED_obj((SV*)av, mg));
PUSHs(val);
PUTBACK;
ENTER;
call_method("PUSH", G_SCALAR|G_DISCARD);
LEAVE;
```

Let's examine the whole implementation, for practice:

```
PUSHMARK(SP);
```

Push the current state of the stack pointer onto the mark stack. This is so that when we've finished adding items to the argument stack, Perl knows how many things we've added recently.

```
EXTEND(SP,2);
PUSHs(SvTIED_obj((SV*)av, mg));
PUSHs(val);
```

We're going to add two more items onto the argument stack: when you have a tied array, the `PUSH` subroutine receives the object and the value to be pushed, and that's exactly what we have here - the tied object, retrieved with `SvTIED_obj`, and the value, the SV `val`.

```
PUTBACK;
```

Next we tell Perl to update the global stack pointer from our internal variable: `dSP` only gave us a local copy, not a reference to the global.

```
ENTER;
call_method("PUSH", G_SCALAR|G_DISCARD);
LEAVE;
```

`ENTER` and `LEAVE` localise a block of code - they make sure that all variables are tidied up, everything that has been localised gets its previous value returned, and so on. Think of them as the `{` and `}` of a Perl block.

To actually do the magic method call, we have to call a subroutine in Perl space: `call_method` takes care of that, and it's described in Section 7.1 [perlcall NAME], page 28. We call the `PUSH` method in scalar context, and we're going to discard its return value. The call_method() function removes the top element of the mark stack, so there is nothing for the caller to clean up.

33.5.3 Save stack

C doesn't have a concept of local scope, so perl provides one. We've seen that `ENTER` and `LEAVE` are used as scoping braces; the save stack implements the C equivalent of, for example:

```
{
    local $foo = 42;
    ...
}
```

See Section 28.3.23 [perlguts Localizing changes], page 534 for how to use the save stack.

33.6 MILLIONS OF MACROS

One thing you'll notice about the Perl source is that it's full of macros. Some have called the pervasive use of macros the hardest thing to understand, others find it adds to clarity. Let's take an example, the code which implements the addition operator:

```
1  PP(pp_add)
2  {
3      dSP; dATARGET; tryAMAGICbin(add,opASSIGN);
4      {
5        dPOPTOPnnrl_ul;
6        SETn( left + right );
7        RETURN;
8      }
9  }
```

Every line here (apart from the braces, of course) contains a macro. The first line sets up the function declaration as Perl expects for PP code; line 3 sets up variable declarations for the argument stack and the target, the return value of the operation. Finally, it tries to see if the addition operation is overloaded; if so, the appropriate subroutine is called.

Line 5 is another variable declaration - all variable declarations start with `d` - which pops from the top of the argument stack two NVs (hence `nn`) and puts them into the variables `right` and `left`, hence the `rl`. These are the two operands to the addition operator. Next, we call `SETn` to set the NV of the return value to the result of adding the two values. This done, we return - the `RETURN` macro makes sure that our return value is properly handled, and we pass the next operator to run back to the main run loop.

Most of these macros are explained in `perlapi`, and some of the more important ones are explained in `perlxs` as well. Pay special attention to Section 28.9.1 [perlguts Background and PERL_IMPLICIT_CONTEXT], page 547 for information on the `[pad]THX_?` macros.

33.7 FURTHER READING

For more information on the Perl internals, please see the documents listed at Section 1.3.4 [perl Internals and C Language Interface], page 3.

34 perlintro

34.1 NAME

perlintro – a brief introduction and overview of Perl

34.2 DESCRIPTION

This document is intended to give you a quick overview of the Perl programming language, along with pointers to further documentation. It is intended as a "bootstrap" guide for those who are new to the language, and provides just enough information for you to be able to read other peoples' Perl and understand roughly what it's doing, or write your own simple scripts.

This introductory document does not aim to be complete. It does not even aim to be entirely accurate. In some cases perfection has been sacrificed in the goal of getting the general idea across. You are *strongly* advised to follow this introduction with more information from the full Perl manual, the table of contents to which can be found in `perltoc`.

Throughout this document you'll see references to other parts of the Perl documentation. You can read that documentation using the `perldoc` command or whatever method you're using to read this document.

Throughout Perl's documentation, you'll find numerous examples intended to help explain the discussed features. Please keep in mind that many of them are code fragments rather than complete programs.

These examples often reflect the style and preference of the author of that piece of the documentation, and may be briefer than a corresponding line of code in a real program. Except where otherwise noted, you should assume that `use strict` and `use warnings` statements appear earlier in the "program", and that any variables used have already been declared, even if those declarations have been omitted to make the example easier to read.

Do note that the examples have been written by many different authors over a period of several decades. Styles and techniques will therefore differ, although some effort has been made to not vary styles too widely in the same sections. Do not consider one style to be better than others - "There's More Than One Way To Do It" is one of Perl's mottos. After all, in your journey as a programmer, you are likely to encounter different styles.

34.2.1 What is Perl?

Perl is a general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, GUI development, and more.

The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). Its major features are that it's easy to use, supports both procedural and object-oriented (OO) programming, has powerful built-in support for text processing, and has one of the world's most impressive collections of third-party modules.

Different definitions of Perl are given in Section 1.1 [perl NAME], page 1, `perlfaq1` and no doubt other places. From this we can determine that Perl is different things to different people, but that lots of people think it's at least worth writing about.

34.2.2 Running Perl programs

To run a Perl program from the Unix command line:

```
perl progname.pl
```

Alternatively, put this as the first line of your script:

```
#!/usr/bin/env perl
```

... and run the script as `/path/to/script.pl`. Of course, it'll need to be executable first, so `chmod 755 script.pl` (under Unix).

(This start line assumes you have the **env** program. You can also put directly the path to your perl executable, like in `#!/usr/bin/perl`).

For more information, including instructions for other platforms such as Windows and Mac OS, read Section 69.1 [perlrun NAME], page 1176.

34.2.3 Safety net

Perl by default is very forgiving. In order to make it more robust it is recommended to start every program with the following lines:

```
#!/usr/bin/perl
use strict;
use warnings;
```

The two additional lines request from perl to catch various common problems in your code. They check different things so you need both. A potential problem caught by `use strict;` will cause your code to stop immediately when it is encountered, while `use warnings;` will merely give a warning (like the command-line switch **-w**) and let your code run. To read more about them check their respective manual pages at `strict` and `warnings`.

34.2.4 Basic syntax overview

A Perl script or program consists of one or more statements. These statements are simply written in the script in a straightforward fashion. There is no need to have a `main()` function or anything of that kind.

Perl statements end in a semi-colon:

```
print "Hello, world";
```

Comments start with a hash symbol and run to the end of the line

```
# This is a comment
```

Whitespace is irrelevant:

```
print
    "Hello, world"
    ;
```

... except inside quoted strings:

```
# this would print with a linebreak in the middle
print "Hello
world";
```

Double quotes or single quotes may be used around literal strings:

```
print "Hello, world";
print 'Hello, world';
```

However, only double quotes "interpolate" variables and special characters such as newlines (\n):

```
print "Hello, $name\n";     # works fine
print 'Hello, $name\n';     # prints $name\n literally
```

Numbers don't need quotes around them:

```
print 42;
```

You can use parentheses for functions' arguments or omit them according to your personal taste. They are only required occasionally to clarify issues of precedence.

```
print("Hello, world\n");
print "Hello, world\n";
```

More detailed information about Perl syntax can be found in Section 74.1 [perlsyn NAME], page 1249.

34.2.5 Perl variable types

Perl has three main variable types: scalars, arrays, and hashes.

Scalars

A scalar represents a single value:

```
my $animal = "camel";
my $answer = 42;
```

Scalar values can be strings, integers or floating point numbers, and Perl will automatically convert between them as required. There is no need to predeclare your variable types, but you have to declare them using the `my` keyword the first time you use them. (This is one of the requirements of `use strict;`.)

Scalar values can be used in various ways:

```
print $animal;
print "The animal is $animal\n";
print "The square of $answer is ", $answer * $answer, "\n";
```

There are a number of "magic" scalars with names that look like punctuation or line noise. These special variables are used for all kinds of purposes, and are documented in Section 86.1 [perlvar NAME], page 1375. The only one you need to know about for now is `$_` which is the "default variable". It's used as the default argument to a number of functions in Perl, and it's set implicitly by certain looping constructs.

```
print;          # prints contents of $_ by default
```

Arrays

An array represents a list of values:

```
my @animals = ("camel", "llama", "owl");
my @numbers = (23, 42, 69);
my @mixed   = ("camel", 42, 1.23);
```

Arrays are zero-indexed. Here's how you get at elements in an array:

```
print $animals[0];              # prints "camel"
print $animals[1];              # prints "llama"
```

The special variable `$#array` tells you the index of the last element of an array:

```
print $mixed[$#mixed];       # last element, prints 1.23
```

You might be tempted to use `$#array + 1` to tell you how many items there are in an array. Don't bother. As it happens, using `@array` where Perl expects to find a scalar value ("in scalar context") will give you the number of elements in the array:

```
if (@animals < 5) { ... }
```

The elements we're getting from the array start with a `$` because we're getting just a single value out of the array; you ask for a scalar, you get a scalar.

To get multiple values from an array:

```
@animals[0,1];                 # gives ("camel", "llama");
@animals[0..2];                # gives ("camel", "llama", "owl");
@animals[1..$#animals];        # gives all except the first element
```

This is called an "array slice".

You can do various useful things to lists:

```
my @sorted    = sort @animals;
my @backwards = reverse @numbers;
```

There are a couple of special arrays too, such as `@ARGV` (the command line arguments to your script) and `@_` (the arguments passed to a subroutine). These are documented in Section 86.1 [perlvar NAME], page 1375.

Hashes

A hash represents a set of key/value pairs:

```
my %fruit_color = ("apple", "red", "banana", "yellow");
```

You can use whitespace and the `=>` operator to lay them out more nicely:

```
my %fruit_color = (
    apple  => "red",
    banana => "yellow",
);
```

To get at hash elements:

```
$fruit_color{"apple"};           # gives "red"
```

You can get at lists of keys and values with `keys()` and `values()`.

```
my @fruits = keys %fruit_colors;
my @colors = values %fruit_colors;
```

Hashes have no particular internal order, though you can sort the keys and loop through them.

Just like special scalars and arrays, there are also special hashes. The most well known of these is `%ENV` which contains environment variables. Read all about it (and other special variables) in Section 86.1 [perlvar NAME], page 1375.

Scalars, arrays and hashes are documented more fully in Section 11.1 [perldata NAME], page 70.

More complex data types can be constructed using references, which allow you to build lists and hashes within lists and hashes.

A reference is a scalar value and can refer to any other Perl data type. So by storing a reference as the value of an array or hash element, you can easily create lists and hashes within lists and hashes. The following example shows a 2 level hash of hash structure using anonymous hash references.

```
my $variables = {
    scalar  =>  {
                 description => "single item",
                 sigil => '$',
                },
    array   =>  {
                 description => "ordered list of items",
                 sigil => '@',
                },
    hash    =>  {
                 description => "key/value pairs",
                 sigil => '%',
                },
};

print "Scalars begin with a $variables->{'scalar'}->{'sigil'}\n";
```

Exhaustive information on the topic of references can be found in Section 63.1 [perlreftut NAME], page 1092, Section 39.1 [perllol NAME], page 725, Section 62.1 [perlref NAME], page 1077 and Section 17.1 [perldsc NAME], page 246.

34.2.6 Variable scoping

Throughout the previous section all the examples have used the syntax:

```
my $var = "value";
```

The `my` is actually not required; you could just use:

```
$var = "value";
```

However, the above usage will create global variables throughout your program, which is bad programming practice. `my` creates lexically scoped variables instead. The variables are scoped to the block (i.e. a bunch of statements surrounded by curly-braces) in which they are defined.

```
my $x = "foo";
my $some_condition = 1;
if ($some_condition) {
    my $y = "bar";
    print $x;           # prints "foo"
    print $y;           # prints "bar"
}
print $x;               # prints "foo"
print $y;               # prints nothing; $y has fallen out of scope
```

Using `my` in combination with a `use strict;` at the top of your Perl scripts means that the interpreter will pick up certain common programming errors. For instance, in the example above, the final `print $y` would cause a compile-time error and prevent you from running the program. Using `strict` is highly recommended.

34.2.7 Conditional and looping constructs

Perl has most of the usual conditional and looping constructs. As of Perl 5.10, it even has a case/switch statement (spelled `given/when`). See Section 74.2.11 [perlsyn Switch Statements], page 1258 for more details.

The conditions can be any Perl expression. See the list of operators in the next section for information on comparison and boolean logic operators, which are commonly used in conditional statements.

if

```
if ( condition ) {
    ...
} elsif ( other condition ) {
    ...
} else {
    ...
}
```

There's also a negated version of it:

```
unless ( condition ) {
    ...
}
```

This is provided as a more readable version of `if (!`*`condition`*`)`.

Note that the braces are required in Perl, even if you've only got one line in the block. However, there is a clever way of making your one-line conditional blocks more English like:

```
# the traditional way
if ($zippy) {
    print "Yow!";
}

# the Perlish post-condition way
print "Yow!" if $zippy;
print "We have no bananas" unless $bananas;
```

while

```
while ( condition ) {
    ...
}
```

There's also a negated version, for the same reason we have `unless`:

```
until ( condition ) {
    ...
}
```

You can also use `while` in a post-condition:

```
print "LA LA LA\n" while 1;          # loops forever
```

for

Exactly like C:

```
for ($i = 0; $i <= $max; $i++) {
    ...
}
```

The C style for loop is rarely needed in Perl since Perl provides the more friendly list scanning `foreach` loop.

foreach

```
foreach (@array) {
    print "This element is $_\n";
}

print $list[$_] foreach 0 .. $max;

# you don't have to use the default $_ either...
foreach my $key (keys %hash) {
    print "The value of $key is $hash{$key}\n";
}
```

The `foreach` keyword is actually a synonym for the `for` keyword. See `Section 74.2.9 [perlsyn Foreach Loops], page 1256`.

For more detail on looping constructs (and some that weren't mentioned in this overview) see Section 74.1 [perlsyn NAME], page 1249.

34.2.8 Builtin operators and functions

Perl comes with a wide selection of builtin functions. Some of the ones we've already seen include `print`, `sort` and `reverse`. A list of them is given at the start of Section 25.1 [perlfunc NAME], page 351 and you can easily read about any given function by using `perldoc -f` *`functionname`*.

Perl operators are documented in full in Section 48.1 [perlop NAME], page 798, but here are a few of the most common ones:

Arithmetic

```
+   addition
-   subtraction
*   multiplication
/   division
```

Numeric comparison

```
==  equality
!=  inequality
<   less than
>   greater than
<=  less than or equal
>=  greater than or equal
```

String comparison

`eq`	`equality`
`ne`	`inequality`
`lt`	`less than`
`gt`	`greater than`
`le`	`less than or equal`
`ge`	`greater than or equal`

(Why do we have separate numeric and string comparisons? Because we don't have special variable types, and Perl needs to know whether to sort numerically (where 99 is less than 100) or alphabetically (where 100 comes before 99).

Boolean logic

`&&`	`and`
`||`	`or`
`!`	`not`

(`and`, `or` and `not` aren't just in the above table as descriptions of the operators. They're also supported as operators in their own right. They're more readable than the C-style operators, but have different precedence to `&&` and friends. Check Section 48.1 [perlop NAME], page 798 for more detail.)

Miscellaneous

`=`	`assignment`
`.`	`string concatenation`
`x`	`string multiplication`
`..`	`range operator (creates a list of numbers or strings)`

Many operators can be combined with a `=` as follows:

```
$a += 1;        # same as $a = $a + 1
$a -= 1;        # same as $a = $a - 1
$a .= "\n";     # same as $a = $a . "\n";
```

34.2.9 Files and I/O

You can open a file for input or output using the `open()` function. It's documented in extravagant detail in Section 25.1 [perlfunc NAME], page 351 and Section 49.1 [perlopentut NAME], page 852, but in short:

```
open(my $in,  "<",  "input.txt")  or die "Can't open input.txt: $!";
open(my $out, ">",  "output.txt") or die "Can't open output.txt: $!";
open(my $log, ">>", "my.log")     or die "Can't open my.log: $!";
```

You can read from an open filehandle using the `<>` operator. In scalar context it reads a single line from the filehandle, and in list context it reads the whole file in, assigning each line to an element of the list:

```
my $line  = <$in>;
my @lines = <$in>;
```

Reading in the whole file at one time is called slurping. It can be useful but it may be a memory hog. Most text file processing can be done a line at a time with Perl's looping constructs.

The `<>` operator is most often seen in a `while` loop:

```
while (<$in>) {     # assigns each line in turn to $_
    print "Just read in this line: $_";
}
```

We've already seen how to print to standard output using `print()`. However, `print()` can also take an optional first argument specifying which filehandle to print to:

```
print STDERR "This is your final warning.\n";
print $out $record;
print $log $logmessage;
```

When you're done with your filehandles, you should `close()` them (though to be honest, Perl will clean up after you if you forget):

```
close $in or die "$in: $!";
```

34.2.10 Regular expressions

Perl's regular expression support is both broad and deep, and is the subject of lengthy documentation in Section 66.1 [perlrequick NAME], page 1116, Section 68.1 [perlretut NAME], page 1131, and elsewhere. However, in short:

Simple matching

```
if (/foo/)       { ... }  # true if $_ contains "foo"
if ($a =~ /foo/) { ... }  # true if $a contains "foo"
```

The `//` matching operator is documented in Section 48.1 [perlop NAME], page 798. It operates on `$_` by default, or can be bound to another variable using the `=~` binding operator (also documented in Section 48.1 [perlop NAME], page 798).

Simple substitution

```
s/foo/bar/;               # replaces foo with bar in $_
$a =~ s/foo/bar/;         # replaces foo with bar in $a
$a =~ s/foo/bar/g;        # replaces ALL INSTANCES of foo with bar
                          # in $a
```

The `s///` substitution operator is documented in Section 48.1 [perlop NAME], page 798.

More complex regular expressions

You don't just have to match on fixed strings. In fact, you can match on just about anything you could dream of by using more complex regular expressions. These are documented at great length in Section 58.1 [perlre NAME], page 989, but for the meantime, here's a quick cheat sheet:

```
.                   a single character
\s                  a whitespace character (space, tab, newline,
                    ...)
\S                  non-whitespace character
\d                  a digit (0-9)
\D                  a non-digit
\w                  a word character (a-z, A-Z, 0-9, _)
\W                  a non-word character
[aeiou]             matches a single character in the given set
```

```
[^aeiou]            matches a single character outside the given
                    set
(foo|bar|baz)       matches any of the alternatives specified

^                   start of string
$                   end of string
```

Quantifiers can be used to specify how many of the previous thing you want to match on, where "thing" means either a literal character, one of the metacharacters listed above, or a group of characters or metacharacters in parentheses.

```
*                   zero or more of the previous thing
+                   one or more of the previous thing
?                   zero or one of the previous thing
{3}                 matches exactly 3 of the previous thing
{3,6}               matches between 3 and 6 of the previous thing
{3,}                matches 3 or more of the previous thing
```

Some brief examples:

```
/^\d+/              string starts with one or more digits
/^$/                nothing in the string (start and end are
                    adjacent)
/(\d\s){3}/         three digits, each followed by a whitespace
                    character (eg "3 4 5 ")
/(a.)+/             matches a string in which every odd-numbered
                    letter is a (eg "abacadaf")

# This loop reads from STDIN, and prints non-blank lines:
while (<>) {
    next if /^$/;
    print;
}
```

Parentheses for capturing

As well as grouping, parentheses serve a second purpose. They can be used to capture the results of parts of the regexp match for later use. The results end up in $1, $2 and so on.

```
# a cheap and nasty way to break an email address up into parts

if ($email =~ /([^@]+)@(.+)/) {
    print "Username is $1\n";
    print "Hostname is $2\n";
}
```

Other regexp features

Perl regexps also support backreferences, lookaheads, and all kinds of other complex details. Read all about them in Section 66.1 [perlrequick NAME], page 1116, Section 68.1 [perlretut NAME], page 1131, and Section 58.1 [perlre NAME], page 989.

34.2.11 Writing subroutines

Writing subroutines is easy:

```
sub logger {
   my $logmessage = shift;
   open my $logfile, ">>", "my.log" or die "Could not open my.log: $!";
   print $logfile $logmessage;
}
```

Now we can use the subroutine just as any other built-in function:

```
logger("We have a logger subroutine!");
```

What's that shift? Well, the arguments to a subroutine are available to us as a special array called @_ (see Section 86.1 [perlvar NAME], page 1375 for more on that). The default argument to the shift function just happens to be @_. So my $logmessage = shift; shifts the first item off the list of arguments and assigns it to $logmessage.

We can manipulate @_ in other ways too:

```
my ($logmessage, $priority) = @_;       # common
my $logmessage = $_[0];                 # uncommon, and ugly
```

Subroutines can also return values:

```
sub square {
    my $num = shift;
    my $result = $num * $num;
    return $result;
}
```

Then use it like:

```
$sq = square(8);
```

For more information on writing subroutines, see Section 73.1 [perlsub NAME], page 1216.

34.2.12 OO Perl

OO Perl is relatively simple and is implemented using references which know what sort of object they are based on Perl's concept of packages. However, OO Perl is largely beyond the scope of this document. Read Section 47.1 [perlootut NAME], page 786 and Section 46.1 [perlobj NAME], page 769.

As a beginning Perl programmer, your most common use of OO Perl will be in using third-party modules, which are documented below.

34.2.13 Using Perl modules

Perl modules provide a range of features to help you avoid reinventing the wheel, and can be downloaded from CPAN (http://www.cpan.org/). A number of popular modules are included with the Perl distribution itself.

Categories of modules range from text manipulation to network protocols to database integration to graphics. A categorized list of modules is also available from CPAN.

To learn how to install modules you download from CPAN, read Section 41.1 [perlmodinstall NAME], page 742.

To learn how to use a particular module, use `perldoc` *`Module::Name`*. Typically you will want to `use` *`Module::Name`*, which will then give you access to exported functions or an OO interface to the module.

`perlfaq` contains questions and answers related to many common tasks, and often provides suggestions for good CPAN modules to use.

Section 40.1 [perlmod NAME], page 732 describes Perl modules in general. `perlmodlib` lists the modules which came with your Perl installation.

If you feel the urge to write Perl modules, Section 44.1 [perlnewmod NAME], page 760 will give you good advice.

34.3 AUTHOR

Kirrily "Skud" Robert <skud@cpan.org>

35 perliol

35.1 NAME

perliol - C API for Perl's implementation of IO in Layers.

35.2 SYNOPSIS

```
    /* Defining a layer ... */
    #include <perliol.h>
```

35.3 DESCRIPTION

This document describes the behavior and implementation of the PerlIO abstraction described in Section 2.1 [perlapio NAME], page 9 when `USE_PERLIO` is defined.

35.3.1 History and Background

The PerlIO abstraction was introduced in perl5.003_02 but languished as just an abstraction until perl5.7.0. However during that time a number of perl extensions switched to using it, so the API is mostly fixed to maintain (source) compatibility.

The aim of the implementation is to provide the PerlIO API in a flexible and platform neutral manner. It is also a trial of an "Object Oriented C, with vtables" approach which may be applied to Perl 6.

35.3.2 Basic Structure

PerlIO is a stack of layers.

The low levels of the stack work with the low-level operating system calls (file descriptors in C) getting bytes in and out, the higher layers of the stack buffer, filter, and otherwise manipulate the I/O, and return characters (or bytes) to Perl. Terms *above* and *below* are used to refer to the relative positioning of the stack layers.

A layer contains a "vtable", the table of I/O operations (at C level a table of function pointers), and status flags. The functions in the vtable implement operations like "open", "read", and "write".

When I/O, for example "read", is requested, the request goes from Perl first down the stack using "read" functions of each layer, then at the bottom the input is requested from the operating system services, then the result is returned up the stack, finally being interpreted as Perl data.

The requests do not necessarily go always all the way down to the operating system: that's where PerlIO buffering comes into play.

When you do an open() and specify extra PerlIO layers to be deployed, the layers you specify are "pushed" on top of the already existing default stack. One way to see it is that "operating system is on the left" and "Perl is on the right".

What exact layers are in this default stack depends on a lot of things: your operating system, Perl version, Perl compile time configuration, and Perl runtime configuration. See `PerlIO`, [perlrun PERLIO], page 1191, and `open` for more information.

binmode() operates similarly to open(): by default the specified layers are pushed on top of the existing stack.

However, note that even as the specified layers are "pushed on top" for open() and binmode(), this doesn't mean that the effects are limited to the "top": PerlIO layers can be very 'active' and inspect and affect layers also deeper in the stack. As an example there is a layer called "raw" which repeatedly "pops" layers until it reaches the first layer that has declared itself capable of handling binary data. The "pushed" layers are processed in left-to-right order.

sysopen() operates (unsurprisingly) at a lower level in the stack than open(). For example in Unix or Unix-like systems sysopen() operates directly at the level of file descriptors: in the terms of PerlIO layers, it uses only the "unix" layer, which is a rather thin wrapper on top of the Unix file descriptors.

35.3.3 Layers vs Disciplines

Initial discussion of the ability to modify IO streams behaviour used the term "discipline" for the entities which were added. This came (I believe) from the use of the term in "sfio", which in turn borrowed it from "line disciplines" on Unix terminals. However, this document (and the C code) uses the term "layer".

This is, I hope, a natural term given the implementation, and should avoid connotations that are inherent in earlier uses of "discipline" for things which are rather different.

35.3.4 Data Structures

The basic data structure is a PerlIOl:

```
typedef struct _PerlIO PerlIOl;
typedef struct _PerlIO_funcs PerlIO_funcs;
typedef PerlIOl *PerlIO;

struct _PerlIO
{
 PerlIOl *      next;       /* Lower layer */
 PerlIO_funcs * tab;        /* Functions for this layer */
 U32            flags;      /* Various flags for state */
};
```

A `PerlIOl *` is a pointer to the struct, and the *application* level `PerlIO *` is a pointer to a `PerlIOl *` - i.e. a pointer to a pointer to the struct. This allows the application level `PerlIO *` to remain constant while the actual `PerlIOl *` underneath changes. (Compare perl's `SV *` which remains constant while its `sv_any` field changes as the scalar's type changes.) An IO stream is then in general represented as a pointer to this linked-list of "layers".

It should be noted that because of the double indirection in a `PerlIO *`, a `&(perlio->next)` "is" a `PerlIO *`, and so to some degree at least one layer can use the "standard" API on the next layer down.

A "layer" is composed of two parts:

1. The functions and attributes of the "layer class".
2. The per-instance data for a particular handle.

35.3.5 Functions and Attributes

The functions and attributes are accessed via the "tab" (for table) member of PerlIOl. The functions (methods of the layer "class") are fixed, and are defined by the PerlIO_funcs type. They are broadly the same as the public PerlIO_xxxxx functions:

```
struct _PerlIO_funcs
{
 Size_t              fsize;
 char *              name;
 Size_t              size;
 IV          kind;
 IV          (*Pushed)(pTHX_ PerlIO *f,const char *mode,SV *arg, PerlIO_funcs *tab);
 IV          (*Popped)(pTHX_ PerlIO *f);
 PerlIO *    (*Open)(pTHX_ PerlIO_funcs *tab,
                     PerlIO_list_t *layers, IV n,
                     const char *mode,
                     int fd, int imode, int perm,
                     PerlIO *old,
                     int narg, SV **args);
 IV          (*Binmode)(pTHX_ PerlIO *f);
 SV *        (*Getarg)(pTHX_ PerlIO *f, CLONE_PARAMS *param, int flags)
 IV          (*Fileno)(pTHX_ PerlIO *f);
 PerlIO *    (*Dup)(pTHX_ PerlIO *f, PerlIO *o, CLONE_PARAMS *param, int flags)
 /* Unix-like functions - cf sfio line disciplines */
 SSize_t     (*Read)(pTHX_ PerlIO *f, void *vbuf, Size_t count);
 SSize_t     (*Unread)(pTHX_ PerlIO *f, const void *vbuf, Size_t count);
 SSize_t     (*Write)(pTHX_ PerlIO *f, const void *vbuf, Size_t count);
 IV          (*Seek)(pTHX_ PerlIO *f, Off_t offset, int whence);
 Off_t       (*Tell)(pTHX_ PerlIO *f);
 IV          (*Close)(pTHX_ PerlIO *f);
 /* Stdio-like buffered IO functions */
 IV          (*Flush)(pTHX_ PerlIO *f);
 IV          (*Fill)(pTHX_ PerlIO *f);
 IV          (*Eof)(pTHX_ PerlIO *f);
 IV          (*Error)(pTHX_ PerlIO *f);
 void        (*Clearerr)(pTHX_ PerlIO *f);
 void        (*Setlinebuf)(pTHX_ PerlIO *f);
 /* Perl's snooping functions */
 STDCHAR *   (*Get_base)(pTHX_ PerlIO *f);
 Size_t      (*Get_bufsiz)(pTHX_ PerlIO *f);
 STDCHAR *   (*Get_ptr)(pTHX_ PerlIO *f);
 SSize_t     (*Get_cnt)(pTHX_ PerlIO *f);
 void        (*Set_ptrcnt)(pTHX_ PerlIO *f,STDCHAR *ptr,SSize_t cnt);
};
```

The first few members of the struct give a function table size for compatibility check "name" for the layer, the size to malloc for the per-instance data, and some flags which are attributes of the class as whole (such as whether it is a buffering layer), then follow the functions which fall into four basic groups:

1. Opening and setup functions
2. Basic IO operations
3. Stdio class buffering options.
4. Functions to support Perl's traditional "fast" access to the buffer.

A layer does not have to implement all the functions, but the whole table has to be present. Unimplemented slots can be NULL (which will result in an error when called) or can be filled in with stubs to "inherit" behaviour from a "base class". This "inheritance" is fixed for all instances of the layer, but as the layer chooses which stubs to populate the table, limited "multiple inheritance" is possible.

35.3.6 Per-instance Data

The per-instance data are held in memory beyond the basic PerlIOl struct, by making a PerlIOl the first member of the layer's struct thus:

```
typedef struct
{
 struct _PerlIO base;       /* Base "class" info */
 STDCHAR *      buf;        /* Start of buffer */
 STDCHAR *      end;        /* End of valid part of buffer */
 STDCHAR *      ptr;        /* Current position in buffer */
 Off_t          posn;       /* Offset of buf into the file */
 Size_t         bufsiz;     /* Real size of buffer */
 IV             oneword;    /* Emergency buffer */
} PerlIOBuf;
```

In this way (as for perl's scalars) a pointer to a PerlIOBuf can be treated as a pointer to a PerlIOl.

35.3.7 Layers in action.

```
                table           perlio          unix
            |           |
            +-----------+    +----------+    +--------+
   PerlIO ->|           |--->|  next    |--->|  NULL  |
            +-----------+    +----------+    +--------+
            |           |    |  buffer  |    |   fd   |
            +-----------+    |          |    +--------+
            |           |    +----------+
```

The above attempts to show how the layer scheme works in a simple case. The application's `PerlIO *` points to an entry in the table(s) representing open (allocated) handles. For example the first three slots in the table correspond to `stdin`,`stdout` and `stderr`. The table in turn points to the current "top" layer for the handle - in this case an instance of the generic buffering layer "perlio". That layer in turn points to the next layer down - in this case the low-level "unix" layer.

The above is roughly equivalent to a "stdio" buffered stream, but with much more flexibility:

- If Unix level `read`/`write`/`lseek` is not appropriate for (say) sockets then the "unix" layer can be replaced (at open time or even dynamically) with a "socket" layer.
- Different handles can have different buffering schemes. The "top" layer could be the "mmap" layer if reading disk files was quicker using `mmap` than `read`. An "unbuffered" stream can be implemented simply by not having a buffer layer.
- Extra layers can be inserted to process the data as it flows through. This was the driving need for including the scheme in perl 5.7.0+ - we needed a mechanism to allow data to be translated between perl's internal encoding (conceptually at least Unicode as UTF-8), and the "native" format used by the system. This is provided by the ":encoding(xxxx)" layer which typically sits above the buffering layer.
- A layer can be added that does "\n" to CRLF translation. This layer can be used on any platform, not just those that normally do such things.

35.3.8 Per-instance flag bits

The generic flag bits are a hybrid of `O_XXXXX` style flags deduced from the mode string passed to `PerlIO_open()`, and state bits for typical buffer layers.

PERLIO_F_EOF
: End of file.

PERLIO_F_CANWRITE
: Writes are permitted, i.e. opened as "w" or "r+" or "a", etc.

PERLIO_F_CANREAD
: Reads are permitted i.e. opened "r" or "w+" (or even "a+" - ick).

PERLIO_F_ERROR
: An error has occurred (for `PerlIO_error()`).

PERLIO_F_TRUNCATE
: Truncate file suggested by open mode.

PERLIO_F_APPEND
: All writes should be appends.

PERLIO_F_CRLF
: Layer is performing Win32-like "\n" mapped to CR,LF for output and CR,LF mapped to "\n" for input. Normally the provided "crlf" layer is the only layer that need bother about this. `PerlIO_binmode()` will mess with this flag rather than add/remove layers if the `PERLIO_K_CANCRLF` bit is set for the layers class.

PERLIO_F_UTF8
: Data written to this layer should be UTF-8 encoded; data provided by this layer should be considered UTF-8 encoded. Can be set on any layer by ":utf8" dummy layer. Also set on ":encoding" layer.

PERLIO_F_UNBUF
: Layer is unbuffered - i.e. write to next layer down should occur for each write to this layer.

PERLIO_F_WRBUF
: The buffer for this layer currently holds data written to it but not sent to next layer.

PERLIO_F_RDBUF

The buffer for this layer currently holds unconsumed data read from layer below.

PERLIO_F_LINEBUF

Layer is line buffered. Write data should be passed to next layer down whenever a "\n" is seen. Any data beyond the "\n" should then be processed.

PERLIO_F_TEMP

File has been `unlink()`ed, or should be deleted on `close()`.

PERLIO_F_OPEN

Handle is open.

PERLIO_F_FASTGETS

This instance of this layer supports the "fast `gets`" interface. Normally set based on `PERLIO_K_FASTGETS` for the class and by the existence of the function(s) in the table. However a class that normally provides that interface may need to avoid it on a particular instance. The "pending" layer needs to do this when it is pushed above a layer which does not support the interface. (Perl's `sv_gets()` does not expect the streams fast `gets` behaviour to change during one "get".)

35.3.9 Methods in Detail

fsize

```
        Size_t fsize;
```

Size of the function table. This is compared against the value PerlIO code "knows" as a compatibility check. Future versions *may* be able to tolerate layers compiled against an old version of the headers.

name

```
        char * name;
```

The name of the layer whose open() method Perl should invoke on open(). For example if the layer is called APR, you will call:

```
  open $fh, ">:APR", ...
```

and Perl knows that it has to invoke the PerlIOAPR_open() method implemented by the APR layer.

size

```
        Size_t size;
```

The size of the per-instance data structure, e.g.:

```
  sizeof(PerlIOAPR)
```

If this field is zero then `PerlIO_pushed` does not malloc anything and assumes layer's Pushed function will do any required layer stack manipulation - used to avoid malloc/free overhead for dummy layers. If the field is non-zero it must be at least the size of `PerlIOl`, `PerlIO_pushed` will allocate memory for the layer's data structures and link new layer onto the stream's stack. (If the layer's Pushed method returns an error indication the layer is popped again.)

kind

```
IV kind;
```

- PERLIO_K_BUFFERED

  The layer is buffered.

- PERLIO_K_RAW

  The layer is acceptable to have in a binmode(FH) stack - i.e. it does not (or will configure itself not to) transform bytes passing through it.

- PERLIO_K_CANCRLF

  Layer can translate between `"\n"` and CRLF line ends.

- PERLIO_K_FASTGETS

  Layer allows buffer snooping.

- PERLIO_K_MULTIARG

  Used when the layer's open() accepts more arguments than usual. The extra arguments should come not before the `MODE` argument. When this flag is used it's up to the layer to validate the args.

Pushed

```
IV      (*Pushed)(pTHX_ PerlIO *f,const char *mode, SV *arg);
```

The only absolutely mandatory method. Called when the layer is pushed onto the stack. The `mode` argument may be NULL if this occurs post-open. The `arg` will be non-`NULL` if an argument string was passed. In most cases this should call `PerlIOBase_pushed()` to convert `mode` into the appropriate `PERLIO_F_XXXXX` flags in addition to any actions the layer itself takes. If a layer is not expecting an argument it need neither save the one passed to it, nor provide `Getarg()` (it could perhaps `Perl_warn` that the argument was un-expected).

Returns 0 on success. On failure returns -1 and should set errno.

Popped

```
IV      (*Popped)(pTHX_ PerlIO *f);
```

Called when the layer is popped from the stack. A layer will normally be popped after `Close()` is called. But a layer can be popped without being closed if the program is dynamically managing layers on the stream. In such cases `Popped()` should free any resources (buffers, translation tables, ...) not held directly in the layer's struct. It should also `Unread()` any unconsumed data that has been read and buffered from the layer below back to that layer, so that it can be re-provided to what ever is now above.

Returns 0 on success and failure. If `Popped()` returns *true* then *perlio.c* assumes that either the layer has popped itself, or the layer is super special and needs to be retained for other reasons. In most cases it should return *false*.

Open

```
PerlIO *        (*Open)(...);
```

The `Open()` method has lots of arguments because it combines the functions of perl's `open`, `PerlIO_open`, perl's `sysopen`, `PerlIO_fdopen` and `PerlIO_reopen`. The full prototype is as follows:

```
 PerlIO *       (*Open)(pTHX_ PerlIO_funcs *tab,
                        PerlIO_list_t *layers, IV n,
                        const char *mode,
                        int fd, int imode, int perm,
                        PerlIO *old,
                        int narg, SV **args);
```

Open should (perhaps indirectly) call `PerlIO_allocate()` to allocate a slot in the table and associate it with the layers information for the opened file, by calling `PerlIO_push`. The *layers* is an array of all the layers destined for the `PerlIO *`, and any arguments passed to them, *n* is the index into that array of the layer being called. The macro `PerlIOArg` will return a (possibly `NULL`) SV * for the argument passed to the layer.

The *mode* string is an "`fopen()`-like" string which would match the regular expression `/^[I#]?[rwa]\+?[bt]?$/`.

The `'I'` prefix is used during creation of `stdin`..`stderr` via special `PerlIO_fdopen` calls; the `'#'` prefix means that this is `sysopen` and that *imode* and *perm* should be passed to `PerlLIO_open3`; `'r'` means **r**ead, `'w'` means **w**rite and `'a'` means **a**ppend. The `'+'` suffix means that both reading and writing/appending are permitted. The `'b'` suffix means file should be binary, and `'t'` means it is text. (Almost all layers should do the IO in binary mode, and ignore the b/t bits. The `:crlf` layer should be pushed to handle the distinction.)

If *old* is not `NULL` then this is a `PerlIO_reopen`. Perl itself does not use this (yet?) and semantics are a little vague.

If *fd* not negative then it is the numeric file descriptor *fd*, which will be open in a manner compatible with the supplied mode string, the call is thus equivalent to `PerlIO_fdopen`. In this case *nargs* will be zero.

If *nargs* is greater than zero then it gives the number of arguments passed to `open`, otherwise it will be 1 if for example `PerlIO_open` was called. In simple cases SvPV_nolen(*args) is the pathname to open.

If a layer provides `Open()` it should normally call the `Open()` method of next layer down (if any) and then push itself on top if that succeeds. `PerlIOBase_open` is provided to do exactly that, so in most cases you don't have to write your own `Open()` method. If this method is not defined, other layers may have difficulty pushing themselves on top of it during open.

If `PerlIO_push` was performed and open has failed, it must `PerlIO_pop` itself, since if it's not, the layer won't be removed and may cause bad problems.

Returns `NULL` on failure.

Binmode

```
        IV        (*Binmode)(pTHX_ PerlIO *f);
```

Optional. Used when `:raw` layer is pushed (explicitly or as a result of binmode(FH)). If not present layer will be popped. If present should configure layer as binary (or pop itself) and return 0. If it returns -1 for error `binmode` will fail with layer still on the stack.

Getarg

```
        SV *      (*Getarg)(pTHX_ PerlIO *f,
                            CLONE_PARAMS *param, int flags);
```

Optional. If present should return an SV * representing the string argument passed to the layer when it was pushed. e.g. ":encoding(ascii)" would return an SvPV with value "ascii". (*param* and *flags* arguments can be ignored in most cases)

`Dup` uses `Getarg` to retrieve the argument originally passed to `Pushed`, so you must implement this function if your layer has an extra argument to `Pushed` and will ever be `Dupe`d.

Fileno

```
        IV        (*Fileno)(pTHX_ PerlIO *f);
```

Returns the Unix/Posix numeric file descriptor for the handle. Normally `PerlIOBase_fileno()` (which just asks next layer down) will suffice for this.

Returns -1 on error, which is considered to include the case where the layer cannot provide such a file descriptor.

Dup

```
        PerlIO * (*Dup)(pTHX_ PerlIO *f, PerlIO *o,
                        CLONE_PARAMS *param, int flags);
```

XXX: Needs more docs.

Used as part of the "clone" process when a thread is spawned (in which case param will be non-NULL) and when a stream is being duplicated via '&' in the `open`.

Similar to `Open`, returns PerlIO* on success, `NULL` on failure.

Read

```
        SSize_t (*Read)(pTHX_ PerlIO *f, void *vbuf, Size_t count);
```

Basic read operation.

Typically will call `Fill` and manipulate pointers (possibly via the API). `PerlIOBuf_read()` may be suitable for derived classes which provide "fast gets" methods.

Returns actual bytes read, or -1 on an error.

Unread

```
        SSize_t (*Unread)(pTHX_ PerlIO *f,
                          const void *vbuf, Size_t count);
```

A superset of stdio's `ungetc()`. Should arrange for future reads to see the bytes in `vbuf`. If there is no obviously better implementation then `PerlIOBase_unread()` provides the function by pushing a "fake" "pending" layer above the calling layer.

Returns the number of unread chars.

Write

```
        SSize_t (*Write)(PerlIO *f, const void *vbuf, Size_t count);
```

Basic write operation.

Returns bytes written or -1 on an error.

Seek

```
IV      (*Seek)(pTHX_ PerlIO *f, Off_t offset, int whence);
```

Position the file pointer. Should normally call its own `Flush` method and then the `Seek` method of next layer down.

Returns 0 on success, -1 on failure.

Tell

```
Off_t   (*Tell)(pTHX_ PerlIO *f);
```

Return the file pointer. May be based on layers cached concept of position to avoid overhead.

Returns -1 on failure to get the file pointer.

Close

```
IV      (*Close)(pTHX_ PerlIO *f);
```

Close the stream. Should normally call `PerlIOBase_close()` to flush itself and close layers below, and then deallocate any data structures (buffers, translation tables, ...) not held directly in the data structure.

Returns 0 on success, -1 on failure.

Flush

```
IV      (*Flush)(pTHX_ PerlIO *f);
```

Should make stream's state consistent with layers below. That is, any buffered write data should be written, and file position of lower layers adjusted for data read from below but not actually consumed. (Should perhaps `Unread()` such data to the lower layer.)

Returns 0 on success, -1 on failure.

Fill

```
IV      (*Fill)(pTHX_ PerlIO *f);
```

The buffer for this layer should be filled (for read) from layer below. When you "subclass" PerlIOBuf layer, you want to use its *_read* method and to supply your own fill method, which fills the PerlIOBuf's buffer.

Returns 0 on success, -1 on failure.

Eof

```
IV      (*Eof)(pTHX_ PerlIO *f);
```

Return end-of-file indicator. `PerlIOBase_eof()` is normally sufficient.

Returns 0 on end-of-file, 1 if not end-of-file, -1 on error.

Error

```
IV      (*Error)(pTHX_ PerlIO *f);
```

Return error indicator. `PerlIOBase_error()` is normally sufficient.

Returns 1 if there is an error (usually when `PERLIO_F_ERROR` is set), 0 otherwise.

Clearerr

```
void    (*Clearerr)(pTHX_ PerlIO *f);
```

Clear end-of-file and error indicators. Should call `PerlIOBase_clearerr()` to set the `PERLIO_F_XXXXX` flags, which may suffice.

Setlinebuf

```
void    (*Setlinebuf)(pTHX_ PerlIO *f);
```

Mark the stream as line buffered. `PerlIOBase_setlinebuf()` sets the PERLIO_F_LINEBUF flag and is normally sufficient.

Get_base

```
STDCHAR *       (*Get_base)(pTHX_ PerlIO *f);
```

Allocate (if not already done so) the read buffer for this layer and return pointer to it. Return NULL on failure.

Get_bufsiz

```
Size_t  (*Get_bufsiz)(pTHX_ PerlIO *f);
```

Return the number of bytes that last `Fill()` put in the buffer.

Get_ptr

```
STDCHAR *       (*Get_ptr)(pTHX_ PerlIO *f);
```

Return the current read pointer relative to this layer's buffer.

Get_cnt

```
SSize_t (*Get_cnt)(pTHX_ PerlIO *f);
```

Return the number of bytes left to be read in the current buffer.

Set_ptrcnt

```
void    (*Set_ptrcnt)(pTHX_ PerlIO *f,
                      STDCHAR *ptr, SSize_t cnt);
```

Adjust the read pointer and count of bytes to match `ptr` and/or `cnt`. The application (or layer above) must ensure they are consistent. (Checking is allowed by the paranoid.)

35.3.10 Utilities

To ask for the next layer down use PerlIONext(PerlIO *f).

To check that a PerlIO* is valid use PerlIOValid(PerlIO *f). (All this does is really just to check that the pointer is non-NULL and that the pointer behind that is non-NULL.)

PerlIOBase(PerlIO *f) returns the "Base" pointer, or in other words, the `PerlIOl*` pointer.

PerlIOSelf(PerlIO* f, type) return the PerlIOBase cast to a type.

Perl_PerlIO_or_Base(PerlIO* f, callback, base, failure, args) either calls the *callback* from the functions of the layer *f* (just by the name of the IO function, like "Read") with the *args*, or if there is no such callback, calls the *base* version of the callback with the same args, or if the f is invalid, set errno to EBADF and return *failure*.

Perl_PerlIO_or_fail(PerlIO* f, callback, failure, args) either calls the *callback* of the functions of the layer *f* with the *args*, or if there is no such callback, set errno to EINVAL. Or if the f is invalid, set errno to EBADF and return *failure*.

Perl_PerlIO_or_Base_void(PerlIO* f, callback, base, args) either calls the *callback* of the functions of the layer *f* with the *args*, or if there is no such callback, calls the *base* version of the callback with the same args, or if the f is invalid, set errno to EBADF.

Perl_PerlIO_or_fail_void(PerlIO* f, callback, args) either calls the *callback* of the functions of the layer *f* with the *args*, or if there is no such callback, set errno to EINVAL. Or if the f is invalid, set errno to EBADF.

35.3.11 Implementing PerlIO Layers

If you find the implementation document unclear or not sufficient, look at the existing PerlIO layer implementations, which include:

- C implementations

  The `perlio.c` and `perliol.h` in the Perl core implement the "unix", "perlio", "stdio", "crlf", "utf8", "byte", "raw", "pending" layers, and also the "mmap" and "win32" layers if applicable. (The "win32" is currently unfinished and unused, to see what is used instead in Win32, see Section “Querying the layers of filehandles” in `PerlIO` .)

  PerlIO::encoding, PerlIO::scalar, PerlIO::via in the Perl core.

  PerlIO::gzip and APR::PerlIO (mod_perl 2.0) on CPAN.

- Perl implementations

  PerlIO::via::QuotedPrint in the Perl core and PerlIO::via::* on CPAN.

If you are creating a PerlIO layer, you may want to be lazy, in other words, implement only the methods that interest you. The other methods you can either replace with the "blank" methods

```
PerlIOBase_noop_ok
PerlIOBase_noop_fail
```

(which do nothing, and return zero and -1, respectively) or for certain methods you may assume a default behaviour by using a NULL method. The Open method looks for help in the 'parent' layer. The following table summarizes the behaviour:

method	behaviour with NULL
Clearerr	PerlIOBase_clearerr
Close	PerlIOBase_close
Dup	PerlIOBase_dup
Eof	PerlIOBase_eof
Error	PerlIOBase_error
Fileno	PerlIOBase_fileno
Fill	FAILURE
Flush	SUCCESS
Getarg	SUCCESS
Get_base	FAILURE
Get_bufsiz	FAILURE
Get_cnt	FAILURE
Get_ptr	FAILURE
Open	INHERITED
Popped	SUCCESS

```
   Pushed      SUCCESS
   Read        PerlIOBase_read
   Seek        FAILURE
   Set_cnt     FAILURE
   Set_ptrcnt  FAILURE
   Setlinebuf  PerlIOBase_setlinebuf
   Tell        FAILURE
   Unread      PerlIOBase_unread
   Write       FAILURE

FAILURE        Set errno (to EINVAL in Unixish, to LIB$_INVARG in VMS) and
               return -1 (for numeric return values) or NULL (for pointers)
INHERITED      Inherited from the layer below
SUCCESS        Return 0 (for numeric return values) or a pointer
```

35.3.12 Core Layers

The file `perlio.c` provides the following layers:

"unix"

A basic non-buffered layer which calls Unix/POSIX `read()`, `write()`, `lseek()`, `close()`. No buffering. Even on platforms that distinguish between O_TEXT and O_BINARY this layer is always O_BINARY.

"perlio"

A very complete generic buffering layer which provides the whole of PerlIO API. It is also intended to be used as a "base class" for other layers. (For example its `Read()` method is implemented in terms of the `Get_cnt()`/`Get_ptr()`/`Set_ptrcnt()` methods).

"perlio" over "unix" provides a complete replacement for stdio as seen via PerlIO API. This is the default for USE_PERLIO when system's stdio does not permit perl's "fast gets" access, and which do not distinguish between `O_TEXT` and `O_BINARY`.

"stdio"

A layer which provides the PerlIO API via the layer scheme, but implements it by calling system's stdio. This is (currently) the default if system's stdio provides sufficient access to allow perl's "fast gets" access and which do not distinguish between `O_TEXT` and `O_BINARY`.

"crlf"

A layer derived using "perlio" as a base class. It provides Win32-like "\n" to CR,LF translation. Can either be applied above "perlio" or serve as the buffer layer itself. "crlf" over "unix" is the default if system distinguishes between `O_TEXT` and `O_BINARY` opens. (At some point "unix" will be replaced by a "native" Win32 IO layer on that platform, as Win32's read/write layer has various drawbacks.) The "crlf" layer is a reasonable model for a layer which transforms data in some way.

"mmap"

If Configure detects `mmap()` functions this layer is provided (with "perlio" as a "base") which does "read" operations by mmap()ing the file. Performance improvement is marginal on modern systems, so it is mainly there as a proof of concept. It is likely to be unbundled from the core at some point. The "mmap" layer is a reasonable model for a minimalist "derived" layer.

"pending"

An "internal" derivative of "perlio" which can be used to provide Unread() function for layers which have no buffer or cannot be bothered. (Basically this layer's `Fill()` pops itself off the stack and so resumes reading from layer below.)

"raw"

A dummy layer which never exists on the layer stack. Instead when "pushed" it actually pops the stack removing itself, it then calls Binmode function table entry on all the layers in the stack - normally this (via PerlIOBase_binmode) removes any layers which do not have `PERLIO_K_RAW` bit set. Layers can modify that behaviour by defining their own Binmode entry.

"utf8"

Another dummy layer. When pushed it pops itself and sets the `PERLIO_F_UTF8` flag on the layer which was (and now is once more) the top of the stack.

In addition `perlio.c` also provides a number of `PerlIOBase_xxxx()` functions which are intended to be used in the table slots of classes which do not need to do anything special for a particular method.

35.3.13 Extension Layers

Layers can be made available by extension modules. When an unknown layer is encountered the PerlIO code will perform the equivalent of :

```
use PerlIO 'layer';
```

Where *layer* is the unknown layer. `PerlIO.pm` will then attempt to:

```
require PerlIO::layer;
```

If after that process the layer is still not defined then the `open` will fail.

The following extension layers are bundled with perl:

":encoding"

```
use Encoding;
```

makes this layer available, although `PerlIO.pm` "knows" where to find it. It is an example of a layer which takes an argument as it is called thus:

```
open( $fh, "<:encoding(iso-8859-7)", $pathname );
```

":scalar"

Provides support for reading data from and writing data to a scalar.

```
open( $fh, "+<:scalar", \$scalar );
```

When a handle is so opened, then reads get bytes from the string value of *$scalar*, and writes change the value. In both cases the position in *$scalar* starts as zero but can be altered via `seek`, and determined via `tell`.

Please note that this layer is implied when calling open() thus:

```
open( $fh, "+<", \$scalar );
```

":via"

Provided to allow layers to be implemented as Perl code. For instance:

```
use PerlIO::via::StripHTML;
open( my $fh, "<:via(StripHTML)", "index.html" );
```

See `PerlIO-via` for details.

35.4 TODO

Things that need to be done to improve this document.

- Explain how to make a valid fh without going through open()(i.e. apply a layer). For example if the file is not opened through perl, but we want to get back a fh, like it was opened by Perl.

  How PerlIO_apply_layera fits in, where its docs, was it made public?

  Currently the example could be something like this:

```
PerlIO *foo_to_PerlIO(pTHX_ char *mode, ...)
{
    char *mode; /* "w", "r", etc */
    const char *layers = ":APR"; /* the layer name */
    PerlIO *f = PerlIO_allocate(aTHX);
    if (!f) {
        return NULL;
    }

    PerlIO_apply_layers(aTHX_ f, mode, layers);

    if (f) {
        PerlIOAPR *st = PerlIOSelf(f, PerlIOAPR);
        /* fill in the st struct, as in _open() */
        st->file = file;
        PerlIOBase(f)->flags |= PERLIO_F_OPEN;

        return f;
    }
    return NULL;
}
```

- fix/add the documentation in places marked as XXX.
- The handling of errors by the layer is not specified. e.g. when $! should be set explicitly, when the error handling should be just delegated to the top layer.

  Probably give some hints on using SETERRNO() or pointers to where they can be found.
- I think it would help to give some concrete examples to make it easier to understand the API. Of course I agree that the API has to be concise, but since there is no second

document that is more of a guide, I think that it'd make it easier to start with the doc which is an API, but has examples in it in places where things are unclear, to a person who is not a PerlIO guru (yet).

36 perlipc

36.1 NAME

perlipc - Perl interprocess communication (signals, fifos, pipes, safe subprocesses, sockets, and semaphores)

36.2 DESCRIPTION

The basic IPC facilities of Perl are built out of the good old Unix signals, named pipes, pipe opens, the Berkeley socket routines, and SysV IPC calls. Each is used in slightly different situations.

36.3 Signals

Perl uses a simple signal handling model: the %SIG hash contains names or references of user-installed signal handlers. These handlers will be called with an argument which is the name of the signal that triggered it. A signal may be generated intentionally from a particular keyboard sequence like control-C or control-Z, sent to you from another process, or triggered automatically by the kernel when special events transpire, like a child process exiting, your own process running out of stack space, or hitting a process file-size limit.

For example, to trap an interrupt signal, set up a handler like this:

```
our $shucks;

sub catch_zap {
    my $signame = shift;
    $shucks++;
    die "Somebody sent me a SIG$signame";
}
$SIG{INT} = __PACKAGE__ . "::catch_zap";
$SIG{INT} = \&catch_zap;  # best strategy
```

Prior to Perl 5.8.0 it was necessary to do as little as you possibly could in your handler; notice how all we do is set a global variable and then raise an exception. That's because on most systems, libraries are not re-entrant; particularly, memory allocation and I/O routines are not. That meant that doing nearly *anything* in your handler could in theory trigger a memory fault and subsequent core dump - see Section 36.3.2 [Deferred Signals (Safe Signals)], page 671 below.

The names of the signals are the ones listed out by `kill -l` on your system, or you can retrieve them using the CPAN module `IPC-Signal`.

You may also choose to assign the strings `"IGNORE"` or `"DEFAULT"` as the handler, in which case Perl will try to discard the signal or do the default thing.

On most Unix platforms, the `CHLD` (sometimes also known as `CLD`) signal has special behavior with respect to a value of `"IGNORE"`. Setting `$SIG{CHLD}` to `"IGNORE"` on such a platform has the effect of not creating zombie processes when the parent process fails to `wait()` on its child processes (i.e., child processes are automatically reaped). Calling `wait()` with `$SIG{CHLD}` set to `"IGNORE"` usually returns `-1` on such platforms.

Some signals can be neither trapped nor ignored, such as the KILL and STOP (but not the TSTP) signals. Note that ignoring signals makes them disappear. If you only want them blocked temporarily without them getting lost you'll have to use POSIX' sigprocmask.

Sending a signal to a negative process ID means that you send the signal to the entire Unix process group. This code sends a hang-up signal to all processes in the current process group, and also sets $SIG{HUP} to "IGNORE" so it doesn't kill itself:

```
# block scope for local
{
    local $SIG{HUP} = "IGNORE";
    kill HUP => -$$;
    # snazzy writing of: kill("HUP", -$$)
}
```

Another interesting signal to send is signal number zero. This doesn't actually affect a child process, but instead checks whether it's alive or has changed its UIDs.

```
unless (kill 0 => $kid_pid) {
    warn "something wicked happened to $kid_pid";
}
```

Signal number zero may fail because you lack permission to send the signal when directed at a process whose real or saved UID is not identical to the real or effective UID of the sending process, even though the process is alive. You may be able to determine the cause of failure using `$!` or `%!`.

```
unless (kill(0 => $pid) || $!{EPERM}) {
    warn "$pid looks dead";
}
```

You might also want to employ anonymous functions for simple signal handlers:

```
$SIG{INT} = sub { die "\nOutta here!\n" };
```

SIGCHLD handlers require some special care. If a second child dies while in the signal handler caused by the first death, we won't get another signal. So must loop here else we will leave the unreaped child as a zombie. And the next time two children die we get another zombie. And so on.

```
use POSIX ":sys_wait_h";
$SIG{CHLD} = sub {
    while ((my $child = waitpid(-1, WNOHANG)) > 0) {
        $Kid_Status{$child} = $?;
    }
};
# do something that forks...
```

Be careful: qx(), system(), and some modules for calling external commands do a fork(), then wait() for the result. Thus, your signal handler will be called. Because wait() was already called by system() or qx(), the wait() in the signal handler will see no more zombies and will therefore block.

The best way to prevent this issue is to use waitpid(), as in the following example:

```
use POSIX ":sys_wait_h"; # for nonblocking read
```

```
my %children;

$SIG{CHLD} = sub {
    # don’t change $! and $? outside handler
    local ($!, $?);
    while ( (my $pid = waitpid(-1, WNOHANG)) > 0 ) {
        delete $children{$pid};
        cleanup_child($pid, $?);
    }
};

while (1) {
    my $pid = fork();
    die "cannot fork" unless defined $pid;
    if ($pid == 0) {
        # ...
        exit 0;
    } else {
        $children{$pid}=1;
        # ...
        system($command);
        # ...
   }
}
```

Signal handling is also used for timeouts in Unix. While safely protected within an `eval{}` block, you set a signal handler to trap alarm signals and then schedule to have one delivered to you in some number of seconds. Then try your blocking operation, clearing the alarm when it’s done but not before you’ve exited your `eval{}` block. If it goes off, you’ll use die() to jump out of the block.

Here’s an example:

```
my $ALARM_EXCEPTION = "alarm clock restart";
eval {
    local $SIG{ALRM} = sub { die $ALARM_EXCEPTION };
    alarm 10;
    flock(FH, 2)    # blocking write lock
                    || die "cannot flock: $!";
    alarm 0;
};
if ($@ && $@ !~ quotemeta($ALARM_EXCEPTION)) { die }
```

If the operation being timed out is system() or qx(), this technique is liable to generate zombies. If this matters to you, you’ll need to do your own fork() and exec(), and kill the errant child process.

For more complex signal handling, you might see the standard POSIX module. Lamentably, this is almost entirely undocumented, but the `t/lib/posix.t` file from the Perl source distribution has some examples in it.

36.3.1 Handling the SIGHUP Signal in Daemons

A process that usually starts when the system boots and shuts down when the system is shut down is called a daemon (Disk And Execution MONitor). If a daemon process has a configuration file which is modified after the process has been started, there should be a way to tell that process to reread its configuration file without stopping the process. Many daemons provide this mechanism using a SIGHUP signal handler. When you want to tell the daemon to reread the file, simply send it the SIGHUP signal.

The following example implements a simple daemon, which restarts itself every time the SIGHUP signal is received. The actual code is located in the subroutine code(), which just prints some debugging info to show that it works; it should be replaced with the real code.

```
#!/usr/bin/perl

use strict;
use warnings;

use POSIX ();
use FindBin ();
use File::Basename ();
use File::Spec::Functions qw(catfile);

$| = 1;

# make the daemon cross-platform, so exec always calls the script
# itself with the right path, no matter how the script was invoked.
my $script = File::Basename::basename($0);
my $SELF  = catfile($FindBin::Bin, $script);

# POSIX unmasks the sigprocmask properly
$SIG{HUP} = sub {
    print "got SIGHUP\n";
    exec($SELF, @ARGV)        || die "$0: couldn't restart: $!";
};

code();

sub code {
    print "PID: $$\n";
    print "ARGV: @ARGV\n";
    my $count = 0;
    while (1) {
        sleep 2;
        print ++$count, "\n";
    }
}
```

36.3.2 Deferred Signals (Safe Signals)

Before Perl 5.8.0, installing Perl code to deal with signals exposed you to danger from two things. First, few system library functions are re-entrant. If the signal interrupts while Perl is executing one function (like malloc(3) or printf(3)), and your signal handler then calls the same function again, you could get unpredictable behavior–often, a core dump. Second, Perl isn't itself re-entrant at the lowest levels. If the signal interrupts Perl while Perl is changing its own internal data structures, similarly unpredictable behavior may result.

There were two things you could do, knowing this: be paranoid or be pragmatic. The paranoid approach was to do as little as possible in your signal handler. Set an existing integer variable that already has a value, and return. This doesn't help you if you're in a slow system call, which will just restart. That means you have to `die` to longjmp(3) out of the handler. Even this is a little cavalier for the true paranoiac, who avoids `die` in a handler because the system *is* out to get you. The pragmatic approach was to say "I know the risks, but prefer the convenience", and to do anything you wanted in your signal handler, and be prepared to clean up core dumps now and again.

Perl 5.8.0 and later avoid these problems by "deferring" signals. That is, when the signal is delivered to the process by the system (to the C code that implements Perl) a flag is set, and the handler returns immediately. Then at strategic "safe" points in the Perl interpreter (e.g. when it is about to execute a new opcode) the flags are checked and the Perl level handler from %SIG is executed. The "deferred" scheme allows much more flexibility in the coding of signal handlers as we know the Perl interpreter is in a safe state, and that we are not in a system library function when the handler is called. However the implementation does differ from previous Perls in the following ways:

Long-running opcodes

As the Perl interpreter looks at signal flags only when it is about to execute a new opcode, a signal that arrives during a long-running opcode (e.g. a regular expression operation on a very large string) will not be seen until the current opcode completes.

If a signal of any given type fires multiple times during an opcode (such as from a fine-grained timer), the handler for that signal will be called only once, after the opcode completes; all other instances will be discarded. Furthermore, if your system's signal queue gets flooded to the point that there are signals that have been raised but not yet caught (and thus not deferred) at the time an opcode completes, those signals may well be caught and deferred during subsequent opcodes, with sometimes surprising results. For example, you may see alarms delivered even after calling `alarm(0)` as the latter stops the raising of alarms but does not cancel the delivery of alarms raised but not yet caught. Do not depend on the behaviors described in this paragraph as they are side effects of the current implementation and may change in future versions of Perl.

Interrupting IO

When a signal is delivered (e.g., SIGINT from a control-C) the operating system breaks into IO operations like *read*(2), which is used to implement Perl's readline() function, the `<>` operator. On older Perls the handler was called immediately (and as `read` is not "unsafe", this worked well). With the "deferred" scheme the handler is *not* called immediately, and if Perl is using the system's `stdio` library that library may restart the `read` without returning to Perl to give it a chance to call the %SIG handler. If this happens on your system the solution is to use the `:perlio` layer to do IO–at least on those handles that you want to be able to break into with signals. (The `:perlio` layer checks the signal flags and calls %SIG handlers before resuming IO operation.)

The default in Perl 5.8.0 and later is to automatically use the `:perlio` layer.

Note that it is not advisable to access a file handle within a signal handler where that signal has interrupted an I/O operation on that same handle. While perl will at least try hard not to crash, there are no guarantees of data integrity; for example, some data might get dropped or written twice.

Some networking library functions like gethostbyname() are known to have their own implementations of timeouts which may conflict with your timeouts. If you have problems with such functions, try using the POSIX sigaction() function, which bypasses Perl safe signals. Be warned that this does subject you to possible memory corruption, as described above.

Instead of setting `$SIG{ALRM}`:

```
   local $SIG{ALRM} = sub { die "alarm" };
```

try something like the following:

```
  use POSIX qw(SIGALRM);
  POSIX::sigaction(SIGALRM, POSIX::SigAction->new(sub { die "alarm" }))
          || die "Error setting SIGALRM handler: $!\n";
```

Another way to disable the safe signal behavior locally is to use the `Perl::Unsafe::Signals` module from CPAN, which affects all signals.

Restartable system calls

On systems that supported it, older versions of Perl used the SA_RESTART flag when installing %SIG handlers. This meant that restartable system calls would continue rather than returning when a signal arrived. In order to deliver deferred signals promptly, Perl 5.8.0 and later do *not* use SA_RESTART. Consequently, restartable system calls can fail (with $! set to `EINTR`) in places where they previously would have succeeded.

The default `:perlio` layer retries `read`, `write` and `close` as described above; interrupted `wait` and `waitpid` calls will always be retried.

Signals as "faults"

Certain signals like SEGV, ILL, and BUS are generated by virtual memory addressing errors and similar "faults". These are normally fatal: there is little a Perl-level handler can do with them. So Perl delivers them immediately rather than attempting to defer them.

Signals triggered by operating system state

On some operating systems certain signal handlers are supposed to "do something" before returning. One example can be CHLD or CLD, which indicates a child process has completed. On some operating systems the signal handler is expected to `wait` for the completed child process. On such systems the deferred signal scheme will not work for those signals: it does not do the `wait`. Again the failure will look like a loop as the operating system will reissue the signal because there are completed child processes that have not yet been `wait`ed for.

If you want the old signal behavior back despite possible memory corruption, set the environment variable `PERL_SIGNALS` to `"unsafe"`. This feature first appeared in Perl 5.8.1.

36.4 Named Pipes

A named pipe (often referred to as a FIFO) is an old Unix IPC mechanism for processes communicating on the same machine. It works just like regular anonymous pipes, except that the processes rendezvous using a filename and need not be related.

To create a named pipe, use the `POSIX::mkfifo()` function.

```
use POSIX qw(mkfifo);
mkfifo($path, 0700)     ||  die "mkfifo $path failed: $!";
```

You can also use the Unix command mknod(1), or on some systems, mkfifo(1). These may not be in your normal path, though.

```
# system return val is backwards, so && not ||
#
$ENV{PATH} .= ":/etc:/usr/etc";
if  (      system("mknod",  $path, "p")
        && system("mkfifo", $path) )
{
    die "mk{nod,fifo} $path failed";
}
```

A fifo is convenient when you want to connect a process to an unrelated one. When you open a fifo, the program will block until there's something on the other end.

For example, let's say you'd like to have your `.signature` file be a named pipe that has a Perl program on the other end. Now every time any program (like a mailer, news reader, finger program, etc.) tries to read from that file, the reading program will read the new signature from your program. We'll use the pipe-checking file-test operator, **-p**, to find out whether anyone (or anything) has accidentally removed our fifo.

```
chdir();    # go home
my $FIFO = ".signature";

while (1) {
    unless (-p $FIFO) {
        unlink $FIFO;   # discard any failure, will catch later
        require POSIX;  # delayed loading of heavy module
        POSIX::mkfifo($FIFO, 0700)
                            || die "can't mkfifo $FIFO: $!";
    }

    # next line blocks till there's a reader
    open (FIFO, "> $FIFO")  || die "can't open $FIFO: $!";
    print FIFO "John Smith (smith\@host.org)\n", `fortune -s`;
    close(FIFO)             || die "can't close $FIFO: $!";
```

```
        sleep 2;                # to avoid dup signals
    }
```

36.5 Using open() for IPC

Perl's basic open() statement can also be used for unidirectional interprocess communication by either appending or prepending a pipe symbol to the second argument to open(). Here's how to start something up in a child process you intend to write to:

```
    open(SPOOLER, "| cat -v | lpr -h 2>/dev/null")
                        || die "can't fork: $!";
    local $SIG{PIPE} = sub { die "spooler pipe broke" };
    print SPOOLER "stuff\n";
    close SPOOLER       || die "bad spool: $! $?";
```

And here's how to start up a child process you intend to read from:

```
    open(STATUS, "netstat -an 2>&1 |")
                        || die "can't fork: $!";
    while (<STATUS>) {
        next if /^(tcp|udp)/;
        print;
    }
    close STATUS        || die "bad netstat: $! $?";
```

If one can be sure that a particular program is a Perl script expecting filenames in @ARGV, the clever programmer can write something like this:

```
    % program f1 "cmd1|" - f2 "cmd2|" f3 < tmpfile
```

and no matter which sort of shell it's called from, the Perl program will read from the file `f1`, the process `cmd1`, standard input (`tmpfile` in this case), the `f2` file, the `cmd2` command, and finally the `f3` file. Pretty nifty, eh?

You might notice that you could use backticks for much the same effect as opening a pipe for reading:

```
    print grep { !/^(tcp|udp)/ } `netstat -an 2>&1`;
    die "bad netstatus ($?)" if $?;
```

While this is true on the surface, it's much more efficient to process the file one line or record at a time because then you don't have to read the whole thing into memory at once. It also gives you finer control of the whole process, letting you kill off the child process early if you'd like.

Be careful to check the return values from both open() and close(). If you're *writing* to a pipe, you should also trap SIGPIPE. Otherwise, think of what happens when you start up a pipe to a command that doesn't exist: the open() will in all likelihood succeed (it only reflects the fork()'s success), but then your output will fail–spectacularly. Perl can't know whether the command worked, because your command is actually running in a separate process whose exec() might have failed. Therefore, while readers of bogus commands return just a quick EOF, writers to bogus commands will get hit with a signal, which they'd best be prepared to handle. Consider:

```
    open(FH, "|bogus")      || die "can't fork: $!";
    print FH "bang\n";      #  neither necessary nor sufficient
```

```
                             #  to check print retval!
    close(FH)                || die "can't close: $!";
```

The reason for not checking the return value from print() is because of pipe buffering; physical writes are delayed. That won't blow up until the close, and it will blow up with a SIGPIPE. To catch it, you could use this:

```
    $SIG{PIPE} = "IGNORE";
    open(FH, "|bogus")  || die "can't fork: $!";
    print FH "bang\n";
    close(FH)           || die "can't close: status=$?";
```

36.5.1 Filehandles

Both the main process and any child processes it forks share the same STDIN, STDOUT, and STDERR filehandles. If both processes try to access them at once, strange things can happen. You may also want to close or reopen the filehandles for the child. You can get around this by opening your pipe with open(), but on some systems this means that the child process cannot outlive the parent.

36.5.2 Background Processes

You can run a command in the background with:

```
    system("cmd &");
```

The command's STDOUT and STDERR (and possibly STDIN, depending on your shell) will be the same as the parent's. You won't need to catch SIGCHLD because of the double-fork taking place; see below for details.

36.5.3 Complete Dissociation of Child from Parent

In some cases (starting server processes, for instance) you'll want to completely dissociate the child process from the parent. This is often called daemonization. A well-behaved daemon will also chdir() to the root directory so it doesn't prevent unmounting the filesystem containing the directory from which it was launched, and redirect its standard file descriptors from and to /dev/null so that random output doesn't wind up on the user's terminal.

```
    use POSIX "setsid";

    sub daemonize {
        chdir("/")                      || die "can't chdir to /: $!";
        open(STDIN,  "< /dev/null")     || die "can't read /dev/null: $!";
        open(STDOUT, "> /dev/null")     || die "can't write to /dev/null: $!";
        defined(my $pid = fork())       || die "can't fork: $!";
        exit if $pid;                   # non-zero now means I am the parent
        (setsid() != -1)                || die "Can't start a new session: $!";
        open(STDERR, ">&STDOUT")        || die "can't dup stdout: $!";
    }
```

The fork() has to come before the setsid() to ensure you aren't a process group leader; the setsid() will fail if you are. If your system doesn't have the setsid() function, open /dev/tty and use the TIOCNOTTY ioctl() on it instead. See tty(4) for details.

Non-Unix users should check their *Your_OS*::Process module for other possible solutions.

36.5.4 Safe Pipe Opens

Another interesting approach to IPC is making your single program go multiprocess and communicate between–or even amongst–yourselves. The open() function will accept a file argument of either "-|" or "|-" to do a very interesting thing: it forks a child connected to the filehandle you've opened. The child is running the same program as the parent. This is useful for safely opening a file when running under an assumed UID or GID, for example. If you open a pipe *to* minus, you can write to the filehandle you opened and your kid will find it in *his* STDIN. If you open a pipe *from* minus, you can read from the filehandle you opened whatever your kid writes to *his* STDOUT.

```
use English;
my $PRECIOUS = "/path/to/some/safe/file";
my $sleep_count;
my $pid;

do {
    $pid = open(KID_TO_WRITE, "|-");
    unless (defined $pid) {
        warn "cannot fork: $!";
        die "bailing out" if $sleep_count++ > 6;
        sleep 10;
    }
} until defined $pid;

if ($pid) {                 # I am the parent
    print KID_TO_WRITE @some_data;
    close(KID_TO_WRITE)     || warn "kid exited $?";
} else {                    # I am the child
    # drop permissions in setuid and/or setgid programs:
    ($EUID, $EGID) = ($UID, $GID);
    open (OUTFILE, "> $PRECIOUS")
                            || die "can't open $PRECIOUS: $!";
    while (<STDIN>) {
        print OUTFILE;      # child's STDIN is parent's KID_TO_WRITE
    }
    close(OUTFILE)          || die "can't close $PRECIOUS: $!";
    exit(0);                # don't forget this!!
}
```

Another common use for this construct is when you need to execute something without the shell's interference. With system(), it's straightforward, but you can't use a pipe open or backticks safely. That's because there's no way to stop the shell from getting its hands on your arguments. Instead, use lower-level control to call exec() directly.

Here's a safe backtick or pipe open for read:

```
my $pid = open(KID_TO_READ, "-|");
```

```
defined($pid)           || die "can't fork: $!";

if ($pid) {             # parent
    while (<KID_TO_READ>) {
                        # do something interesting
    }
    close(KID_TO_READ)  || warn "kid exited $?";

} else {                # child
    ($EUID, $EGID) = ($UID, $GID); # suid only
    exec($program, @options, @args)
                        || die "can't exec program: $!";
    # NOTREACHED
}
```

And here's a safe pipe open for writing:

```
my $pid = open(KID_TO_WRITE, "|-");
defined($pid)           || die "can't fork: $!";

$SIG{PIPE} = sub { die "whoops, $program pipe broke" };

if ($pid) {             # parent
    print KID_TO_WRITE @data;
    close(KID_TO_WRITE) || warn "kid exited $?";

} else {                # child
    ($EUID, $EGID) = ($UID, $GID);
    exec($program, @options, @args)
                        || die "can't exec program: $!";
    # NOTREACHED
}
```

It is very easy to dead-lock a process using this form of open(), or indeed with any use of pipe() with multiple subprocesses. The example above is "safe" because it is simple and calls exec(). See Section 36.5.5 [Avoiding Pipe Deadlocks], page 679 for general safety principles, but there are extra gotchas with Safe Pipe Opens.

In particular, if you opened the pipe using `open FH, "|-"`, then you cannot simply use close() in the parent process to close an unwanted writer. Consider this code:

```
my $pid = open(WRITER, "|-");        # fork open a kid
defined($pid)               || die "first fork failed: $!";
if ($pid) {
    if (my $sub_pid = fork()) {
        defined($sub_pid)   || die "second fork failed: $!";
        close(WRITER)       || die "couldn't close WRITER: $!";
        # now do something else...
    }
    else {
        # first write to WRITER
        # ...
        # then when finished
        close(WRITER)       || die "couldn't close WRITER: $!";
        exit(0);
    }
}
else {
    # first do something with STDIN, then
    exit(0);
}
```

In the example above, the true parent does not want to write to the WRITER filehandle, so it closes it. However, because WRITER was opened using `open FH, "|-"`, it has a special behavior: closing it calls waitpid() (see ⟨undefined⟩ [perlfunc waitpid], page ⟨undefined⟩), which waits for the subprocess to exit. If the child process ends up waiting for something happening in the section marked "do something else", you have deadlock.

This can also be a problem with intermediate subprocesses in more complicated code, which will call waitpid() on all open filehandles during global destruction–in no predictable order.

To solve this, you must manually use pipe(), fork(), and the form of open() which sets one file descriptor to another, as shown below:

```
pipe(READER, WRITER)        || die "pipe failed: $!";
$pid = fork();
defined($pid)               || die "first fork failed: $!";
if ($pid) {
    close READER;
    if (my $sub_pid = fork()) {
        defined($sub_pid)   || die "first fork failed: $!";
        close(WRITER)       || die "can't close WRITER: $!";
    }
    else {
        # write to WRITER...
        # ...
        # then  when finished
        close(WRITER)       || die "can't close WRITER: $!";
        exit(0);
    }
    # write to WRITER...
}
else {
    open(STDIN, "<&READER") || die "can't reopen STDIN: $!";
    close(WRITER)           || die "can't close WRITER: $!";
    # do something...
    exit(0);
}
```

Since Perl 5.8.0, you can also use the list form of `open` for pipes. This is preferred when you wish to avoid having the shell interpret metacharacters that may be in your command string.

So for example, instead of using:

```
open(PS_PIPE, "ps aux|")    || die "can't open ps pipe: $!";
```

One would use either of these:

```
open(PS_PIPE, "-|", "ps", "aux")
                            || die "can't open ps pipe: $!";

@ps_args = qw[ ps aux ];
open(PS_PIPE, "-|", @ps_args)
                            || die "can't open @ps_args|: $!";
```

Because there are more than three arguments to open(), forks the ps(1) command *without* spawning a shell, and reads its standard output via the `PS_PIPE` filehandle. The corresponding syntax to *write* to command pipes is to use `"|-"` in place of `"-|"`.

This was admittedly a rather silly example, because you're using string literals whose content is perfectly safe. There is therefore no cause to resort to the harder-to-read, multi-argument form of pipe open(). However, whenever you cannot be assured that the program arguments are free of shell metacharacters, the fancier form of open() should be used. For example:

```
@grep_args = ("egrep", "-i", $some_pattern, @many_files);
open(GREP_PIPE, "-|", @grep_args)
                    || die "can't open @grep_args|: $!";
```

Here the multi-argument form of pipe open() is preferred because the pattern and indeed even the filenames themselves might hold metacharacters.

Be aware that these operations are full Unix forks, which means they may not be correctly implemented on all alien systems.

36.5.5 Avoiding Pipe Deadlocks

Whenever you have more than one subprocess, you must be careful that each closes whichever half of any pipes created for interprocess communication it is not using. This is because any child process reading from the pipe and expecting an EOF will never receive it, and therefore never exit. A single process closing a pipe is not enough to close it; the last process with the pipe open must close it for it to read EOF.

Certain built-in Unix features help prevent this most of the time. For instance, filehandles have a "close on exec" flag, which is set *en masse* under control of the `$^F` variable. This is so any filehandles you didn't explicitly route to the STDIN, STDOUT or STDERR of a child *program* will be automatically closed.

Always explicitly and immediately call close() on the writable end of any pipe, unless that process is actually writing to it. Even if you don't explicitly call close(), Perl will still close() all filehandles during global destruction. As previously discussed, if those filehandles have been opened with Safe Pipe Open, this will result in calling waitpid(), which may again deadlock.

36.5.6 Bidirectional Communication with Another Process

While this works reasonably well for unidirectional communication, what about bidirectional communication? The most obvious approach doesn't work:

```
# THIS DOES NOT WORK!!
open(PROG_FOR_READING_AND_WRITING, "| some program |")
```

If you forget to `use warnings`, you'll miss out entirely on the helpful diagnostic message:

```
Can't do bidirectional pipe at -e line 1.
```

If you really want to, you can use the standard open2() from the `IPC::Open2` module to catch both ends. There's also an open3() in `IPC::Open3` for tridirectional I/O so you can also catch your child's STDERR, but doing so would then require an awkward select() loop and wouldn't allow you to use normal Perl input operations.

If you look at its source, you'll see that open2() uses low-level primitives like the pipe() and exec() syscalls to create all the connections. Although it might have been more efficient by using socketpair(), this would have been even less portable than it already is. The open2() and open3() functions are unlikely to work anywhere except on a Unix system, or at least one purporting POSIX compliance.

Here's an example of using open2():

```
use FileHandle;
use IPC::Open2;
$pid = open2(*Reader, *Writer, "cat -un");
print Writer "stuff\n";
$got = <Reader>;
```

The problem with this is that buffering is really going to ruin your day. Even though your `Writer` filehandle is auto-flushed so the process on the other end gets your data in a timely manner, you can't usually do anything to force that process to give its data to you in a similarly quick fashion. In this special case, we could actually so, because we gave *cat* a **-u** flag to make it unbuffered. But very few commands are designed to operate over pipes, so this seldom works unless you yourself wrote the program on the other end of the double-ended pipe.

A solution to this is to use a library which uses pseudottys to make your program behave more reasonably. This way you don't have to have control over the source code of the program you're using. The `Expect` module from CPAN also addresses this kind of thing. This module requires two other modules from CPAN, `IO::Pty` and `IO::Stty`. It sets up a pseudo terminal to interact with programs that insist on talking to the terminal device driver. If your system is supported, this may be your best bet.

36.5.7 Bidirectional Communication with Yourself

If you want, you may make low-level pipe() and fork() syscalls to stitch this together by hand. This example only talks to itself, but you could reopen the appropriate handles to STDIN and STDOUT and call other processes. (The following example lacks proper error checking.)

```
#!/usr/bin/perl -w
# pipe1 - bidirectional communication using two pipe pairs
#         designed for the socketpair-challenged
```

```
use IO::Handle;               # thousands of lines just for autoflush :-(
pipe(PARENT_RDR, CHILD_WTR);  # XXX: check failure?
pipe(CHILD_RDR,  PARENT_WTR); # XXX: check failure?
CHILD_WTR->autoflush(1);
PARENT_WTR->autoflush(1);

if ($pid = fork()) {
    close PARENT_RDR;
    close PARENT_WTR;
    print CHILD_WTR "Parent Pid $$ is sending this\n";
    chomp($line = <CHILD_RDR>);
    print "Parent Pid $$ just read this: '$line'\n";
    close CHILD_RDR; close CHILD_WTR;
    waitpid($pid, 0);
} else {
    die "cannot fork: $!" unless defined $pid;
    close CHILD_RDR;
    close CHILD_WTR;
    chomp($line = <PARENT_RDR>);
    print "Child Pid $$ just read this: '$line'\n";
    print PARENT_WTR "Child Pid $$ is sending this\n";
    close PARENT_RDR;
    close PARENT_WTR;
    exit(0);
}
```

But you don't actually have to make two pipe calls. If you have the socketpair() system call, it will do this all for you.

```
#!/usr/bin/perl -w
# pipe2 - bidirectional communication using socketpair
#   "the best ones always go both ways"

use Socket;
use IO::Handle;  # thousands of lines just for autoflush :-(

# We say AF_UNIX because although *_LOCAL is the
# POSIX 1003.1g form of the constant, many machines
# still don't have it.
socketpair(CHILD, PARENT, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
                            ||  die "socketpair: $!";

CHILD->autoflush(1);
PARENT->autoflush(1);

if ($pid = fork()) {
    close PARENT;
    print CHILD "Parent Pid $$ is sending this\n";
```

```
        chomp($line = <CHILD>);
        print "Parent Pid $$ just read this: '$line'\n";
        close CHILD;
        waitpid($pid, 0);
    } else {
        die "cannot fork: $!" unless defined $pid;
        close CHILD;
        chomp($line = <PARENT>);
        print "Child Pid $$ just read this: '$line'\n";
        print PARENT "Child Pid $$ is sending this\n";
        close PARENT;
        exit(0);
    }
```

36.6 Sockets: Client/Server Communication

While not entirely limited to Unix-derived operating systems (e.g., WinSock on PCs provides socket support, as do some VMS libraries), you might not have sockets on your system, in which case this section probably isn't going to do you much good. With sockets, you can do both virtual circuits like TCP streams and datagrams like UDP packets. You may be able to do even more depending on your system.

The Perl functions for dealing with sockets have the same names as the corresponding system calls in C, but their arguments tend to differ for two reasons. First, Perl filehandles work differently than C file descriptors. Second, Perl already knows the length of its strings, so you don't need to pass that information.

One of the major problems with ancient, antemillennial socket code in Perl was that it used hard-coded values for some of the constants, which severely hurt portability. If you ever see code that does anything like explicitly setting `$AF_INET = 2`, you know you're in for big trouble. An immeasurably superior approach is to use the `Socket` module, which more reliably grants access to the various constants and functions you'll need.

If you're not writing a server/client for an existing protocol like NNTP or SMTP, you should give some thought to how your server will know when the client has finished talking, and vice-versa. Most protocols are based on one-line messages and responses (so one party knows the other has finished when a `"\n"` is received) or multi-line messages and responses that end with a period on an empty line (`"\n.\n"` terminates a message/response).

36.6.1 Internet Line Terminators

The Internet line terminator is `"\015\012"`. Under ASCII variants of Unix, that could usually be written as `"\r\n"`, but under other systems, `"\r\n"` might at times be `"\015\015\012"`, `"\012\012\015"`, or something completely different. The standards specify writing `"\015\012"` to be conformant (be strict in what you provide), but they also recommend accepting a lone `"\012"` on input (be lenient in what you require). We haven't always been very good about that in the code in this manpage, but unless you're on a Mac from way back in its pre-Unix dark ages, you'll probably be ok.

36.6.2 Internet TCP Clients and Servers

Use Internet-domain sockets when you want to do client-server communication that might extend to machines outside of your own system.

Here's a sample TCP client using Internet-domain sockets:

```
#!/usr/bin/perl -w
use strict;
use Socket;
my ($remote, $port, $iaddr, $paddr, $proto, $line);

$remote  = shift || "localhost";
$port    = shift || 2345;  # random port
if ($port =~ /\D/) { $port = getservbyname($port, "tcp") }
die "No port" unless $port;
$iaddr   = inet_aton($remote)       || die "no host: $remote";
$paddr   = sockaddr_in($port, $iaddr);

$proto   = getprotobyname("tcp");
socket(SOCK, PF_INET, SOCK_STREAM, $proto)  || die "socket: $!";
connect(SOCK, $paddr)               || die "connect: $!";
while ($line = <SOCK>) {
    print $line;
}

close (SOCK)                        || die "close: $!";
exit(0);
```

And here's a corresponding server to go along with it. We'll leave the address as `INADDR_ANY` so that the kernel can choose the appropriate interface on multihomed hosts. If you want sit on a particular interface (like the external side of a gateway or firewall machine), fill this in with your real address instead.

```
#!/usr/bin/perl -Tw
use strict;
BEGIN { $ENV{PATH} = "/usr/bin:/bin" }
use Socket;
use Carp;
my $EOL = "\015\012";

sub logmsg { print "$0 $$: @_ at ", scalar localtime(), "\n" }

my $port  = shift || 2345;
die "invalid port" unless if $port =~ /^ \d+ $/x;

my $proto = getprotobyname("tcp");

socket(Server, PF_INET, SOCK_STREAM, $proto)    || die "socket: $!";
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR, pack("l", 1))
                                                || die "setsockopt: $!";
```

```
bind(Server, sockaddr_in($port, INADDR_ANY))    || die "bind: $!";
listen(Server, SOMAXCONN)                       || die "listen: $!";

logmsg "server started on port $port";

my $paddr;

$SIG{CHLD} = \&REAPER;

for ( ; $paddr = accept(Client, Server); close Client) {
    my($port, $iaddr) = sockaddr_in($paddr);
    my $name = gethostbyaddr($iaddr, AF_INET);

    logmsg "connection from $name [",
            inet_ntoa($iaddr), "]
            at port $port";

    print Client "Hello there, $name, it's now ",
                    scalar localtime(), $EOL;
}
```

And here's a multitasking version. It's multitasked in that like most typical servers, it spawns (fork()s) a slave server to handle the client request so that the master server can quickly go back to service a new client.

```
#!/usr/bin/perl -Tw
use strict;
BEGIN { $ENV{PATH} = "/usr/bin:/bin" }
use Socket;
use Carp;
my $EOL = "\015\012";

sub spawn;  # forward declaration
sub logmsg { print "$0 $$: @_ at ", scalar localtime(), "\n" }

my $port  = shift || 2345;
die "invalid port" unless $port =~ /^ \d+ $/x;

my $proto = getprotobyname("tcp");

socket(Server, PF_INET, SOCK_STREAM, $proto)    || die "socket: $!";
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR, pack("l", 1))
                                                || die "setsockopt: $!";
bind(Server, sockaddr_in($port, INADDR_ANY))    || die "bind: $!";
listen(Server, SOMAXCONN)                       || die "listen: $!";

logmsg "server started on port $port";
```

```
my $waitedpid = 0;
my $paddr;

use POSIX ":sys_wait_h";
use Errno;

sub REAPER {
    local $!;   # don’t let waitpid() overwrite current error
    while ((my $pid = waitpid(-1, WNOHANG)) > 0 && WIFEXITED($?)) {
        logmsg "reaped $waitedpid" . ($? ? " with exit $?" : "");
    }
    $SIG{CHLD} = \&REAPER;  # loathe SysV
}

$SIG{CHLD} = \&REAPER;

while (1) {
    $paddr = accept(Client, Server) || do {
        # try again if accept() returned because got a signal
        next if $!{EINTR};
        die "accept: $!";
    };
    my ($port, $iaddr) = sockaddr_in($paddr);
    my $name = gethostbyaddr($iaddr, AF_INET);

    logmsg "connection from $name [",
           inet_ntoa($iaddr),
           "] at port $port";

    spawn sub {
        $| = 1;
        print "Hello there, $name, it’s now ", scalar localtime(), $EOL;
        exec "/usr/games/fortune"       # XXX: "wrong" line terminators
            or confess "can’t exec fortune: $!";
    };
    close Client;
}

sub spawn {
    my $coderef = shift;

    unless (@_ == 0 && $coderef && ref($coderef) eq "CODE") {
        confess "usage: spawn CODEREF";
    }

    my $pid;
    unless (defined($pid = fork())) {
        logmsg "cannot fork: $!";
        return;
    }
    elsif ($pid) {
        logmsg "begat $pid";
        return; # I'm the parent
    }
    # else I'm the child -- go spawn

    open(STDIN,  "<&Client")    || die "can't dup client to stdin";
    open(STDOUT, ">&Client")    || die "can't dup client to stdout";
    ## open(STDERR, ">&STDOUT") || die "can't dup stdout to stderr";
    exit($coderef->());
}
```

This server takes the trouble to clone off a child version via fork() for each incoming request. That way it can handle many requests at once, which you might not always want. Even if you don't fork(), the listen() will allow that many pending connections. Forking servers have to be particularly careful about cleaning up their dead children (called "zombies" in Unix parlance), because otherwise you'll quickly fill up your process table. The REAPER subroutine is used here to call waitpid() for any child processes that have finished, thereby ensuring that they terminate cleanly and don't join the ranks of the living dead.

Within the while loop we call accept() and check to see if it returns a false value. This would normally indicate a system error needs to be reported. However, the introduction of safe signals (see Section 36.3.2 [Deferred Signals (Safe Signals)], page 671 above) in Perl 5.8.0 means that accept() might also be interrupted when the process receives a signal. This typically happens when one of the forked subprocesses exits and notifies the parent process with a CHLD signal.

If accept() is interrupted by a signal, $! will be set to EINTR. If this happens, we can safely continue to the next iteration of the loop and another call to accept(). It is important that your signal handling code not modify the value of $!, or else this test will likely fail. In the REAPER subroutine we create a local version of $! before calling waitpid(). When waitpid() sets $! to ECHILD as it inevitably does when it has no more children waiting, it updates the local copy and leaves the original unchanged.

You should use the **-T** flag to enable taint checking (see Section 70.1 [perlsec NAME], page 1198) even if we aren't running setuid or setgid. This is always a good idea for servers or any program run on behalf of someone else (like CGI scripts), because it lessens the chances that people from the outside will be able to compromise your system.

Let's look at another TCP client. This one connects to the TCP "time" service on a number of different machines and shows how far their clocks differ from the system on which it's being run:

```
    #!/usr/bin/perl  -w
    use strict;
    use Socket;
```

```
my $SECS_OF_70_YEARS = 2208988800;
sub ctime { scalar localtime(shift() || time()) }

my $iaddr = gethostbyname("localhost");
my $proto = getprotobyname("tcp");
my $port = getservbyname("time", "tcp");
my $paddr = sockaddr_in(0, $iaddr);
my($host);

$| = 1;
printf "%-24s %8s %s\n", "localhost", 0, ctime();

foreach $host (@ARGV) {
    printf "%-24s ", $host;
    my $hisiaddr = inet_aton($host)     || die "unknown host";
    my $hispaddr = sockaddr_in($port, $hisiaddr);
    socket(SOCKET, PF_INET, SOCK_STREAM, $proto)
                                        || die "socket: $!";
    connect(SOCKET, $hispaddr)          || die "connect: $!";
    my $rtime = pack("C4", ());
    read(SOCKET, $rtime, 4);
    close(SOCKET);
    my $histime = unpack("N", $rtime) - $SECS_OF_70_YEARS;
    printf "%8d %s\n", $histime - time(), ctime($histime);
}
```

36.6.3 Unix-Domain TCP Clients and Servers

That's fine for Internet-domain clients and servers, but what about local communications? While you can use the same setup, sometimes you don't want to. Unix-domain sockets are local to the current host, and are often used internally to implement pipes. Unlike Internet domain sockets, Unix domain sockets can show up in the file system with an ls(1) listing.

```
% ls -l /dev/log
srw-rw-rw-  1 root            0 Oct 31 07:23 /dev/log
```

You can test for these with Perl's **-S** file test:

```
unless (-S "/dev/log") {
    die "something's wicked with the log system";
}
```

Here's a sample Unix-domain client:

```
#!/usr/bin/perl -w
use Socket;
use strict;
my ($rendezvous, $line);

$rendezvous = shift || "catsock";
socket(SOCK, PF_UNIX, SOCK_STREAM, 0)     || die "socket: $!";
connect(SOCK, sockaddr_un($rendezvous))   || die "connect: $!";
```

```
    while (defined($line = <SOCK>)) {
        print $line;
    }
    exit(0);
```

And here's a corresponding server. You don't have to worry about silly network terminators here because Unix domain sockets are guaranteed to be on the localhost, and thus everything works right.

```
    #!/usr/bin/perl -Tw
    use strict;
    use Socket;
    use Carp;

    BEGIN { $ENV{PATH} = "/usr/bin:/bin" }
    sub spawn;  # forward declaration
    sub logmsg { print "$0 $$: @_ at ", scalar localtime(), "\n" }

    my $NAME = "catsock";
    my $uaddr = sockaddr_un($NAME);
    my $proto = getprotobyname("tcp");

    socket(Server, PF_UNIX, SOCK_STREAM, 0) || die "socket: $!";
    unlink($NAME);
    bind  (Server, $uaddr)                  || die "bind: $!";
    listen(Server, SOMAXCONN)               || die "listen: $!";

    logmsg "server started on $NAME";

    my $waitedpid;

    use POSIX ":sys_wait_h";
    sub REAPER {
        my $child;
        while (($waitedpid = waitpid(-1, WNOHANG)) > 0) {
            logmsg "reaped $waitedpid" . ($? ? " with exit $?" : "");
        }
        $SIG{CHLD} = \&REAPER;  # loathe SysV
    }

    $SIG{CHLD} = \&REAPER;


    for ( $waitedpid = 0;
          accept(Client, Server) || $waitedpid;
          $waitedpid = 0, close Client)
    {
        next if $waitedpid;
        logmsg "connection on $NAME";
        spawn sub {
            print "Hello there, it’s now ", scalar localtime(), "\n";
            exec("/usr/games/fortune")  || die "can’t exec fortune: $!";
        };
    }

    sub spawn {
        my $coderef = shift();

        unless (@_ == 0 && $coderef && ref($coderef) eq "CODE") {
            confess "usage: spawn CODEREF";
        }

        my $pid;
        unless (defined($pid = fork())) {
            logmsg "cannot fork: $!";
            return;
        }
        elsif ($pid) {
            logmsg "begat $pid";
            return; # I’m the parent
        }
        else {
            # I’m the child -- go spawn
        }

        open(STDIN,  "<&Client")    || die "can’t dup client to stdin";
        open(STDOUT, ">&Client")    || die "can’t dup client to stdout";
        ## open(STDERR, ">&STDOUT") || die "can’t dup stdout to stderr";
        exit($coderef->());
    }
```

As you see, it’s remarkably similar to the Internet domain TCP server, so much so, in fact, that we’ve omitted several duplicate functions–spawn(), logmsg(), ctime(), and REAPER()–which are the same as in the other server.

So why would you ever want to use a Unix domain socket instead of a simpler named pipe? Because a named pipe doesn’t give you sessions. You can’t tell one process’s data from another’s. With socket programming, you get a separate session for each client; that’s why accept() takes two arguments.

For example, let’s say that you have a long-running database server daemon that you want folks to be able to access from the Web, but only if they go through a CGI interface. You’d have a small, simple CGI program that does whatever checks and logging you feel like, and then acts as a Unix-domain client and connects to your private server.

36.7 TCP Clients with IO::Socket

For those preferring a higher-level interface to socket programming, the IO::Socket module provides an object-oriented approach. If for some reason you lack this module, you can just fetch IO::Socket from CPAN, where you'll also find modules providing easy interfaces to the following systems: DNS, FTP, Ident (RFC 931), NIS and NISPlus, NNTP, Ping, POP3, SMTP, SNMP, SSLeay, Telnet, and Time–to name just a few.

36.7.1 A Simple Client

Here's a client that creates a TCP connection to the "daytime" service at port 13 of the host name "localhost" and prints out everything that the server there cares to provide.

```
#!/usr/bin/perl -w
use IO::Socket;
$remote = IO::Socket::INET->new(
                    Proto    => "tcp",
                    PeerAddr => "localhost",
                    PeerPort => "daytime(13)",
                )
              || die "can't connect to daytime service on localhost";
while (<$remote>) { print }
```

When you run this program, you should get something back that looks like this:

```
Wed May 14 08:40:46 MDT 1997
```

Here are what those parameters to the new() constructor mean:

`Proto`

This is which protocol to use. In this case, the socket handle returned will be connected to a TCP socket, because we want a stream-oriented connection, that is, one that acts pretty much like a plain old file. Not all sockets are this of this type. For example, the UDP protocol can be used to make a datagram socket, used for message-passing.

`PeerAddr`

This is the name or Internet address of the remote host the server is running on. We could have specified a longer name like `"www.perl.com"`, or an address like `"207.171.7.72"`. For demonstration purposes, we've used the special hostname `"localhost"`, which should always mean the current machine you're running on. The corresponding Internet address for localhost is `"127.0.0.1"`, if you'd rather use that.

`PeerPort`

This is the service name or port number we'd like to connect to. We could have gotten away with using just `"daytime"` on systems with a well-configured system services file,[FOOTNOTE: The system services file is found in */etc/services* under Unixy systems.] but here we've specified the port number (13) in parentheses. Using just the number would have also worked, but numeric literals make careful programmers nervous.

Notice how the return value from the `new` constructor is used as a filehandle in the `while` loop? That's what's called an *indirect filehandle*, a scalar variable containing a filehandle. You can use it the same way you would a normal filehandle. For example, you can read one line from it this way:

```
$line = <$handle>;
```

all remaining lines from is this way:

```
@lines = <$handle>;
```

and send a line of data to it this way:

```
print $handle "some data\n";
```

36.7.2 A Webget Client

Here's a simple client that takes a remote host to fetch a document from, and then a list of files to get from that host. This is a more interesting client than the previous one because it first sends something to the server before fetching the server's response.

```
#!/usr/bin/perl -w
use IO::Socket;
unless (@ARGV > 1) { die "usage: $0 host url ..." }
$host = shift(@ARGV);
$EOL = "\015\012";
$BLANK = $EOL x 2;
for my $document (@ARGV) {
    $remote = IO::Socket::INET->new( Proto     => "tcp",
                                     PeerAddr  => $host,
                                     PeerPort  => "http(80)",
              )     || die "cannot connect to httpd on $host";
    $remote->autoflush(1);
    print $remote "GET $document HTTP/1.0" . $BLANK;
    while ( <$remote> ) { print }
    close $remote;
}
```

The web server handling the HTTP service is assumed to be at its standard port, number 80. If the server you're trying to connect to is at a different port, like 1080 or 8080, you should specify it as the named-parameter pair, `PeerPort => 8080`. The `autoflush` method is used on the socket because otherwise the system would buffer up the output we sent it. (If you're on a prehistoric Mac, you'll also need to change every `"\n"` in your code that sends data over the network to be a `"\015\012"` instead.)

Connecting to the server is only the first part of the process: once you have the connection, you have to use the server's language. Each server on the network has its own little command language that it expects as input. The string that we send to the server starting with "GET" is in HTTP syntax. In this case, we simply request each specified document. Yes, we really are making a new connection for each document, even though it's the same host. That's the way you always used to have to speak HTTP. Recent versions of web browsers may request that the remote server leave the connection open a little while, but the server doesn't have to honor such a request.

Here's an example of running that program, which we'll call *webget*:

```
% webget www.perl.com /guanaco.html
HTTP/1.1 404 File Not Found
Date: Thu, 08 May 1997 18:02:32 GMT
Server: Apache/1.2b6
Connection: close
Content-type: text/html

<HEAD><TITLE>404 File Not Found</TITLE></HEAD>
<BODY><H1>File Not Found</H1>
The requested URL /guanaco.html was not found on this server.<P>
</BODY>
```

Ok, so that's not very interesting, because it didn't find that particular document. But a long response wouldn't have fit on this page.

For a more featureful version of this program, you should look to the *lwp-request* program included with the LWP modules from CPAN.

36.7.3 Interactive Client with IO::Socket

Well, that's all fine if you want to send one command and get one answer, but what about setting up something fully interactive, somewhat like the way *telnet* works? That way you can type a line, get the answer, type a line, get the answer, etc.

This client is more complicated than the two we've done so far, but if you're on a system that supports the powerful `fork` call, the solution isn't that rough. Once you've made the connection to whatever service you'd like to chat with, call `fork` to clone your process. Each of these two identical process has a very simple job to do: the parent copies everything from the socket to standard output, while the child simultaneously copies everything from standard input to the socket. To accomplish the same thing using just one process would be *much* harder, because it's easier to code two processes to do one thing than it is to code one process to do two things. (This keep-it-simple principle a cornerstones of the Unix philosophy, and good software engineering as well, which is probably why it's spread to other systems.)

Here's the code:

```
#!/usr/bin/perl -w
use strict;
use IO::Socket;
my ($host, $port, $kidpid, $handle, $line);

unless (@ARGV == 2) { die "usage: $0 host port" }
($host, $port) = @ARGV;

# create a tcp connection to the specified host and port
$handle = IO::Socket::INET->new(Proto     => "tcp",
                                PeerAddr  => $host,
                                PeerPort  => $port)
           || die "can't connect to port $port on $host: $!";

$handle->autoflush(1);       # so output gets there right away
```

```
print STDERR "[Connected to $host:$port]\n";

# split the program into two processes, identical twins
die "can’t fork: $!" unless defined($kidpid = fork());

# the if{} block runs only in the parent process
if ($kidpid) {
    # copy the socket to standard output
    while (defined ($line = <$handle>)) {
        print STDOUT $line;
    }
    kill("TERM", $kidpid);   # send SIGTERM to child
}
# the else{} block runs only in the child process
else {
    # copy standard input to the socket
    while (defined ($line = <STDIN>)) {
        print $handle $line;
    }
    exit(0);                # just in case
}
```

The `kill` function in the parent’s `if` block is there to send a signal to our child process, currently running in the `else` block, as soon as the remote server has closed its end of the connection.

If the remote server sends data a byte at time, and you need that data immediately without waiting for a newline (which might not happen), you may wish to replace the `while` loop in the parent with the following:

```
my $byte;
while (sysread($handle, $byte, 1) == 1) {
    print STDOUT $byte;
}
```

Making a system call for each byte you want to read is not very efficient (to put it mildly) but is the simplest to explain and works reasonably well.

36.8 TCP Servers with IO::Socket

As always, setting up a server is little bit more involved than running a client. The model is that the server creates a special kind of socket that does nothing but listen on a particular port for incoming connections. It does this by calling the `IO::Socket::INET->new()` method with slightly different arguments than the client did.

Proto

This is which protocol to use. Like our clients, we’ll still specify `"tcp"` here.

LocalPort

We specify a local port in the `LocalPort` argument, which we didn’t do for the client. This is service name or port number for which you want to be the server.

(Under Unix, ports under 1024 are restricted to the superuser.) In our sample, we'll use port 9000, but you can use any port that's not currently in use on your system. If you try to use one already in used, you'll get an "Address already in use" message. Under Unix, the `netstat -a` command will show which services current have servers.

Listen

The `Listen` parameter is set to the maximum number of pending connections we can accept until we turn away incoming clients. Think of it as a call-waiting queue for your telephone. The low-level Socket module has a special symbol for the system maximum, which is SOMAXCONN.

Reuse

The `Reuse` parameter is needed so that we restart our server manually without waiting a few minutes to allow system buffers to clear out.

Once the generic server socket has been created using the parameters listed above, the server then waits for a new client to connect to it. The server blocks in the `accept` method, which eventually accepts a bidirectional connection from the remote client. (Make sure to autoflush this handle to circumvent buffering.)

To add to user-friendliness, our server prompts the user for commands. Most servers don't do this. Because of the prompt without a newline, you'll have to use the `sysread` variant of the interactive client above.

This server accepts one of five different commands, sending output back to the client. Unlike most network servers, this one handles only one incoming client at a time. Multitasking servers are covered in Chapter 16 of the Camel.

Here's the code. We'll

```
#!/usr/bin/perl -w
use IO::Socket;
use Net::hostent;      # for OOish version of gethostbyaddr

$PORT = 9000;          # pick something not in use

$server = IO::Socket::INET->new( Proto     => "tcp",
                                 LocalPort => $PORT,
                                 Listen    => SOMAXCONN,
                                 Reuse     => 1);

die "can't setup server" unless $server;
print "[Server $0 accepting clients]\n";

while ($client = $server->accept()) {
  $client->autoflush(1);
  print $client "Welcome to $0; type help for command list.\n";
  $hostinfo = gethostbyaddr($client->peeraddr);
  printf "[Connect from %s]\n", $hostinfo ? $hostinfo->name : $client->peerhost;
  print $client "Command? ";
  while ( <$client>) {
    next unless /\S/;       # blank line
    if    (/quit|exit/i)    { last                                     }
    elsif (/date|time/i)    { printf $client "%s\n", scalar localtime() }
    elsif (/who/i )         { print  $client `who 2>&1`                }
    elsif (/cookie/i )      { print  $client `/usr/games/fortune 2>&1`  }
    elsif (/motd/i )        { print  $client `cat /etc/motd 2>&1`       }
    else {
      print $client "Commands: quit date who cookie motd\n";
    }
  } continue {
     print $client "Command? ";
  }
  close $client;
}
```

36.9 UDP: Message Passing

Another kind of client-server setup is one that uses not connections, but messages. UDP communications involve much lower overhead but also provide less reliability, as there are no promises that messages will arrive at all, let alone in order and unmangled. Still, UDP offers some advantages over TCP, including being able to "broadcast" or "multicast" to a whole bunch of destination hosts at once (usually on your local subnet). If you find yourself overly concerned about reliability and start building checks into your message system, then you probably should use just TCP to start with.

UDP datagrams are *not* a bytestream and should not be treated as such. This makes using I/O mechanisms with internal buffering like stdio (i.e. print() and friends) especially cumbersome. Use syswrite(), or better send(), like in the example below.

Here's a UDP program similar to the sample Internet TCP client given earlier. However, instead of checking one host at a time, the UDP version will check many of them asynchronously by simulating a multicast and then using select() to do a timed-out wait for I/O. To do something similar with TCP, you'd have to use a different socket handle for each host.

```
#!/usr/bin/perl -w
use strict;
use Socket;
use Sys::Hostname;

my ( $count, $hisiaddr, $hispaddr, $histime,
     $host, $iaddr, $paddr, $port, $proto,
     $rin, $rout, $rtime, $SECS_OF_70_YEARS);

$SECS_OF_70_YEARS = 2_208_988_800;

$iaddr = gethostbyname(hostname());
$proto = getprotobyname("udp");
$port = getservbyname("time", "udp");
```

```
    $paddr = sockaddr_in(0, $iaddr); # 0 means let kernel pick

    socket(SOCKET, PF_INET, SOCK_DGRAM, $proto)   || die "socket: $!";
    bind(SOCKET, $paddr)                          || die "bind: $!";

    $| = 1;
    printf "%-12s %8s %s\n",  "localhost", 0, scalar localtime();
    $count = 0;
    for $host (@ARGV) {
        $count++;
        $hisiaddr = inet_aton($host)              || die "unknown host";
        $hispaddr = sockaddr_in($port, $hisiaddr);
        defined(send(SOCKET, 0, 0, $hispaddr))    || die "send $host: $!";
    }

    $rin = "";
    vec($rin, fileno(SOCKET), 1) = 1;

    # timeout after 10.0 seconds
    while ($count && select($rout = $rin, undef, undef, 10.0)) {
        $rtime = "";
        $hispaddr = recv(SOCKET, $rtime, 4, 0)    || die "recv: $!";
        ($port, $hisiaddr) = sockaddr_in($hispaddr);
        $host = gethostbyaddr($hisiaddr, AF_INET);
        $histime = unpack("N", $rtime) - $SECS_OF_70_YEARS;
        printf "%-12s ", $host;
        printf "%8d %s\n", $histime - time(), scalar localtime($histime);
        $count--;
    }
```

This example does not include any retries and may consequently fail to contact a reachable host. The most prominent reason for this is congestion of the queues on the sending host if the number of hosts to contact is sufficiently large.

36.10 SysV IPC

While System V IPC isn't so widely used as sockets, it still has some interesting uses. However, you cannot use SysV IPC or Berkeley mmap() to have a variable shared amongst several processes. That's because Perl would reallocate your string when you weren't wanting it to. You might look into the `IPC::Shareable` or `threads::shared` modules for that.

Here's a small example showing shared memory usage.

```
    use IPC::SysV qw(IPC_PRIVATE IPC_RMID S_IRUSR S_IWUSR);

    $size = 2000;
    $id = shmget(IPC_PRIVATE, $size, S_IRUSR | S_IWUSR);
    defined($id)                    || die "shmget: $!";
    print "shm key $id\n";
```

```
$message = "Message #1";
shmwrite($id, $message, 0, 60)  || die "shmwrite: $!";
print "wrote: '$message'\n";
shmread($id, $buff, 0, 60)      || die "shmread: $!";
print "read : '$buff'\n";

# the buffer of shmread is zero-character end-padded.
substr($buff, index($buff, "\0")) = "";
print "un" unless $buff eq $message;
print "swell\n";

print "deleting shm $id\n";
shmctl($id, IPC_RMID, 0)        || die "shmctl: $!";
```

Here's an example of a semaphore:

```
use IPC::SysV qw(IPC_CREAT);

$IPC_KEY = 1234;
$id = semget($IPC_KEY, 10, 0666 | IPC_CREAT);
defined($id)                    || die "semget: $!";
print "sem id $id\n";
```

Put this code in a separate file to be run in more than one process. Call the file `take`:

```
# create a semaphore

$IPC_KEY = 1234;
$id = semget($IPC_KEY, 0, 0);
defined($id)                    || die "semget: $!";

$semnum  = 0;
$semflag = 0;

# "take" semaphore
# wait for semaphore to be zero
$semop = 0;
$opstring1 = pack("s!s!s!", $semnum, $semop, $semflag);

# Increment the semaphore count
$semop = 1;
$opstring2 = pack("s!s!s!", $semnum, $semop,  $semflag);
$opstring  = $opstring1 . $opstring2;

semop($id, $opstring)   || die "semop: $!";
```

Put this code in a separate file to be run in more than one process. Call this file `give`:

```
# "give" the semaphore
# run this in the original process and you will see
# that the second process continues
```

```
$IPC_KEY = 1234;
$id = semget($IPC_KEY, 0, 0);
die unless defined($id);

$semnum  = 0;
$semflag = 0;

# Decrement the semaphore count
$semop = -1;
$opstring = pack("s!s!s!", $semnum, $semop, $semflag);

semop($id, $opstring)   || die "semop: $!";
```

The SysV IPC code above was written long ago, and it's definitely clunky looking. For a more modern look, see the IPC::SysV module.

A small example demonstrating SysV message queues:

```
use IPC::SysV qw(IPC_PRIVATE IPC_RMID IPC_CREAT S_IRUSR S_IWUSR);

my $id = msgget(IPC_PRIVATE, IPC_CREAT | S_IRUSR | S_IWUSR);
defined($id)                || die "msgget failed: $!";

my $sent      = "message";
my $type_sent = 1234;

msgsnd($id, pack("l! a*", $type_sent, $sent), 0)
                            || die "msgsnd failed: $!";

msgrcv($id, my $rcvd_buf, 60, 0, 0)
                            || die "msgrcv failed: $!";

my($type_rcvd, $rcvd) = unpack("l! a*", $rcvd_buf);

if ($rcvd eq $sent) {
    print "okay\n";
} else {
    print "not okay\n";
}

msgctl($id, IPC_RMID, 0)    || die "msgctl failed: $!\n";
```

36.11 NOTES

Most of these routines quietly but politely return `undef` when they fail instead of causing your program to die right then and there due to an uncaught exception. (Actually, some of the new *Socket* conversion functions do croak() on bad arguments.) It is therefore essential to check return values from these functions. Always begin your socket programs this way for optimal success, and don't forget to add the **-T** taint-checking flag to the `#!` line for servers:

```
#!/usr/bin/perl -Tw
use strict;
use sigtrap;
use Socket;
```

36.12 BUGS

These routines all create system-specific portability problems. As noted elsewhere, Perl is at the mercy of your C libraries for much of its system behavior. It's probably safest to assume broken SysV semantics for signals and to stick with simple TCP and UDP socket operations; e.g., don't try to pass open file descriptors over a local UDP datagram socket if you want your code to stand a chance of being portable.

36.13 AUTHOR

Tom Christiansen, with occasional vestiges of Larry Wall's original version and suggestions from the Perl Porters.

36.14 SEE ALSO

There's a lot more to networking than this, but this should get you started.

For intrepid programmers, the indispensable textbook is *Unix Network Programming, 2nd Edition, Volume 1* by W. Richard Stevens (published by Prentice-Hall). Most books on networking address the subject from the perspective of a C programmer; translation to Perl is left as an exercise for the reader.

The IO::Socket(3) manpage describes the object library, and the Socket(3) manpage describes the low-level interface to sockets. Besides the obvious functions in Section 25.1 [perlfunc NAME], page 351, you should also check out the `modules` file at your nearest CPAN site, especially `http://www.cpan.org/modules/00modlist.long.html#ID5_Networking_`. See `perlmodlib` or best yet, the `Perl FAQ` for a description of what CPAN is and where to get it if the previous link doesn't work for you.

Section 5 of CPAN's `modules` file is devoted to "Networking, Device Control (modems), and Interprocess Communication", and contains numerous unbundled modules numerous networking modules, Chat and Expect operations, CGI programming, DCE, FTP, IPC, NNTP, Proxy, Ptty, RPC, SNMP, SMTP, Telnet, Threads, and ToolTalk–to name just a few.

37 perllexwarn

37.1 NAME

perllexwarn - Perl Lexical Warnings

37.2 DESCRIPTION

Perl v5.6.0 introduced lexical control over the handling of warnings by category. The `warnings` pragma generally replaces the command line flag **`-w`**. Documentation on the use of lexical warnings, once partly found in this document, is now found in the `warnings` documentation.

38 perllocale

38.1 NAME

perllocale - Perl locale handling (internationalization and localization)

38.2 DESCRIPTION

In the beginning there was ASCII, the "American Standard Code for Information Interchange", which works quite well for Americans with their English alphabet and dollar-denominated currency. But it doesn't work so well even for other English speakers, who may use different currencies, such as the pound sterling (as the symbol for that currency is not in ASCII); and it's hopelessly inadequate for many of the thousands of the world's other languages.

To address these deficiencies, the concept of locales was invented (formally the ISO C, XPG4, POSIX 1.c "locale system"). And applications were and are being written that use the locale mechanism. The process of making such an application take account of its users' preferences in these kinds of matters is called **internationalization** (often abbreviated as **i18n**); telling such an application about a particular set of preferences is known as **localization** (**l10n**).

Perl has been extended to support the locale system. This is controlled per application by using one pragma, one function call, and several environment variables.

Unfortunately, there are quite a few deficiencies with the design (and often, the implementations) of locales. Unicode was invented (see Section 84.1 [perlunitut NAME], page 1367 for an introduction to that) in part to address these design deficiencies, and nowadays, there is a series of "UTF-8 locales", based on Unicode. These are locales whose character set is Unicode, encoded in UTF-8. Starting in v5.20, Perl fully supports UTF-8 locales, except for sorting and string comparisons. (Use `Unicode-Collate` for these.) Perl continues to support the old non UTF-8 locales as well. There are currently no UTF-8 locales for EBCDIC platforms.

(Unicode is also creating `CLDR`, the "Common Locale Data Repository", `http://cldr.unicode.org/` which includes more types of information than are available in the POSIX locale system. At the time of this writing, there was no CPAN module that provides access to this XML-encoded data. However, many of its locales have the POSIX-only data extracted, and are available as UTF-8 locales at `http://unicode.org/Public/cldr/latest/`.)

38.3 WHAT IS A LOCALE

A locale is a set of data that describes various aspects of how various communities in the world categorize their world. These categories are broken down into the following types (some of which include a brief note here):

Category `LC_NUMERIC`: Numeric formatting

This indicates how numbers should be formatted for human readability, for example the character used as the decimal point.

Category `LC_MONETARY`: Formatting of monetary amounts

Category `LC_TIME`: Date/Time formatting

Category `LC_MESSAGES`: Error and other messages

This is used by Perl itself only for accessing operating system error messages via [$!], page 1400 and [$^E], page 1399.

Category `LC_COLLATE`: Collation

This indicates the ordering of letters for comparison and sorting. In Latin alphabets, for example, `"b"`, generally follows `"a"`.

Category `LC_CTYPE`: Character Types

This indicates, for example if a character is an uppercase letter.

Other categories

Some platforms have other categories, dealing with such things as measurement units and paper sizes. None of these are used directly by Perl, but outside operations that Perl interacts with may use these. See [Not within the scope of "use locale"], page 703 below.

More details on the categories used by Perl are given below in Section 38.6 [LOCALE CATEGORIES], page 711.

Together, these categories go a long way towards being able to customize a single program to run in many different locations. But there are deficiencies, so keep reading.

38.4 PREPARING TO USE LOCALES

Perl itself (outside the `POSIX` module) will not use locales unless specifically requested to (but again note that Perl may interact with code that does use them). Even if there is such a request, **all** of the following must be true for it to work properly:

- **Your operating system must support the locale system**. If it does, you should find that the `setlocale()` function is a documented part of its C library.
- **Definitions for locales that you use must be installed**. You, or your system administrator, must make sure that this is the case. The available locales, the location in which they are kept, and the manner in which they are installed all vary from system to system. Some systems provide only a few, hard-wired locales and do not allow more to be added. Others allow you to add "canned" locales provided by the system supplier. Still others allow you or the system administrator to define and add arbitrary locales. (You may have to ask your supplier to provide canned locales that are not delivered with your operating system.) Read your system documentation for further illumination.
- **Perl must believe that the locale system is supported**. If it does, `perl -V:d_setlocale` will say that the value for `d_setlocale` is `define`.

If you want a Perl application to process and present your data according to a particular locale, the application code should include the `use locale` pragma (see Section 38.5.1 [The "use locale" pragma], page 703) where appropriate, and **at least one** of the following must be true:

1. **The locale-determining environment variables (see Section 38.8 [ENVIRONMENT], page 718) must be correctly set up** at the time the application is started, either by yourself or by whomever set up your system account; or
2. **The application must set its own locale** using the method described in Section 38.5.2 [The setlocale function], page 705.

38.5 USING LOCALES

38.5.1 The "`use locale`" pragma

By default, Perl itself (outside the `POSIX` module) ignores the current locale. The `use locale` pragma tells Perl to use the current locale for some operations. Starting in v5.16, there are optional parameters to this pragma, described below, which restrict which operations are affected by it.

The current locale is set at execution time by Section 38.5.2 [setlocale()], page 705 described below. If that function hasn't yet been called in the course of the program's execution, the current locale is that which was determined by the Section 38.8 [ENVIRONMENT], page 718 in effect at the start of the program. If there is no valid environment, the current locale is whatever the system default has been set to. On POSIX systems, it is likely, but not necessarily, the "C" locale. On Windows, the default is set via the computer's `Control Panel->Regional and Language Options` (or its current equivalent).

The operations that are affected by locale are:

Not within the scope of "`use locale`"

Only certain operations originating outside Perl should be affected, as follows:

- The current locale is used when going outside of Perl with operations like [system()], page 471 or [qx//], page 832, if those operations are locale-sensitive.
- Also Perl gives access to various C library functions through the `POSIX` module. Some of those functions are always affected by the current locale. For example, `POSIX::strftime()` uses `LC_TIME`; `POSIX::strtod()` uses `LC_NUMERIC`; `POSIX::strcoll()` and `POSIX::strxfrm()` use `LC_COLLATE`; and character classification functions like `POSIX::isalnum()` use `LC_CTYPE`. All such functions will behave according to the current underlying locale, even if that locale isn't exposed to Perl space.
- XS modules for all categories but `LC_NUMERIC` get the underlying locale, and hence any C library functions they call will use that underlying locale. For more discussion, see Section "CAVEATS" in `perlxs`.

Note that all C programs (including the perl interpreter, which is written in C) always have an underlying locale. That locale is the "C" locale unless changed by a call to Section 38.5.2 [setlocale()], page 705. When Perl starts up, it changes the underlying locale to the one which is indicated by the Section 38.8 [ENVIRONMENT], page 718. When using the `POSIX` module or writing XS code, it is important to keep in mind that the underlying locale may be something other than "C", even if the program hasn't explicitly changed it.

Lingering effects of `use locale`

Certain Perl operations that are set-up within the scope of a `use locale` retain that effect even outside the scope. These include:

- The output format of a [write()], page 489 is determined by an earlier format declaration ([perlfunc format], page 387), so whether or not the output is affected by locale is determined by if the `format()` is within the scope of a `use locale`, not whether the `write()` is.
- Regular expression patterns can be compiled using [qr//], page 823 with actual matching deferred to later. Again, it is whether or not the compilation was done within the scope of `use locale` that determines the match behavior, not if the matches are done within such a scope or not.

Under `"use locale";`

- All the above operations
- **Format declarations** ([perlfunc format], page 387) and hence any subsequent `write()`s use `LC_NUMERIC`.
- **stringification and output** use `LC_NUMERIC`. These include the results of `print()`, `printf()`, `say()`, and `sprintf()`.
- **The comparison operators** (`lt`, `le`, `cmp`, `ge`, and `gt`) use `LC_COLLATE`. `sort()` is also affected if used without an explicit comparison function, because it uses `cmp` by default.

  Note: `eq` and `ne` are unaffected by locale: they always perform a char-by-char comparison of their scalar operands. What's more, if `cmp` finds that its operands are equal according to the collation sequence specified by the current locale, it goes on to perform a char-by-char comparison, and only returns *0* (equal) if the operands are char-for-char identical. If you really want to know whether two strings–which `eq` and `cmp` may consider different–are equal as far as collation in the locale is concerned, see the discussion in [Category LC_COLLATE: Collation], page 702.
- **Regular expressions and case-modification functions** (`uc()`, `lc()`, `ucfirst()`, and `lcfirst()`) use `LC_CTYPE`
- **The variables [$!], page 1400** (and its synonyms `$ERRNO` and `$OS_ERROR`) **and [$^E], page 1399** (and its synonym `$EXTENDED_OS_ERROR`) when used as strings use `LC_MESSAGES`.

The default behavior is restored with the `no locale` pragma, or upon reaching the end of the block enclosing `use locale`. Note that `use locale` calls may be nested, and that what is in effect within an inner scope will revert to the outer scope's rules at the end of the inner scope.

The string result of any operation that uses locale information is tainted, as it is possible for a locale to be untrustworthy. See Section 38.7 [SECURITY], page 716.

Starting in Perl v5.16 in a very limited way, and more generally in v5.22, you can restrict which category or categories are enabled by this particular instance of the pragma by adding parameters to it. For example,

```
use locale qw(:ctype :numeric);
```

enables locale awareness within its scope of only those operations (listed above) that are affected by LC_CTYPE and LC_NUMERIC.

The possible categories are: :collate, :ctype, :messages, :monetary, :numeric, :time, and the pseudo category :characters (described below).

Thus you can say

```
use locale ':messages';
```

and only [$!], page 1400 and [$^E], page 1399 will be locale aware. Everything else is unaffected.

Since Perl doesn't currently do anything with the LC_MONETARY category, specifying :monetary does effectively nothing. Some systems have other categories, such as LC_PAPER_SIZE, but Perl also doesn't know anything about them, and there is no way to specify them in this pragma's arguments.

You can also easily say to use all categories but one, by either, for example,

```
use locale ':!ctype';
use locale ':not_ctype';
```

both of which mean to enable locale awarness of all categories but LC_CTYPE. Only one category argument may be specified in a use locale if it is of the negated form.

Prior to v5.22 only one form of the pragma with arguments is available:

```
use locale ':not_characters';
```

(and you have to say not_; you can't use the bang ! form). This pseudo category is a shorthand for specifying both :collate and :ctype. Hence, in the negated form, it is nearly the same thing as saying

```
use locale qw(:messages :monetary :numeric :time);
```

We use the term "nearly", because :not_characters also turns on use feature 'unicode_strings' within its scope. This form is less useful in v5.20 and later, and is described fully in Section 38.10 [Unicode and UTF-8], page 721, but briefly, it tells Perl to not use the character portions of the locale definition, that is the LC_CTYPE and LC_COLLATE categories. Instead it will use the native character set (extended by Unicode). When using this parameter, you are responsible for getting the external character set translated into the native/Unicode one (which it already will be if it is one of the increasingly popular UTF-8 locales). There are convenient ways of doing this, as described in Section 38.10 [Unicode and UTF-8], page 721.

38.5.2 The setlocale function

You can switch locales as often as you wish at run time with the POSIX::setlocale() function:

```
        # Import locale-handling tool set from POSIX module.
        # This example uses: setlocale -- the function call
        #                    LC_CTYPE -- explained below
        # (Showing the testing for success/failure of operations is
        # omitted in these examples to avoid distracting from the main
        # point)
        use POSIX qw(locale_h);
        use locale;
        my $old_locale;

        # query and save the old locale
        $old_locale = setlocale(LC_CTYPE);

        setlocale(LC_CTYPE, "fr_CA.ISO8859-1");
        # LC_CTYPE now in locale "French, Canada, codeset ISO 8859-1"

        setlocale(LC_CTYPE, "");
        # LC_CTYPE now reset to the default defined by the
        # LC_ALL/LC_CTYPE/LANG environment variables, or to the system
        # default.  See below for documentation.

        # restore the old locale
        setlocale(LC_CTYPE, $old_locale);
```

This simultaneously affects all threads of the program, so it may be problematic to use locales in threaded applications except where there is a single locale applicable to all threads.

The first argument of `setlocale()` gives the **category**, the second the **locale**. The category tells in what aspect of data processing you want to apply locale-specific rules. Category names are discussed in Section 38.6 [LOCALE CATEGORIES], page 711 and Section 38.8 [ENVIRONMENT], page 718. The locale is the name of a collection of customization information corresponding to a particular combination of language, country or territory, and codeset. Read on for hints on the naming of locales: not all systems name locales as in the example.

If no second argument is provided and the category is something other than `LC_ALL`, the function returns a string naming the current locale for the category. You can use this value as the second argument in a subsequent call to `setlocale()`, **but** on some platforms the string is opaque, not something that most people would be able to decipher as to what locale it means.

If no second argument is provided and the category is `LC_ALL`, the result is implementation-dependent. It may be a string of concatenated locale names (separator also implementation-dependent) or a single locale name. Please consult your setlocale(3) man page for details.

If a second argument is given and it corresponds to a valid locale, the locale for the category is set to that value, and the function returns the now-current locale value. You can then use this in yet another call to `setlocale()`. (In some implementations, the return value may sometimes differ from the value you gave as the second argument–think of it as an alias for the value you gave.)

As the example shows, if the second argument is an empty string, the category's locale is returned to the default specified by the corresponding environment variables. Generally, this results in a return to the default that was in force when Perl started up: changes to the environment made by the application after startup may or may not be noticed, depending on your system's C library.

Note that when a form of `use locale` that doesn't include all categories is specified, Perl ignores the excluded categories.

If `set_locale()` fails for some reason (for example, an attempt to set to a locale unknown to the system), the locale for the category is not changed, and the function returns `undef`.

For further information about the categories, consult setlocale(3).

38.5.3 Finding locales

For locales available in your system, consult also setlocale(3) to see whether it leads to the list of available locales (search for the *SEE ALSO* section). If that fails, try the following command lines:

```
locale -a

nlsinfo

ls /usr/lib/nls/loc

ls /usr/lib/locale

ls /usr/lib/nls

ls /usr/share/locale
```

and see whether they list something resembling these

```
en_US.ISO8859-1     de_DE.ISO8859-1     ru_RU.ISO8859-5
en_US.iso88591      de_DE.iso88591      ru_RU.iso88595
en_US               de_DE               ru_RU
en                  de                  ru
english             german              russian
english.iso88591    german.iso88591     russian.iso88595
english.roman8                          russian.koi8r
```

Sadly, even though the calling interface for `setlocale()` has been standardized, names of locales and the directories where the configuration resides have not been. The basic form of the name is *language_territory***.***codeset*, but the latter parts after *language* are not always present. The *language* and *country* are usually from the standards **ISO 3166** and **ISO 639**, the two-letter abbreviations for the countries and the languages of the world, respectively. The *codeset* part often mentions some **ISO 8859** character set, the Latin codesets. For example, `ISO 8859-1` is the so-called "Western European codeset" that can be used to encode most Western European languages adequately. Again, there are several ways to write even the name of that one standard. Lamentably.

Two special locales are worth particular mention: `"C"` and `"POSIX"`. Currently these are effectively the same locale: the difference is mainly that the first one is defined by the C standard, the second by the POSIX standard. They define the **default locale** in which every program starts in the absence of locale information in its environment. (The *default* default locale, if you will.) Its language is (American) English and its character codeset ASCII or, rarely, a superset thereof (such as the "DEC Multinational Character Set (DEC-MCS)"). **Warning**. The C locale delivered by some vendors may not actually exactly match what the C standard calls for. So beware.

NOTE: Not all systems have the "POSIX" locale (not all systems are POSIX-conformant), so use "C" when you need explicitly to specify this default locale.

38.5.4 LOCALE PROBLEMS

You may encounter the following warning message at Perl startup:

```
        perl: warning: Setting locale failed.
        perl: warning: Please check that your locale settings:
                LC_ALL = "En_US",
                LANG = (unset)
            are supported and installed on your system.
        perl: warning: Falling back to the standard locale ("C").
```

This means that your locale settings had `LC_ALL` set to "En_US" and LANG exists but has no value. Perl tried to believe you but could not. Instead, Perl gave up and fell back to the "C" locale, the default locale that is supposed to work no matter what. (On Windows, it first tries falling back to the system default locale.) This usually means your locale settings were wrong, they mention locales your system has never heard of, or the locale installation in your system has problems (for example, some system files are broken or missing). There are quick and temporary fixes to these problems, as well as more thorough and lasting fixes.

38.5.5 Testing for broken locales

If you are building Perl from source, the Perl test suite file `lib/locale.t` can be used to test the locales on your system. Setting the environment variable `PERL_DEBUG_FULL_TEST` to 1 will cause it to output detailed results. For example, on Linux, you could say

```
 PERL_DEBUG_FULL_TEST=1 ./perl -T -Ilib lib/locale.t > locale.log 2>&1
```

Besides many other tests, it will test every locale it finds on your system to see if they conform to the POSIX standard. If any have errors, it will include a summary near the end of the output of which locales passed all its tests, and which failed, and why.

38.5.6 Temporarily fixing locale problems

The two quickest fixes are either to render Perl silent about any locale inconsistencies or to run Perl under the default locale "C".

Perl's moaning about locale problems can be silenced by setting the environment variable `PERL_BADLANG` to "0" or "". This method really just sweeps the problem under the carpet: you tell Perl to shut up even when Perl sees that something is wrong. Do not be surprised if later something locale-dependent misbehaves.

Perl can be run under the "C" locale by setting the environment variable `LC_ALL` to "C". This method is perhaps a bit more civilized than the `PERL_BADLANG` approach, but setting `LC_ALL` (or other locale variables) may affect other programs as well, not just Perl. In particular, external programs run from within Perl will see these changes. If you make the new settings permanent (read on), all programs you run see the changes. See Section 38.8 [ENVIRONMENT], page 718 for the full list of relevant environment variables and Section 38.5 [USING LOCALES], page 703 for their effects in Perl. Effects in other programs are easily deducible. For example, the variable `LC_COLLATE` may well affect your **sort** program (or whatever the program that arranges "records" alphabetically in your system is called).

You can test out changing these variables temporarily, and if the new settings seem to help, put those settings into your shell startup files. Consult your local documentation for the exact details. For Bourne-like shells (**sh**, **ksh**, **bash**, **zsh**):

```
LC_ALL=en_US.ISO8859-1
export LC_ALL
```

This assumes that we saw the locale "en_US.ISO8859-1" using the commands discussed above. We decided to try that instead of the above faulty locale "En_US"–and in Cshish shells (**csh**, **tcsh**)

```
setenv LC_ALL en_US.ISO8859-1
```

or if you have the "env" application you can do (in any shell)

```
env LC_ALL=en_US.ISO8859-1 perl ...
```

If you do not know what shell you have, consult your local helpdesk or the equivalent.

38.5.7 Permanently fixing locale problems

The slower but superior fixes are when you may be able to yourself fix the misconfiguration of your own environment variables. The mis(sing)configuration of the whole system's locales usually requires the help of your friendly system administrator.

First, see earlier in this document about Section 38.5.3 [Finding locales], page 707. That tells how to find which locales are really supported–and more importantly, installed–on your system. In our example error message, environment variables affecting the locale are listed in the order of decreasing importance (and unset variables do not matter). Therefore, having LC_ALL set to "En_US" must have been the bad choice, as shown by the error message. First try fixing locale settings listed first.

Second, if using the listed commands you see something **exactly** (prefix matches do not count and case usually counts) like "En_US" without the quotes, then you should be okay because you are using a locale name that should be installed and available in your system. In this case, see Section 38.5.8 [Permanently fixing your system's locale configuration], page 709.

38.5.8 Permanently fixing your system's locale configuration

This is when you see something like:

```
perl: warning: Please check that your locale settings:
        LC_ALL = "En_US",
        LANG = (unset)
    are supported and installed on your system.
```

but then cannot see that "En_US" listed by the above-mentioned commands. You may see things like "en_US.ISO8859-1", but that isn't the same. In this case, try running under a locale that you can list and which somehow matches what you tried. The rules for matching locale names are a bit vague because standardization is weak in this area. See again the Section 38.5.3 [Finding locales], page 707 about general rules.

38.5.9 Fixing system locale configuration

Contact a system administrator (preferably your own) and report the exact error message you get, and ask them to read this same documentation you are now reading. They should be able to check whether there is something wrong with the locale configuration of the system. The Section 38.5.3 [Finding locales], page 707 section is unfortunately a bit vague about the exact commands and places because these things are not that standardized.

38.5.10 The localeconv function

The `POSIX::localeconv()` function allows you to get particulars of the locale-dependent numeric formatting information specified by the current underlying `LC_NUMERIC` and `LC_MONETARY` locales (regardless of whether called from within the scope of `use locale` or not). (If you just want the name of the current locale for a particular category, use `POSIX::setlocale()` with a single parameter–see Section 38.5.2 [The setlocale function], page 705.)

```
        use POSIX qw(locale_h);

        # Get a reference to a hash of locale-dependent info
        $locale_values = localeconv();

        # Output sorted list of the values
        for (sort keys %$locale_values) {
            printf "%-20s = %s\n", $_, $locale_values->{$_}
        }
```

`localeconv()` takes no arguments, and returns **a reference to** a hash. The keys of this hash are variable names for formatting, such as `decimal_point` and `thousands_sep`. The values are the corresponding, er, values. See Section "localeconv" in `POSIX` for a longer example listing the categories an implementation might be expected to provide; some provide more and others fewer. You don't need an explicit `use locale`, because `localeconv()` always observes the current locale.

Here's a simple-minded example program that rewrites its command-line parameters as integers correctly formatted in the current locale:

```
    use POSIX qw(locale_h);

    # Get some of locale's numeric formatting parameters
    my ($thousands_sep, $grouping) =
            @{localeconv()}{'thousands_sep', 'grouping'};

    # Apply defaults if values are missing
    $thousands_sep = ',' unless $thousands_sep;

    # grouping and mon_grouping are packed lists
    # of small integers (characters) telling the
    # grouping (thousand_seps and mon_thousand_seps
    # being the group dividers) of numbers and
    # monetary quantities.  The integers' meanings:
```

```
# 255 means no more grouping, 0 means repeat
# the previous grouping, 1-254 means use that
# as the current grouping.  Grouping goes from
# right to left (low to high digits).  In the
# below we cheat slightly by never using anything
# else than the first grouping (whatever that is).
if ($grouping) {
    @grouping = unpack("C*", $grouping);
} else {
    @grouping = (3);
}

# Format command line params for current locale
for (@ARGV) {
    $_ = int;    # Chop non-integer part
    1 while
    s/(\d)(\d{$grouping[0]}($|$thousands_sep))/$1$thousands_sep$2/;
    print "$_";
}
print "\n";
```

Note that if the platform doesn't have `LC_NUMERIC` and/or `LC_MONETARY` available or enabled, the corresponding elements of the hash will be missing.

38.5.11 I18N::Langinfo

Another interface for querying locale-dependent information is the `I18N::Langinfo::langinfo()` function, available at least in Unix-like systems and VMS.

The following example will import the `langinfo()` function itself and three constants to be used as arguments to `langinfo()`: a constant for the abbreviated first day of the week (the numbering starts from Sunday = 1) and two more constants for the affirmative and negative answers for a yes/no question in the current locale.

```
use I18N::Langinfo qw(langinfo ABDAY_1 YESSTR NOSTR);

my ($abday_1, $yesstr, $nostr)
            = map { langinfo } qw(ABDAY_1 YESSTR NOSTR);

print "$abday_1? [$yesstr/$nostr] ";
```

In other words, in the "C" (or English) locale the above will probably print something like:

```
Sun? [yes/no]
```

See `I18N-Langinfo` for more information.

38.6 LOCALE CATEGORIES

The following subsections describe basic locale categories. Beyond these, some combination categories allow manipulation of more than one basic category at a time. See Section 38.8 [ENVIRONMENT], page 718 for a discussion of these.

38.6.1 Category LC_COLLATE: Collation

In the scope of a `use locale` form that includes collation, Perl looks to the `LC_COLLATE` environment variable to determine the application's notions on collation (ordering) of characters. For example, "b" follows "a" in Latin alphabets, but where do "á" and "å" belong? And while "color" follows "chocolate" in English, what about in traditional Spanish?

The following collations all make sense and you may meet any of them if you "`use locale`".

```
A B C D E a b c d e
A a B b C c D d E e
a A b B c C d D e E
a b c d e A B C D E
```

Here is a code snippet to tell what "word" characters are in the current locale, in that locale's order:

```
use locale;
print +(sort grep /\w/, map { chr } 0..255), "\n";
```

Compare this with the characters that you see and their order if you state explicitly that the locale should be ignored:

```
no locale;
print +(sort grep /\w/, map { chr } 0..255), "\n";
```

This machine-native collation (which is what you get unless `use locale` has appeared earlier in the same block) must be used for sorting raw binary data, whereas the locale-dependent collation of the first example is useful for natural text.

As noted in Section 38.5 [USING LOCALES], page 703, `cmp` compares according to the current collation locale when `use locale` is in effect, but falls back to a char-by-char comparison for strings that the locale says are equal. You can use `POSIX::strcoll()` if you don't want this fall-back:

```
use POSIX qw(strcoll);
$equal_in_locale =
    !strcoll("space and case ignored", "SpaceAndCaseIgnored");
```

`$equal_in_locale` will be true if the collation locale specifies a dictionary-like ordering that ignores space characters completely and which folds case.

Perl currently only supports single-byte locales for `LC_COLLATE`. This means that a UTF-8 locale likely will just give you machine-native ordering. Use `Unicode-Collate` for the full implementation of the Unicode Collation Algorithm.

If you have a single string that you want to check for "equality in locale" against several others, you might think you could gain a little efficiency by using `POSIX::strxfrm()` in conjunction with `eq`:

```
use POSIX qw(strxfrm);
$xfrm_string = strxfrm("Mixed-case string");
print "locale collation ignores spaces\n"
    if $xfrm_string eq strxfrm("Mixed-casestring");
print "locale collation ignores hyphens\n"
    if $xfrm_string eq strxfrm("Mixedcase string");
print "locale collation ignores case\n"
```

```
            if $xfrm_string eq strxfrm("mixed-case string");
```

strxfrm() takes a string and maps it into a transformed string for use in char-by-char comparisons against other transformed strings during collation. "Under the hood", locale-affected Perl comparison operators call strxfrm() for both operands, then do a char-by-char comparison of the transformed strings. By calling strxfrm() explicitly and using a non locale-affected comparison, the example attempts to save a couple of transformations. But in fact, it doesn't save anything: Perl magic (see Section 28.3.18 [perlguts Magic Variables], page 528) creates the transformed version of a string the first time it's needed in a comparison, then keeps this version around in case it's needed again. An example rewritten the easy way with cmp runs just about as fast. It also copes with null characters embedded in strings; if you call strxfrm() directly, it treats the first null it finds as a terminator. don't expect the transformed strings it produces to be portable across systems–or even from one revision of your operating system to the next. In short, don't call strxfrm() directly: let Perl do it for you.

Note: use locale isn't shown in some of these examples because it isn't needed: strcoll() and strxfrm() are POSIX functions which use the standard system-supplied libc functions that always obey the current LC_COLLATE locale.

38.6.2 Category LC_CTYPE: Character Types

In the scope of a use locale form that includes LC_CTYPE, Perl obeys the LC_CTYPE locale setting. This controls the application's notion of which characters are alphabetic, numeric, punctuation, *etc.* This affects Perl's \w regular expression metanotation, which stands for alphanumeric characters–that is, alphabetic, numeric, and the platform's native underscore. (Consult Section 58.1 [perlre NAME], page 989 for more information about regular expressions.) Thanks to LC_CTYPE, depending on your locale setting, characters like "æ", "ð", "ß", and "ø" may be understood as \w characters. It also affects things like \s, \D, and the POSIX character classes, like [[:graph:]]. (See Section 61.1 [perlrecharclass NAME], page 1059 for more information on all these.)

The LC_CTYPE locale also provides the map used in transliterating characters between lower and uppercase. This affects the case-mapping functions–fc(), lc(), lcfirst(), uc(), and ucfirst(); case-mapping interpolation with \F, \l, \L, \u, or \U in double-quoted strings and s/// substitutions; and case-independent regular expression pattern matching using the i modifier.

Finally, LC_CTYPE affects the (deprecated) POSIX character-class test functions–POSIX::isalpha(), POSIX::islower(), and so on. For example, if you move from the "C" locale to a 7-bit ISO 646 one, you may find–possibly to your surprise–that "|" moves from the POSIX::ispunct() class to POSIX::isalpha(). Unfortunately, this creates big problems for regular expressions. "|" still means alternation even though it matches \w. Starting in v5.22, a warning will be raised when such a locale is switched into. More details are given several paragraphs further down.

Starting in v5.20, Perl supports UTF-8 locales for LC_CTYPE, but otherwise Perl only supports single-byte locales, such as the ISO 8859 series. This means that wide character locales, for example for Asian languages, are not well-supported. (If the platform has the capability for Perl to detect such a locale, starting in Perl v5.22, Section “Category Hierarchy” in warnings, using the locale warning category, whenever such a locale is switched into.) The UTF-8 locale support is actually a superset of POSIX locales, because it is really full Unicode behavior as if no LC_CTYPE locale were in effect at all (except for tainting; see Section 38.7 [SECURITY], page 716). POSIX locales, even UTF-8 ones, are lacking certain concepts in Unicode, such as the idea that changing the case of a character could expand to be more than one character. Perl in a UTF-8 locale, will give you that expansion. Prior to v5.20, Perl treated a UTF-8 locale on some platforms like an ISO 8859-1 one, with some restrictions, and on other platforms more like the "C" locale. For releases v5.16 and v5.18, use locale 'not_characters could be used as a workaround for this (see Section 38.10 [Unicode and UTF-8], page 721).

Note that there are quite a few things that are unaffected by the current locale. Any literal character is the native character for the given platform. Hence 'A' means the character at code point 65 on ASCII platforms, and 193 on EBCDIC. That may or may not be an 'A' in the current locale, if that locale even has an 'A'. Similarly, all the escape sequences for particular characters, \n for example, always mean the platform's native one. This means, for example, that \N in regular expressions (every character but new-line) works on the platform character set.

Starting in v5.22, Perl will by default warn when switching into a locale that redefines any ASCII printable character (plus \t and \n) into a different class than expected. This is likely to happen on modern locales only on EBCDIC platforms, where, for example, a CCSID 0037 locale on a CCSID 1047 machine moves "[", but it can happen on ASCII platforms with the ISO 646 and other 7-bit locales that are essentially obsolete. Things may still work, depending on what features of Perl are used by the program. For example, in the example from above where "|" becomes a \w, and there are no regular expressions where this matters, the program may still work properly. The warning lists all the characters that it can determine could be adversely affected.

Note: A broken or malicious LC_CTYPE locale definition may result in clearly ineligible characters being considered to be alphanumeric by your application. For strict matching of (mundane) ASCII letters and digits–for example, in command strings–locale-aware applications should use \w with the /a regular expression modifier. See Section 38.7 [SECURITY], page 716.

38.6.3 Category LC_NUMERIC: Numeric Formatting

After a proper POSIX::setlocale() call, and within the scope of of a use locale form that includes numerics, Perl obeys the LC_NUMERIC locale information, which controls an application's idea of how numbers should be formatted for human readability. In most implementations the only effect is to change the character used for the decimal point–perhaps from "." to ",". The functions aren't aware of such niceties as thousands separation and so on. (See Section 38.5.10 [The localeconv function], page 710 if you care about these things.)

```
use POSIX qw(strtod setlocale LC_NUMERIC);
use locale;

setlocale LC_NUMERIC, "";

$n = 5/2;   # Assign numeric 2.5 to $n

$a = " $n"; # Locale-dependent conversion to string
```

```
print "half five is $n\n";       # Locale-dependent output

printf "half five is %g\n", $n;  # Locale-dependent output

print "DECIMAL POINT IS COMMA\n"
         if $n == (strtod("2,5"))[0]; # Locale-dependent conversion
```

See also `I18N-Langinfo` and `RADIXCHAR`.

38.6.4 Category `LC_MONETARY`: Formatting of monetary amounts

The C standard defines the `LC_MONETARY` category, but not a function that is affected by its contents. (Those with experience of standards committees will recognize that the working group decided to punt on the issue.) Consequently, Perl essentially takes no notice of it. If you really want to use `LC_MONETARY`, you can query its contents–see Section 38.5.10 [The localeconv function], page 710–and use the information that it returns in your application's own formatting of currency amounts. However, you may well find that the information, voluminous and complex though it may be, still does not quite meet your requirements: currency formatting is a hard nut to crack.

See also `I18N-Langinfo` and `CRNCYSTR`.

38.6.5 `LC_TIME`

Output produced by `POSIX::strftime()`, which builds a formatted human-readable date/time string, is affected by the current `LC_TIME` locale. Thus, in a French locale, the output produced by the `%B` format element (full month name) for the first month of the year would be "janvier". Here's how to get a list of long month names in the current locale:

```
use POSIX qw(strftime);
for (0..11) {
    $long_month_name[$_] =
        strftime("%B", 0, 0, 0, 1, $_, 96);
}
```

Note: `use locale` isn't needed in this example: `strftime()` is a POSIX function which uses the standard system-supplied `libc` function that always obeys the current `LC_TIME` locale.

See also `I18N-Langinfo` and `ABDAY_1..ABDAY_7`, `DAY_1..DAY_7`, `ABMON_1..ABMON_12`, and `ABMON_1..ABMON_12`.

38.6.6 Other categories

The remaining locale categories are not currently used by Perl itself. But again note that things Perl interacts with may use these, including extensions outside the standard Perl distribution, and by the operating system and its utilities. Note especially that the string value of `$!` and the error messages given by external utilities may be changed by `LC_MESSAGES`. If you want to have portable error codes, use `%!`. See `Errno`.

38.7 SECURITY

Although the main discussion of Perl security issues can be found in Section 70.1 [perlsec NAME], page 1198, a discussion of Perl's locale handling would be incomplete if it did not draw your attention to locale-dependent security issues. Locales–particularly on systems that allow unprivileged users to build their own locales–are untrustworthy. A malicious (or just plain broken) locale can make a locale-aware application give unexpected results. Here are a few possibilities:

- Regular expression checks for safe file names or mail addresses using `\w` may be spoofed by an `LC_CTYPE` locale that claims that characters such as `">"` and `"|"` are alphanumeric.
- String interpolation with case-mapping, as in, say, `$dest = "C:\U$name.$ext"`, may produce dangerous results if a bogus `LC_CTYPE` case-mapping table is in effect.
- A sneaky `LC_COLLATE` locale could result in the names of students with "D" grades appearing ahead of those with "A"s.
- An application that takes the trouble to use information in `LC_MONETARY` may format debits as if they were credits and vice versa if that locale has been subverted. Or it might make payments in US dollars instead of Hong Kong dollars.
- The date and day names in dates formatted by `strftime()` could be manipulated to advantage by a malicious user able to subvert the `LC_DATE` locale. ("Look–it says I wasn't in the building on Sunday.")

Such dangers are not peculiar to the locale system: any aspect of an application's environment which may be modified maliciously presents similar challenges. Similarly, they are not specific to Perl: any programming language that allows you to write programs that take account of their environment exposes you to these issues.

Perl cannot protect you from all possibilities shown in the examples–there is no substitute for your own vigilance–but, when `use locale` is in effect, Perl uses the tainting mechanism (see Section 70.1 [perlsec NAME], page 1198) to mark string results that become locale-dependent, and which may be untrustworthy in consequence. Here is a summary of the tainting behavior of operators and functions that may be affected by the locale:

- **Comparison operators** (`lt`, `le`, `ge`, `gt` and `cmp`):

  Scalar true/false (or less/equal/greater) result is never tainted.
- **Case-mapping interpolation** (with `\l`, `\L`, `\u`, `\U`, or `\F`)

  The result string containing interpolated material is tainted if a `use locale` form that includes `LC_CTYPE` is in effect.
- **Matching operator** (`m//`):

  Scalar true/false result never tainted.

  All subpatterns, either delivered as a list-context result or as `$1` *etc.*, are tainted if a `use locale` form that includes `LC_CTYPE` is in effect, and the subpattern regular expression contains a locale-dependent construct. These constructs include `\w` (to match an alphanumeric character), `\W` (non-alphanumeric character), `\b` and `\B` (word-boundary and non-boundardy, which depend on what `\w` and `\W` match), `\s` (whitespace character), `\S` (non whitespace character), `\d` and `\D` (digits and non-digits), and the POSIX character classes, such as `[:alpha:]` (see Section 61.2.3.5 [perlrecharclass POSIX Character Classes], page 1069).

Tainting is also likely if the pattern is to be matched case-insensitively (via /i). The exception is if all the code points to be matched this way are above 255 and do not have folds under Unicode rules to below 256. Tainting is not done for these because Perl only uses Unicode rules for such code points, and those rules are the same no matter what the current locale.

The matched-pattern variables, $&, $‘ (pre-match), $’ (post-match), and $+ (last match) also are tainted.

- **Substitution operator** (s///):

  Has the same behavior as the match operator. Also, the left operand of =~ becomes tainted when a use locale form that includes LC_CTYPE is in effect, if modified as a result of a substitution based on a regular expression match involving any of the things mentioned in the previous item, or of case-mapping, such as \l, \L,\u, \U, or \F.

- **Output formatting functions** (printf() and write()):

  Results are never tainted because otherwise even output from print, for example print(1/7), should be tainted if use locale is in effect.

- **Case-mapping functions** (lc(), lcfirst(), uc(), ucfirst()):

  Results are tainted if a use locale form that includes LC_CTYPE is in effect.

- **POSIX locale-dependent functions** (localeconv(), strcoll(), strftime(), strxfrm()):

  Results are never tainted.

- **POSIX character class tests** (POSIX::isalnum(), POSIX::isalpha(), POSIX::isdigit(), POSIX::isgraph(), POSIX::islower(), POSIX::isprint(), POSIX::ispunct(), POSIX::isspace(), POSIX::isupper(), POSIX::isxdigit()):

  True/false results are never tainted.

Three examples illustrate locale-dependent tainting. The first program, which ignores its locale, won’t run: a value taken directly from the command line may not be used to name an output file when taint checks are enabled.

```
#/usr/local/bin/perl -T
# Run with taint checking

# Command line sanity check omitted...
$tainted_output_file = shift;

open(F, ">$tainted_output_file")
    or warn "Open of $tainted_output_file failed: $!\n";
```

The program can be made to run by "laundering" the tainted value through a regular expression: the second example–which still ignores locale information–runs, creating the file named on its command line if it can.

```
#/usr/local/bin/perl -T

$tainted_output_file = shift;
$tainted_output_file =~ m%[\w/]+%;
$untainted_output_file = $&;
```

```
        open(F, ">$untainted_output_file")
            or warn "Open of $untainted_output_file failed: $!\n";
```

Compare this with a similar but locale-aware program:

```
        #/usr/local/bin/perl -T

        $tainted_output_file = shift;
        use locale;
        $tainted_output_file =~ m%[\w/]+%;
        $localized_output_file = $&;

        open(F, ">$localized_output_file")
            or warn "Open of $localized_output_file failed: $!\n";
```

This third program fails to run because `$&` is tainted: it is the result of a match involving `\w` while `use locale` is in effect.

38.8 ENVIRONMENT

PERL_SKIP_LOCALE_INIT

This environment variable, available starting in Perl v5.20, if set (to any value), tells Perl to not use the rest of the environment variables to initialize with. Instead, Perl uses whatever the current locale settings are. This is particularly useful in embedded environments, see Section 20.2.14 [perlembed Using embedded Perl with POSIX locales], page 320.

PERL_BADLANG

A string that can suppress Perl's warning about failed locale settings at startup. Failure can occur if the locale support in the operating system is lacking (broken) in some way–or if you mistyped the name of a locale when you set up your environment. If this environment variable is absent, or has a value other than `"0"` or `""`, Perl will complain about locale setting failures.

NOTE: `PERL_BADLANG` only gives you a way to hide the warning message. The message tells about some problem in your system's locale support, and you should investigate what the problem is.

The following environment variables are not specific to Perl: They are part of the standardized (ISO C, XPG4, POSIX 1.c) `setlocale()` method for controlling an application's opinion on data. Windows is non-POSIX, but Perl arranges for the following to work as described anyway. If the locale given by an environment variable is not valid, Perl tries the next lower one in priority. If none are valid, on Windows, the system default locale is then tried. If all else fails, the `"C"` locale is used. If even that doesn't work, something is badly broken, but Perl tries to forge ahead with whatever the locale settings might be.

`LC_ALL`

`LC_ALL` is the "override-all" locale environment variable. If set, it overrides all the rest of the locale environment variables.

`LANGUAGE`

NOTE: LANGUAGE is a GNU extension, it affects you only if you are using the GNU libc. This is the case if you are using e.g. Linux. If you are using "commercial" Unixes you are most probably *not* using GNU libc and you can ignore LANGUAGE.

However, in the case you are using LANGUAGE: it affects the language of informational, warning, and error messages output by commands (in other words, it's like LC_MESSAGES) but it has higher priority than LC_ALL. Moreover, it's not a single value but instead a "path" (":"-separated list) of *languages* (not locales). See the GNU gettext library documentation for more information.

LC_CTYPE

In the absence of LC_ALL, LC_CTYPE chooses the character type locale. In the absence of both LC_ALL and LC_CTYPE, LANG chooses the character type locale.

LC_COLLATE

In the absence of LC_ALL, LC_COLLATE chooses the collation (sorting) locale. In the absence of both LC_ALL and LC_COLLATE, LANG chooses the collation locale.

LC_MONETARY

In the absence of LC_ALL, LC_MONETARY chooses the monetary formatting locale. In the absence of both LC_ALL and LC_MONETARY, LANG chooses the monetary formatting locale.

LC_NUMERIC

In the absence of LC_ALL, LC_NUMERIC chooses the numeric format locale. In the absence of both LC_ALL and LC_NUMERIC, LANG chooses the numeric format.

LC_TIME

In the absence of LC_ALL, LC_TIME chooses the date and time formatting locale. In the absence of both LC_ALL and LC_TIME, LANG chooses the date and time formatting locale.

LANG

LANG is the "catch-all" locale environment variable. If it is set, it is used as the last resort after the overall LC_ALL and the category-specific LC_*foo*.

38.8.1 Examples

The LC_NUMERIC controls the numeric output:

```
use locale;
use POSIX qw(locale_h); # Imports setlocale() and the LC_ constants.
setlocale(LC_NUMERIC, "fr_FR") or die "Pardon";
printf "%g\n", 1.23; # If the "fr_FR" succeeded, probably shows 1,23.
```

and also how strings are parsed by POSIX::strtod() as numbers:

```
use locale;
use POSIX qw(locale_h strtod);
setlocale(LC_NUMERIC, "de_DE") or die "Entschuldigung";
my $x = strtod("2,34") + 5;
print $x, "\n"; # Probably shows 7,34.
```

38.9 NOTES

38.9.1 String eval and LC_NUMERIC

A string [eval], page 377 parses its expression as standard Perl. It is therefore expecting the decimal point to be a dot. If LC_NUMERIC is set to have this be a comma instead, the parsing will be confused, perhaps silently.

```
use locale;
use POSIX qw(locale_h);
setlocale(LC_NUMERIC, "fr_FR") or die "Pardon";
my $a = 1.2;
print eval "$a + 1.5";
print "\n";
```

prints 13,5. This is because in that locale, the comma is the decimal point character. The eval thus expands to:

```
eval "1,2 + 1.5"
```

and the result is not what you likely expected. No warnings are generated. If you do string eval's within the scope of use locale, you should instead change the eval line to do something like:

```
print eval "no locale; $a + 1.5";
```

This prints 2.7.

You could also exclude LC_NUMERIC, if you don't need it, by

```
use locale ':!numeric';
```

38.9.2 Backward compatibility

Versions of Perl prior to 5.004 **mostly** ignored locale information, generally behaving as if something similar to the "C" locale were always in force, even if the program environment suggested otherwise (see Section 38.5.2 [The setlocale function], page 705). By default, Perl still behaves this way for backward compatibility. If you want a Perl application to pay attention to locale information, you **must** use the use locale pragma (see Section 38.5.1 [The "use locale" pragma], page 703) or, in the unlikely event that you want to do so for just pattern matching, the /l regular expression modifier (see Section 58.2.1.2 [perlre Character set modifiers], page 992) to instruct it to do so.

Versions of Perl from 5.002 to 5.003 did use the LC_CTYPE information if available; that is, \w did understand what were the letters according to the locale environment variables. The problem was that the user had no control over the feature: if the C library supported locales, Perl used them.

38.9.3 I18N:Collate obsolete

In versions of Perl prior to 5.004, per-locale collation was possible using the I18N::Collate library module. This module is now mildly obsolete and should be avoided in new applications. The LC_COLLATE functionality is now integrated into the Perl core language: One can use locale-specific scalar data completely normally with use locale, so there is no longer any need to juggle with the scalar references of I18N::Collate.

38.9.4 Sort speed and memory use impacts

Comparing and sorting by locale is usually slower than the default sorting; slow-downs of two to four times have been observed. It will also consume more memory: once a Perl scalar variable has participated in any string comparison or sorting operation obeying the locale collation rules, it will take 3-15 times more memory than before. (The exact multiplier depends on the string's contents, the operating system and the locale.) These downsides are dictated more by the operating system's implementation of the locale system than by Perl.

38.9.5 Freely available locale definitions

The Unicode CLDR project extracts the POSIX portion of many of its locales, available at

`http://unicode.org/Public/cldr/latest/`

There is a large collection of locale definitions at:

`http://std.dkuug.dk/i18n/WG15-collection/locales/`

You should be aware that it is unsupported, and is not claimed to be fit for any purpose. If your system allows installation of arbitrary locales, you may find the definitions useful as they are, or as a basis for the development of your own locales.

38.9.6 I18n and l10n

"Internationalization" is often abbreviated as **i18n** because its first and last letters are separated by eighteen others. (You may guess why the internalin ... internaliti ... i18n tends to get abbreviated.) In the same way, "localization" is often abbreviated to **l10n**.

38.9.7 An imperfect standard

Internationalization, as defined in the C and POSIX standards, can be criticized as incomplete, ungainly, and having too large a granularity. (Locales apply to a whole process, when it would arguably be more useful to have them apply to a single thread, window group, or whatever.) They also have a tendency, like standards groups, to divide the world into nations, when we all know that the world can equally well be divided into bankers, bikers, gamers, and so on.

38.10 Unicode and UTF-8

The support of Unicode is new starting from Perl version v5.6, and more fully implemented in versions v5.8 and later. See Section 83.1 [perluniintro NAME], page 1352.

Starting in Perl v5.20, UTF-8 locales are supported in Perl, except for `LC_COLLATE` (use `Unicode-Collate` instead). If you have Perl v5.16 or v5.18 and can't upgrade, you can use

```
use locale ':not_characters';
```

When this form of the pragma is used, only the non-character portions of locales are used by Perl, for example `LC_NUMERIC`. Perl assumes that you have translated all the characters it is to operate on into Unicode (actually the platform's native character set (ASCII or EBCDIC) plus Unicode). For data in files, this can conveniently be done by also specifying

```
use open ':locale';
```

This pragma arranges for all inputs from files to be translated into Unicode from the current locale as specified in the environment (see Section 38.8 [ENVIRONMENT], page 718), and all outputs to files to be translated back into the locale. (See `open`). On a per-filehandle basis, you can instead use the `PerlIO-locale` module, or the `Encode-Locale` module, both available from CPAN. The latter module also has methods to ease the handling of `ARGV` and environment variables, and can be used on individual strings. If you know that all your locales will be UTF-8, as many are these days, you can use the ⟨undefined⟩ [**-C**], page ⟨undefined⟩ command line switch.

This form of the pragma allows essentially seamless handling of locales with Unicode. The collation order will be by Unicode code point order. It is strongly recommended that when you need to order and sort strings that you use the standard module `Unicode-Collate` which gives much better results in many instances than you can get with the old-style locale handling.

All the modules and switches just described can be used in v5.20 with just plain `use locale`, and, should the input locales not be UTF-8, you'll get the less than ideal behavior, described below, that you get with pre-v5.16 Perls, or when you use the locale pragma without the `:not_characters` parameter in v5.16 and v5.18. If you are using exclusively UTF-8 locales in v5.20 and higher, the rest of this section does not apply to you.

There are two cases, multi-byte and single-byte locales. First multi-byte:

The only multi-byte (or wide character) locale that Perl is ever likely to support is UTF-8. This is due to the difficulty of implementation, the fact that high quality UTF-8 locales are now published for every area of the world (`http://unicode.org/Public/cldr/latest/`), and that failing all that you can use the `Encode` module to translate to/from your locale. So, you'll have to do one of those things if you're using one of these locales, such as Big5 or Shift JIS. For UTF-8 locales, in Perls (pre v5.20) that don't have full UTF-8 locale support, they may work reasonably well (depending on your C library implementation) simply because both they and Perl store characters that take up multiple bytes the same way. However, some, if not most, C library implementations may not process the characters in the upper half of the Latin-1 range (128 - 255) properly under `LC_CTYPE`. To see if a character is a particular type under a locale, Perl uses the functions like `isalnum()`. Your C library may not work for UTF-8 locales with those functions, instead only working under the newer wide library functions like `iswalnum()`, which Perl does not use. These multi-byte locales are treated like single-byte locales, and will have the restrictions described below. Starting in Perl v5.22 a warning message is raised when Perl detects a multi-byte locale that it doesn't fully support.

For single-byte locales, Perl generally takes the tack to use locale rules on code points that can fit in a single byte, and Unicode rules for those that can't (though this isn't uniformly applied, see the note at the end of this section). This prevents many problems in locales that aren't UTF-8. Suppose the locale is ISO8859-7, Greek. The character at 0xD7 there is a capital Chi. But in the ISO8859-1 locale, Latin1, it is a multiplication sign. The POSIX regular expression character class `[[:alpha:]]` will magically match 0xD7 in the Greek locale but not in the Latin one.

However, there are places where this breaks down. Certain Perl constructs are for Unicode only, such as `\p{Alpha}`. They assume that 0xD7 always has its Unicode meaning (or the equivalent on EBCDIC platforms). Since Latin1 is a subset of Unicode and 0xD7 is the multiplication sign in both Latin1 and Unicode, `\p{Alpha}` will never match it, regardless of locale. A similar issue occurs with `\N{...}`. Prior to v5.20, It is therefore a bad idea to use `\p{}` or `\N{}` under plain `use locale`–*unless* you can guarantee that the locale will be ISO8859-1. Use POSIX character classes instead.

Another problem with this approach is that operations that cross the single byte/multiple byte boundary are not well-defined, and so are disallowed. (This boundary is between the codepoints at 255/256.) For example, lower casing LATIN CAPITAL LETTER Y WITH DIAERESIS (U+0178) should return LATIN SMALL LETTER Y WITH DIAERESIS (U+00FF). But in the Greek locale, for example, there is no character at 0xFF, and Perl has no way of knowing what the character at 0xFF is really supposed to represent. Thus it disallows the operation. In this mode, the lowercase of U+0178 is itself.

The same problems ensue if you enable automatic UTF-8-ification of your standard file handles, default `open()` layer, and `@ARGV` on non-ISO8859-1, non-UTF-8 locales (by using either the **-C** command line switch or the `PERL_UNICODE` environment variable; see Section 69.1 [perlrun NAME], page 1176). Things are read in as UTF-8, which would normally imply a Unicode interpretation, but the presence of a locale causes them to be interpreted in that locale instead. For example, a 0xD7 code point in the Unicode input, which should mean the multiplication sign, won't be interpreted by Perl that way under the Greek locale. This is not a problem *provided* you make certain that all locales will always and only be either an ISO8859-1, or, if you don't have a deficient C library, a UTF-8 locale.

Still another problem is that this approach can lead to two code points meaning the same character. Thus in a Greek locale, both U+03A7 and U+00D7 are GREEK CAPITAL LETTER CHI.

Because of all these problems, starting in v5.22, Perl will raise a warning if a multi-byte (hence Unicode) code point is used when a single-byte locale is in effect. (Although it doesn't check for this if doing so would unreasonably slow execution down.)

Vendor locales are notoriously buggy, and it is difficult for Perl to test its locale-handling code because this interacts with code that Perl has no control over; therefore the locale-handling code in Perl may be buggy as well. (However, the Unicode-supplied locales should be better, and there is a feed back mechanism to correct any problems. See Section 38.9.5 [Freely available locale definitions], page 721.)

If you have Perl v5.16, the problems mentioned above go away if you use the `:not_characters` parameter to the locale pragma (except for vendor bugs in the non-character portions). If you don't have v5.16, and you *do* have locales that work, using them may be worthwhile for certain specific purposes, as long as you keep in mind the gotchas already mentioned. For example, if the collation for your locales works, it runs faster under locales than under `Unicode-Collate`; and you gain access to such things as the local currency symbol and the names of the months and days of the week. (But to hammer home the point, in v5.16, you get this access without the downsides of locales by using the `:not_characters` form of the pragma.)

Note: The policy of using locale rules for code points that can fit in a byte, and Unicode rules for those that can't is not uniformly applied. Pre-v5.12, it was somewhat haphazard; in v5.12 it was applied fairly consistently to regular expression matching except for bracketed character classes; in v5.14 it was extended to all regex matches; and in v5.16 to the casing operations such as `\L` and `uc()`. For collation, in all releases so far, the system's `strxfrm()` function is called, and whatever it does is what you get.

38.11 BUGS

38.11.1 Broken systems

In certain systems, the operating system's locale support is broken and cannot be fixed or used by Perl. Such deficiencies can and will result in mysterious hangs and/or Perl core dumps when `use locale` is in effect. When confronted with such a system, please report in excruciating detail to `<perlbug@perl.org>`, and also contact your vendor: bug fixes may exist for these problems in your operating system. Sometimes such bug fixes are called an operating system upgrade. If you have the source for Perl, include in the perlbug email the output of the test described above in Section 38.5.5 [Testing for broken locales], page 708.

38.12 SEE ALSO

`I18N-Langinfo`, Section 83.1 [perluniintro NAME], page 1352, Section 81.1 [perlunicode NAME], page 1317, `open`, Section "isalnum" in `POSIX`, Section "isalpha" in `POSIX`, Section "isdigit" in `POSIX`, Section "isgraph" in `POSIX`, Section "islower" in `POSIX`, Section "isprint" in `POSIX`, Section "ispunct" in `POSIX`, Section "isspace" in `POSIX`, Section "isupper" in `POSIX`, Section "isxdigit" in `POSIX`, Section "localeconv" in `POSIX`, Section "setlocale" in `POSIX`, Section "strcoll" in `POSIX`, Section "strftime" in `POSIX`, Section "strtod" in `POSIX`, Section "strxfrm" in `POSIX`.

For special considerations when Perl is embedded in a C program, see Section 20.2.14 [perlembed Using embedded Perl with POSIX locales], page 320.

38.13 HISTORY

Jarkko Hietaniemi's original `perli18n.pod` heavily hacked by Dominic Dunlop, assisted by the perl5-porters. Prose worked over a bit by Tom Christiansen, and updated by Perl 5 porters.

39 perllol

39.1 NAME

perllol - Manipulating Arrays of Arrays in Perl

39.2 DESCRIPTION

39.2.1 Declaration and Access of Arrays of Arrays

The simplest two-level data structure to build in Perl is an array of arrays, sometimes casually called a list of lists. It's reasonably easy to understand, and almost everything that applies here will also be applicable later on with the fancier data structures.

An array of an array is just a regular old array @AoA that you can get at with two subscripts, like `$AoA[3][2]`. Here's a declaration of the array:

```
    use 5.010;  # so we can use say()

    # assign to our array, an array of array references
    @AoA = (
           [ "fred", "barney", "pebbles", "bambam", "dino", ],
           [ "george", "jane", "elroy", "judy", ],
           [ "homer", "bart", "marge", "maggie", ],
    );
    say $AoA[2][1];
  bart
```

Now you should be very careful that the outer bracket type is a round one, that is, a parenthesis. That's because you're assigning to an @array, so you need parentheses. If you wanted there *not* to be an @AoA, but rather just a reference to it, you could do something more like this:

```
    # assign a reference to array of array references
    $ref_to_AoA = [
        [ "fred", "barney", "pebbles", "bambam", "dino", ],
        [ "george", "jane", "elroy", "judy", ],
        [ "homer", "bart", "marge", "maggie", ],
    ];
    say $ref_to_AoA->[2][1];
  bart
```

Notice that the outer bracket type has changed, and so our access syntax has also changed. That's because unlike C, in perl you can't freely interchange arrays and references thereto. $ref_to_AoA is a reference to an array, whereas @AoA is an array proper. Likewise, `$AoA[2]` is not an array, but an array ref. So how come you can write these:

```
    $AoA[2][2]
    $ref_to_AoA->[2][2]
```

instead of having to write these:

```
$AoA[2]->[2]
$ref_to_AoA->[2]->[2]
```

Well, that's because the rule is that on adjacent brackets only (whether square or curly), you are free to omit the pointer dereferencing arrow. But you cannot do so for the very first one if it's a scalar containing a reference, which means that $ref_to_AoA always needs it.

39.2.2 Growing Your Own

That's all well and good for declaration of a fixed data structure, but what if you wanted to add new elements on the fly, or build it up entirely from scratch?

First, let's look at reading it in from a file. This is something like adding a row at a time. We'll assume that there's a flat file in which each line is a row and each word an element. If you're trying to develop an @AoA array containing all these, here's the right way to do that:

```
while (<>) {
    @tmp = split;
    push @AoA, [ @tmp ];
}
```

You might also have loaded that from a function:

```
for $i ( 1 .. 10 ) {
    $AoA[$i] = [ somefunc($i) ];
}
```

Or you might have had a temporary variable sitting around with the array in it.

```
for $i ( 1 .. 10 ) {
    @tmp = somefunc($i);
    $AoA[$i] = [ @tmp ];
}
```

It's important you make sure to use the `[ ]` array reference constructor. That's because this wouldn't work:

```
$AoA[$i] = @tmp;   # WRONG!
```

The reason that doesn't do what you want is because assigning a named array like that to a scalar is taking an array in scalar context, which means just counts the number of elements in @tmp.

If you are running under `use strict` (and if you aren't, why in the world aren't you?), you'll have to add some declarations to make it happy:

```
use strict;
my(@AoA, @tmp);
while (<>) {
    @tmp = split;
    push @AoA, [ @tmp ];
}
```

Of course, you don't need the temporary array to have a name at all:

```
while (<>) {
    push @AoA, [ split ];
}
```

You also don't have to use push(). You could just make a direct assignment if you knew where you wanted to put it:

```
my (@AoA, $i, $line);
for $i ( 0 .. 10 ) {
    $line = <>;
    $AoA[$i] = [ split " ", $line ];
}
```

or even just

```
my (@AoA, $i);
for $i ( 0 .. 10 ) {
    $AoA[$i] = [ split " ", <> ];
}
```

You should in general be leery of using functions that could potentially return lists in scalar context without explicitly stating such. This would be clearer to the casual reader:

```
my (@AoA, $i);
for $i ( 0 .. 10 ) {
    $AoA[$i] = [ split " ", scalar(<>) ];
}
```

If you wanted to have a $ref_to_AoA variable as a reference to an array, you'd have to do something like this:

```
while (<>) {
    push @$ref_to_AoA, [ split ];
}
```

Now you can add new rows. What about adding new columns? If you're dealing with just matrices, it's often easiest to use simple assignment:

```
for $x (1 .. 10) {
    for $y (1 .. 10) {
        $AoA[$x][$y] = func($x, $y);
    }
}

for $x ( 3, 7, 9 ) {
    $AoA[$x][20] += func2($x);
}
```

It doesn't matter whether those elements are already there or not: it'll gladly create them for you, setting intervening elements to `undef` as need be.

If you wanted just to append to a row, you'd have to do something a bit funnier looking:

```
# add new columns to an existing row
push @{ $AoA[0] }, "wilma", "betty";   # explicit deref
```

Prior to Perl 5.14, this wouldn't even compile:

```
push $AoA[0], "wilma", "betty";        # implicit deref
```

How come? Because once upon a time, the argument to push() had to be a real array, not just a reference to one. That's no longer true. In fact, the line marked "implicit deref" above works just fine–in this instance–to do what the one that says explicit deref did.

The reason I said "in this instance" is because that *only* works because `$AoA[0]` already held an array reference. If you try that on an undefined variable, you'll take an exception. That's because the implicit derefererence will never autovivify an undefined variable the way `@{ }` always will:

```
my $aref = undef;
push $aref,  qw(some more values);  # WRONG!
push @$aref, qw(a few more);        # ok
```

If you want to take advantage of this new implicit dereferencing behavior, go right ahead: it makes code easier on the eye and wrist. Just understand that older releases will choke on it during compilation. Whenever you make use of something that works only in some given release of Perl and later, but not earlier, you should place a prominent

```
use v5.14;   # needed for implicit deref of array refs by array ops
```

directive at the top of the file that needs it. That way when somebody tries to run the new code under an old perl, rather than getting an error like

```
Type of arg 1 to push must be array (not array element) at /tmp/a line 8, near ""betty"
Execution of /tmp/a aborted due to compilation errors.
```

they'll be politely informed that

```
Perl v5.14.0 required--this is only v5.12.3, stopped at /tmp/a line 1.
BEGIN failed--compilation aborted at /tmp/a line 1.
```

39.2.3 Access and Printing

Now it's time to print your data structure out. How are you going to do that? Well, if you want only one of the elements, it's trivial:

```
print $AoA[0][0];
```

If you want to print the whole thing, though, you can't say

```
print @AoA;         # WRONG
```

because you'll get just references listed, and perl will never automatically dereference things for you. Instead, you have to roll yourself a loop or two. This prints the whole structure, using the shell-style for() construct to loop across the outer set of subscripts.

```
for $aref ( @AoA ) {
    say "\t [ @$aref ],";
}
```

If you wanted to keep track of subscripts, you might do this:

```
for $i ( 0 .. $#AoA ) {
    say "\t elt $i is [ @{$AoA[$i]} ],";
}
```

or maybe even this. Notice the inner loop.

```
for $i ( 0 .. $#AoA ) {
    for $j ( 0 .. $#{$AoA[$i]} ) {
        say "elt $i $j is $AoA[$i][$j]";
        }
    }
```

As you can see, it's getting a bit complicated. That's why sometimes is easier to take a temporary on your way through:

```
    for $i ( 0 .. $#AoA ) {
        $aref = $AoA[$i];
        for $j ( 0 .. $#{$aref} ) {
            say "elt $i $j is $AoA[$i][$j]";
        }
    }
```

Hmm... that's still a bit ugly. How about this:

```
    for $i ( 0 .. $#AoA ) {
        $aref = $AoA[$i];
        $n = @$aref - 1;
        for $j ( 0 .. $n ) {
            say "elt $i $j is $AoA[$i][$j]";
        }
    }
```

When you get tired of writing a custom print for your data structures, you might look at the standard Dumpvalue or Data-Dumper modules. The former is what the Perl debugger uses, while the latter generates parsable Perl code. For example:

```
    use v5.14;     # using the + prototype, new to v5.14

    sub show(+) {
        require Dumpvalue;
        state $prettily = new Dumpvalue::
                            tick        => q("),
                            compactDump => 1,  # comment these two lines out
                            veryCompact => 1,  # if you want a bigger dump
                        ;
        dumpValue $prettily @_;
    }

    # Assign a list of array references to an array.
    my @AoA = (
           [ "fred", "barney" ],
           [ "george", "jane", "elroy" ],
           [ "homer", "marge", "bart" ],
    );
    push $AoA[0], "wilma", "betty";
    show @AoA;
```

will print out:

```
    0  0..3  "fred" "barney" "wilma" "betty"
    1  0..2  "george" "jane" "elroy"
    2  0..2  "homer" "marge" "bart"
```

Whereas if you comment out the two lines I said you might wish to, then it shows it to you this way instead:

```
0  ARRAY(0x8031d0)
   0  "fred"
   1  "barney"
   2  "wilma"
   3  "betty"
1  ARRAY(0x803d40)
   0  "george"
   1  "jane"
   2  "elroy"
2  ARRAY(0x803e10)
   0  "homer"
   1  "marge"
   2  "bart"
```

39.2.4 Slices

If you want to get at a slice (part of a row) in a multidimensional array, you're going to have to do some fancy subscripting. That's because while we have a nice synonym for single elements via the pointer arrow for dereferencing, no such convenience exists for slices.

Here's how to do one operation using a loop. We'll assume an @AoA variable as before.

```
@part = ();
$x = 4;
for ($y = 7; $y < 13; $y++) {
    push @part, $AoA[$x][$y];
}
```

That same loop could be replaced with a slice operation:

```
@part = @{$AoA[4]}[7..12];
```

or spaced out a bit:

```
@part = @{ $AoA[4] } [ 7..12 ];
```

But as you might well imagine, this can get pretty rough on the reader.

Ah, but what if you wanted a *two-dimensional slice*, such as having $x run from 4..8 and $y run from 7 to 12? Hmm... here's the simple way:

```
@newAoA = ();
for ($startx = $x = 4; $x <= 8; $x++) {
    for ($starty = $y = 7; $y <= 12; $y++) {
        $newAoA[$x - $startx][$y - $starty] = $AoA[$x][$y];
    }
}
```

We can reduce some of the looping through slices

```
for ($x = 4; $x <= 8; $x++) {
    push @newAoA, [ @{ $AoA[$x] } [ 7..12 ] ];
}
```

If you were into Schwartzian Transforms, you would probably have selected map for that

```
@newAoA = map { [ @{ $AoA[$_] } [ 7..12 ] ] } 4 .. 8;
```

Although if your manager accused you of seeking job security (or rapid insecurity) through inscrutable code, it would be hard to argue. :-) If I were you, I'd put that in a function:

```
@newAoA = splice_2D( \@AoA, 4 => 8, 7 => 12 );
sub splice_2D {
    my $lrr = shift;        # ref to array of array refs!
    my ($x_lo, $x_hi,
        $y_lo, $y_hi) = @_;

    return map {
        [ @{ $lrr->[$_] } [ $y_lo .. $y_hi ] ]
    } $x_lo .. $x_hi;
}
```

39.3 SEE ALSO

Section 11.1 [perldata NAME], page 70, Section 62.1 [perlref NAME], page 1077, Section 17.1 [perldsc NAME], page 246

39.4 AUTHOR

Tom Christiansen `<tchrist@perl.com>`

Last update: Tue Apr 26 18:30:55 MDT 2011

40 perlmod

40.1 NAME

perlmod - Perl modules (packages and symbol tables)

40.2 DESCRIPTION

40.2.1 Is this the document you were after?

There are other documents which might contain the information that you're looking for:

This doc

Perl's packages, namespaces, and some info on classes.

Section 44.1 [perlnewmod NAME], page 760

Tutorial on making a new module.

Section 42.1 [perlmodstyle NAME], page 748

Best practices for making a new module.

40.2.2 Packages

Perl provides a mechanism for alternative namespaces to protect packages from stomping on each other's variables. In fact, there's really no such thing as a global variable in Perl. The package statement declares the compilation unit as being in the given namespace. The scope of the package declaration is from the declaration itself through the end of the enclosing block, `eval`, or file, whichever comes first (the same scope as the my() and local() operators). Unqualified dynamic identifiers will be in this namespace, except for those few identifiers that if unqualified, default to the main package instead of the current one as described below. A package statement affects only dynamic variables–including those you've used local() on–but *not* lexical variables created with my(). Typically it would be the first declaration in a file included by the `do`, `require`, or `use` operators. You can switch into a package in more than one place; it merely influences which symbol table is used by the compiler for the rest of that block. You can refer to variables and filehandles in other packages by prefixing the identifier with the package name and a double colon: `$Package::Variable`. If the package name is null, the `main` package is assumed. That is, `$::sail` is equivalent to `$main::sail`.

The old package delimiter was a single quote, but double colon is now the preferred delimiter, in part because it's more readable to humans, and in part because it's more readable to **emacs** macros. It also makes C++ programmers feel like they know what's going on–as opposed to using the single quote as separator, which was there to make Ada programmers feel like they knew what was going on. Because the old-fashioned syntax is still supported for backwards compatibility, if you try to use a string like `"This is $owner's house"`, you'll be accessing `$owner::s`; that is, the $s variable in package `owner`, which is probably not what you meant. Use braces to disambiguate, as in `"This is ${owner}'s house"`.

Packages may themselves contain package separators, as in `$OUTER::INNER::var`. This implies nothing about the order of name lookups, however. There are no relative packages:

all symbols are either local to the current package, or must be fully qualified from the outer package name down. For instance, there is nowhere within package OUTER that $INNER::var refers to $OUTER::INNER::var. INNER refers to a totally separate global package.

Only identifiers starting with letters (or underscore) are stored in a package's symbol table. All other symbols are kept in package main, including all punctuation variables, like $_. In addition, when unqualified, the identifiers STDIN, STDOUT, STDERR, ARGV, ARGVOUT, ENV, INC, and SIG are forced to be in package main, even when used for other purposes than their built-in ones. If you have a package called m, s, or y, then you can't use the qualified form of an identifier because it would be instead interpreted as a pattern match, a substitution, or a transliteration.

Variables beginning with underscore used to be forced into package main, but we decided it was more useful for package writers to be able to use leading underscore to indicate private variables and method names. However, variables and functions named with a single _, such as $_ and sub _, are still forced into the package main. See also Section 86.2.1 [perlvar The Syntax of Variable Names], page 1375.

evaled strings are compiled in the package in which the eval() was compiled. (Assignments to $SIG{}, however, assume the signal handler specified is in the main package. Qualify the signal handler name if you wish to have a signal handler in a package.) For an example, examine perldb.pl in the Perl library. It initially switches to the DB package so that the debugger doesn't interfere with variables in the program you are trying to debug. At various points, however, it temporarily switches back to the main package to evaluate various expressions in the context of the main package (or wherever you came from). See Section 15.1 [perldebug NAME], page 120.

The special symbol __PACKAGE__ contains the current package, but cannot (easily) be used to construct variable names.

See Section 73.1 [perlsub NAME], page 1216 for other scoping issues related to my() and local(), and Section 62.1 [perlref NAME], page 1077 regarding closures.

40.2.3 Symbol Tables

The symbol table for a package happens to be stored in the hash of that name with two colons appended. The main symbol table's name is thus %main::, or %:: for short. Likewise the symbol table for the nested package mentioned earlier is named %OUTER::INNER::.

The value in each entry of the hash is what you are referring to when you use the *name typeglob notation.

```
   local *main::foo    = *main::bar;
```

You can use this to print out all the variables in a package, for instance. The standard but antiquated dumpvar.pl library and the CPAN module Devel::Symdump make use of this.

The results of creating new symbol table entries directly or modifying any entries that are not already typeglobs are undefined and subject to change between releases of perl.

Assignment to a typeglob performs an aliasing operation, i.e.,

```
    *dick = *richard;
```

causes variables, subroutines, formats, and file and directory handles accessible via the identifier richard also to be accessible via the identifier dick. If you want to alias only a particular variable or subroutine, assign a reference instead:

```
*dick = \$richard;
```

Which makes $richard and $dick the same variable, but leaves @richard and @dick as separate arrays. Tricky, eh?

There is one subtle difference between the following statements:

```
*foo = *bar;
*foo = \$bar;
```

`*foo = *bar` makes the typeglobs themselves synonymous while `*foo = \$bar` makes the SCALAR portions of two distinct typeglobs refer to the same scalar value. This means that the following code:

```
$bar = 1;
*foo = \$bar;       # Make $foo an alias for $bar

{
    local $bar = 2; # Restrict changes to block
    print $foo;     # Prints '1'!
}
```

Would print '1', because `$foo` holds a reference to the *original* `$bar`. The one that was stuffed away by `local()` and which will be restored when the block ends. Because variables are accessed through the typeglob, you can use `*foo = *bar` to create an alias which can be localized. (But be aware that this means you can't have a separate `@foo` and `@bar`, etc.)

What makes all of this important is that the Exporter module uses glob aliasing as the import/export mechanism. Whether or not you can properly localize a variable that has been exported from a module depends on how it was exported:

```
@EXPORT = qw($FOO); # Usual form, can't be localized
@EXPORT = qw(*FOO); # Can be localized
```

You can work around the first case by using the fully qualified name (`$Package::FOO`) where you need a local value, or by overriding it by saying `*FOO = *Package::FOO` in your script.

The `*x = \$y` mechanism may be used to pass and return cheap references into or from subroutines if you don't want to copy the whole thing. It only works when assigning to dynamic variables, not lexicals.

```
%some_hash = ();                    # can't be my()
*some_hash = fn( \%another_hash );
sub fn {
    local *hashsym = shift;
    # now use %hashsym normally, and you
    # will affect the caller's %another_hash
    my %nhash = (); # do what you want
    return \%nhash;
}
```

On return, the reference will overwrite the hash slot in the symbol table specified by the *some_hash typeglob. This is a somewhat tricky way of passing around references cheaply when you don't want to have to remember to dereference variables explicitly.

Another use of symbol tables is for making "constant" scalars.

```
*PI = \3.14159265358979;
```

Now you cannot alter $PI, which is probably a good thing all in all. This isn't the same as a constant subroutine, which is subject to optimization at compile-time. A constant subroutine is one prototyped to take no arguments and to return a constant expression. See Section 73.1 [perlsub NAME], page 1216 for details on these. The `use constant` pragma is a convenient shorthand for these.

You can say `*foo{PACKAGE}` and `*foo{NAME}` to find out what name and package the *foo symbol table entry comes from. This may be useful in a subroutine that gets passed typeglobs as arguments:

```
sub identify_typeglob {
    my $glob = shift;
    print 'You gave me ', *{$glob}{PACKAGE},
        '::', *{$glob}{NAME}, "\n";
}
identify_typeglob *foo;
identify_typeglob *bar::baz;
```

This prints

```
You gave me main::foo
You gave me bar::baz
```

The `*foo{THING}` notation can also be used to obtain references to the individual elements of *foo. See Section 62.1 [perlref NAME], page 1077.

Subroutine definitions (and declarations, for that matter) need not necessarily be situated in the package whose symbol table they occupy. You can define a subroutine outside its package by explicitly qualifying the name of the subroutine:

```
package main;
sub Some_package::foo { ... }   # &foo defined in Some_package
```

This is just a shorthand for a typeglob assignment at compile time:

```
BEGIN { *Some_package::foo = sub { ... } }
```

and is *not* the same as writing:

```
{
    package Some_package;
    sub foo { ... }
}
```

In the first two versions, the body of the subroutine is lexically in the main package, *not* in Some_package. So something like this:

```
package main;

$Some_package::name = "fred";
$main::name = "barney";

sub Some_package::foo {
    print "in ", __PACKAGE__, ": \$name is '$name'\n";
}
```

```
Some_package::foo();
```

prints:

```
in main: $name is 'barney'
```

rather than:

```
in Some_package: $name is 'fred'
```

This also has implications for the use of the SUPER:: qualifier (see Section 46.1 [perlobj NAME], page 769).

40.2.4 BEGIN, UNITCHECK, CHECK, INIT and END

Five specially named code blocks are executed at the beginning and at the end of a running Perl program. These are the `BEGIN`, `UNITCHECK`, `CHECK`, `INIT`, and `END` blocks.

These code blocks can be prefixed with `sub` to give the appearance of a subroutine (although this is not considered good style). One should note that these code blocks don't really exist as named subroutines (despite their appearance). The thing that gives this away is the fact that you can have **more than one** of these code blocks in a program, and they will get **all** executed at the appropriate moment. So you can't execute any of these code blocks by name.

A `BEGIN` code block is executed as soon as possible, that is, the moment it is completely defined, even before the rest of the containing file (or string) is parsed. You may have multiple `BEGIN` blocks within a file (or eval'ed string); they will execute in order of definition. Because a `BEGIN` code block executes immediately, it can pull in definitions of subroutines and such from other files in time to be visible to the rest of the compile and run time. Once a `BEGIN` has run, it is immediately undefined and any code it used is returned to Perl's memory pool.

An `END` code block is executed as late as possible, that is, after perl has finished running the program and just before the interpreter is being exited, even if it is exiting as a result of a die() function. (But not if it's morphing into another program via `exec`, or being blown out of the water by a signal–you have to trap that yourself (if you can).) You may have multiple `END` blocks within a file–they will execute in reverse order of definition; that is: last in, first out (LIFO). `END` blocks are not executed when you run perl with the `-c` switch, or if compilation fails.

Note that `END` code blocks are **not** executed at the end of a string `eval()`: if any `END` code blocks are created in a string `eval()`, they will be executed just as any other `END` code block of that package in LIFO order just before the interpreter is being exited.

Inside an `END` code block, `$?` contains the value that the program is going to pass to `exit()`. You can modify `$?` to change the exit value of the program. Beware of changing `$?` by accident (e.g. by running something via `system`).

Inside of a `END` block, the value of `${^GLOBAL_PHASE}` will be `"END"`.

`UNITCHECK`, `CHECK` and `INIT` code blocks are useful to catch the transition between the compilation phase and the execution phase of the main program.

`UNITCHECK` blocks are run just after the unit which defined them has been compiled. The main program file and each module it loads are compilation units, as are string `evals`, run-time code compiled using the `(?{ })` construct in a regex, calls to `do FILE`, `require FILE`, and code after the `-e` switch on the command line.

`BEGIN` and `UNITCHECK` blocks are not directly related to the phase of the interpreter. They can be created and executed during any phase.

`CHECK` code blocks are run just after the **initial** Perl compile phase ends and before the run time begins, in LIFO order. `CHECK` code blocks are used in the Perl compiler suite to save the compiled state of the program.

Inside of a `CHECK` block, the value of `${^GLOBAL_PHASE}` will be `"CHECK"`.

`INIT` blocks are run just before the Perl runtime begins execution, in "first in, first out" (FIFO) order.

Inside of an `INIT` block, the value of `${^GLOBAL_PHASE}` will be `"INIT"`.

The `CHECK` and `INIT` blocks in code compiled by `require`, string `do`, or string `eval` will not be executed if they occur after the end of the main compilation phase; that can be a problem in mod_perl and other persistent environments which use those functions to load code at runtime.

When you use the **-n** and **-p** switches to Perl, `BEGIN` and `END` work just as they do in **awk**, as a degenerate case. Both `BEGIN` and `CHECK` blocks are run when you use the **-c** switch for a compile-only syntax check, although your main code is not.

The **begincheck** program makes it all clear, eventually:

```
#!/usr/bin/perl

# begincheck

print         "10. Ordinary code runs at runtime.\n";

END { print   "16.   So this is the end of the tale.\n" }
INIT { print  " 7. INIT blocks run FIFO just before runtime.\n" }
UNITCHECK {
  print       " 4.   And therefore before any CHECK blocks.\n"
}
CHECK { print " 6.   So this is the sixth line.\n" }

print         "11.   It runs in order, of course.\n";

BEGIN { print " 1. BEGIN blocks run FIFO during compilation.\n" }
END { print   "15.   Read perlmod for the rest of the story.\n" }
CHECK { print " 5. CHECK blocks run LIFO after all compilation.\n" }
INIT { print  " 8.   Run this again, using Perl’s -c switch.\n" }

print         "12.   This is anti-obfuscated code.\n";

END { print   "14. END blocks run LIFO at quitting time.\n" }
BEGIN { print " 2.   So this line comes out second.\n" }
UNITCHECK {
 print " 3. UNITCHECK blocks run LIFO after each file is compiled.\n"
}
INIT { print  " 9.   You’ll see the difference right away.\n" }
```

```
print          "13.   It only _looks_ like it should be confusing.\n";

__END__
```

40.2.5 Perl Classes

There is no special class syntax in Perl, but a package may act as a class if it provides subroutines to act as methods. Such a package may also derive some of its methods from another class (package) by listing the other package name(s) in its global @ISA array (which must be a package global, not a lexical).

For more on this, see Section 47.1 [perlootut NAME], page 786 and Section 46.1 [perlobj NAME], page 769.

40.2.6 Perl Modules

A module is just a set of related functions in a library file, i.e., a Perl package with the same name as the file. It is specifically designed to be reusable by other modules or programs. It may do this by providing a mechanism for exporting some of its symbols into the symbol table of any package using it, or it may function as a class definition and make its semantics available implicitly through method calls on the class and its objects, without explicitly exporting anything. Or it can do a little of both.

For example, to start a traditional, non-OO module called Some::Module, create a file called `Some/Module.pm` and start with this template:

```
package Some::Module;  # assumes Some/Module.pm

use strict;
use warnings;

BEGIN {
    require Exporter;

    # set the version for version checking
    our $VERSION     = 1.00;

    # Inherit from Exporter to export functions and variables
    our @ISA         = qw(Exporter);

    # Functions and variables which are exported by default
    our @EXPORT      = qw(func1 func2);

    # Functions and variables which can be optionally exported
    our @EXPORT_OK   = qw($Var1 %Hashit func3);
}

# exported package globals go here
our $Var1    = '';
our %Hashit  = ();
```

```
# non-exported package globals go here
# (they are still accessible as $Some::Module::stuff)
our @more    = ();
our $stuff   = '';

# file-private lexicals go here, before any functions which use them
my $priv_var    = '';
my %secret_hash = ();

# here's a file-private function as a closure,
# callable as $priv_func->();
my $priv_func = sub {
    ...
};

# make all your functions, whether exported or not;
# remember to put something interesting in the {} stubs
sub func1      { ... }
sub func2      { ... }

# this one isn't exported, but could be called directly
# as Some::Module::func3()
sub func3      { ... }

END { ... }       # module clean-up code here (global destructor)

1;  # don't forget to return a true value from the file
```

Then go on to declare and use your variables in functions without any qualifications. See `Exporter` and the `perlmodlib` for details on mechanics and style issues in module creation.

Perl modules are included into your program by saying

```
use Module;
```

or

```
use Module LIST;
```

This is exactly equivalent to

```
BEGIN { require 'Module.pm'; 'Module'->import; }
```

or

```
BEGIN { require 'Module.pm'; 'Module'->import( LIST ); }
```

As a special case

```
use Module ();
```

is exactly equivalent to

```
BEGIN { require 'Module.pm'; }
```

All Perl module files have the extension `.pm`. The `use` operator assumes this so you don't have to spell out `"Module.pm"` in quotes. This also helps to differentiate new modules from old `.pl` and `.ph` files. Module names are also capitalized unless they're functioning as pragmas; pragmas are in effect compiler directives, and are sometimes called "pragmatic modules" (or even "pragmata" if you're a classicist).

The two statements:

```
require SomeModule;
require "SomeModule.pm";
```

differ from each other in two ways. In the first case, any double colons in the module name, such as `Some::Module`, are translated into your system's directory separator, usually "/". The second case does not, and would have to be specified literally. The other difference is that seeing the first `require` clues in the compiler that uses of indirect object notation involving "SomeModule", as in `$ob = purge SomeModule`, are method calls, not function calls. (Yes, this really can make a difference.)

Because the `use` statement implies a `BEGIN` block, the importing of semantics happens as soon as the `use` statement is compiled, before the rest of the file is compiled. This is how it is able to function as a pragma mechanism, and also how modules are able to declare subroutines that are then visible as list or unary operators for the rest of the current file. This will not work if you use `require` instead of `use`. With `require` you can get into this problem:

```
require Cwd;                # make Cwd:: accessible
$here = Cwd::getcwd();

use Cwd;                    # import names from Cwd::
$here = getcwd();

require Cwd;                # make Cwd:: accessible
$here = getcwd();           # oops! no main::getcwd()
```

In general, `use Module ()` is recommended over `require Module`, because it determines module availability at compile time, not in the middle of your program's execution. An exception would be if two modules each tried to `use` each other, and each also called a function from that other module. In that case, it's easy to use `require` instead.

Perl packages may be nested inside other package names, so we can have package names containing `::`. But if we used that package name directly as a filename it would make for unwieldy or impossible filenames on some systems. Therefore, if a module's name is, say, `Text::Soundex`, then its definition is actually found in the library file `Text/Soundex.pm`.

Perl modules always have a `.pm` file, but there may also be dynamically linked executables (often ending in `.so`) or autoloaded subroutine definitions (often ending in `.al`) associated with the module. If so, these will be entirely transparent to the user of the module. It is the responsibility of the `.pm` file to load (or arrange to autoload) any additional functionality. For example, although the POSIX module happens to do both dynamic loading and autoloading, the user can say just `use POSIX` to get it all.

40.2.7 Making your module threadsafe

Perl supports a type of threads called interpreter threads (ithreads). These threads can be used explicitly and implicitly.

Ithreads work by cloning the data tree so that no data is shared between different threads. These threads can be used by using the `threads` module or by doing fork() on win32 (fake fork() support). When a thread is cloned all Perl data is cloned, however non-Perl data cannot be cloned automatically. Perl after 5.8.0 has support for the `CLONE` special subroutine. In `CLONE` you can do whatever you need to do, like for example handle the cloning of non-Perl data, if necessary. `CLONE` will be called once as a class method for every package that has it defined (or inherits it). It will be called in the context of the new thread, so all modifications are made in the new area. Currently CLONE is called with no parameters other than the invocant package name, but code should not assume that this will remain unchanged, as it is likely that in future extra parameters will be passed in to give more information about the state of cloning.

If you want to CLONE all objects you will need to keep track of them per package. This is simply done using a hash and Scalar::Util::weaken().

Perl after 5.8.7 has support for the `CLONE_SKIP` special subroutine. Like `CLONE`, `CLONE_SKIP` is called once per package; however, it is called just before cloning starts, and in the context of the parent thread. If it returns a true value, then no objects of that class will be cloned; or rather, they will be copied as unblessed, undef values. For example: if in the parent there are two references to a single blessed hash, then in the child there will be two references to a single undefined scalar value instead. This provides a simple mechanism for making a module threadsafe; just add `sub CLONE_SKIP { 1 }` at the top of the class, and `DESTROY()` will now only be called once per object. Of course, if the child thread needs to make use of the objects, then a more sophisticated approach is needed.

Like `CLONE`, `CLONE_SKIP` is currently called with no parameters other than the invocant package name, although that may change. Similarly, to allow for future expansion, the return value should be a single `0` or `1` value.

40.3 SEE ALSO

See `perlmodlib` for general style issues related to building Perl modules and classes, as well as descriptions of the standard library and CPAN, `Exporter` for how Perl's standard import/export mechanism works, Section 47.1 [perlootut NAME], page 786 and Section 46.1 [perlobj NAME], page 769 for in-depth information on creating classes, Section 46.1 [perlobj NAME], page 769 for a hard-core reference document on objects, Section 73.1 [perlsub NAME], page 1216 for an explanation of functions and scoping, and `perlxstut` and Section 28.1 [perlguts NAME], page 512 for more information on writing extension modules.

41 perlmodinstall

41.1 NAME

perlmodinstall - Installing CPAN Modules

41.2 DESCRIPTION

You can think of a module as the fundamental unit of reusable Perl code; see Section 40.1 [perlmod NAME], page 732 for details. Whenever anyone creates a chunk of Perl code that they think will be useful to the world, they register as a Perl developer at http://www.cpan.org/modules/04pause.html so that they can then upload their code to the CPAN. The CPAN is the Comprehensive Perl Archive Network and can be accessed at http://www.cpan.org/ , and searched at http://search.cpan.org/ .

This documentation is for people who want to download CPAN modules and install them on their own computer.

41.2.1 PREAMBLE

First, are you sure that the module isn't already on your system? Try `perl -MFoo -e 1`. (Replace "Foo" with the name of the module; for instance, `perl -MCGI::Carp -e 1`.)

If you don't see an error message, you have the module. (If you do see an error message, it's still possible you have the module, but that it's not in your path, which you can display with `perl -e "print qq(@INC)"`.) For the remainder of this document, we'll assume that you really honestly truly lack an installed module, but have found it on the CPAN.

So now you have a file ending in .tar.gz (or, less often, .zip). You know there's a tasty module inside. There are four steps you must now take:

DECOMPRESS the file
UNPACK the file into a directory
BUILD the module (sometimes unnecessary)
INSTALL the module.

Here's how to perform each step for each operating system. This is <not> a substitute for reading the README and INSTALL files that might have come with your module!

Also note that these instructions are tailored for installing the module into your system's repository of Perl modules, but you can install modules into any directory you wish. For instance, where I say `perl Makefile.PL`, you can substitute `perl Makefile.PL PREFIX=/my/perl_directory` to install the modules into `/my/perl_directory`. Then you can use the modules from your Perl programs with `use lib "/my/perl_directory/lib/site_perl";` or sometimes just `use "/my/perl_directory";`. If you're on a system that requires superuser/root access to install modules into the directories you see when you type `perl -e "print qq(@INC)"`, you'll want to install them into a local directory (such as your home directory) and use this approach.

- **If you're on a Unix or Unix-like system,**

  You can use Andreas Koenig's CPAN module (http://www.cpan.org/modules/by-module/CPAN) to automate the following steps, from DECOMPRESS through INSTALL.

A. DECOMPRESS

Decompress the file with `gzip -d yourmodule.tar.gz`

You can get gzip from ftp://prep.ai.mit.edu/pub/gnu/

Or, you can combine this step with the next to save disk space:

```
     gzip -dc yourmodule.tar.gz | tar -xof -
```

B. UNPACK

Unpack the result with `tar -xof yourmodule.tar`

C. BUILD

Go into the newly-created directory and type:

```
      perl Makefile.PL
      make test
```

or

```
      perl Makefile.PL PREFIX=/my/perl_directory
```

to install it locally. (Remember that if you do this, you'll have to put `use lib "/my/perl_directory";` near the top of the program that is to use this module.

D. INSTALL

While still in that directory, type:

```
      make install
```

Make sure you have the appropriate permissions to install the module in your Perl 5 library directory. Often, you'll need to be root.

That's all you need to do on Unix systems with dynamic linking. Most Unix systems have dynamic linking. If yours doesn't, or if for another reason you have a statically-linked perl, **and** the module requires compilation, you'll need to build a new Perl binary that includes the module. Again, you'll probably need to be root.

- **If you're running ActivePerl (Win95/98/2K/NT/XP, Linux, Solaris),**

  First, type `ppm` from a shell and see whether ActiveState's PPM repository has your module. If so, you can install it with `ppm` and you won't have to bother with any of the other steps here. You might be able to use the CPAN instructions from the "Unix or Linux" section above as well; give it a try. Otherwise, you'll have to follow the steps below.

  `A. DECOMPRESS`

  You can use the shareware Winzip (http://www.winzip.com) to decompress and unpack modules.

  `B. UNPACK`

  If you used WinZip, this was already done for you.

  `C. BUILD`

  You'll need the `nmake` utility, available at http://download.microsoft.com/download/vc15/Patch/1.52/W95
  US/nmake15.exe or dmake, available on CPAN. http://search.cpan.org/dist/dmake/

  Does the module require compilation (i.e. does it have files that end in .xs, .c, .h, .y, .cc, .cxx, or .C)? If it does, life is now officially tough for you, because you have to compile the module yourself (no easy feat on Windows). You'll need a compiler such as Visual C++. Alternatively, you can download a pre-built PPM package from ActiveState. http://aspn.activestate.com/ASPN/Downloads/ActivePerl/PPM/

Go into the newly-created directory and type:

```
perl Makefile.PL
nmake test
```

```
D. INSTALL
```

While still in that directory, type:

```
nmake install
```

- **If you're using a Macintosh with "Classic" MacOS and MacPerl,**

  A. DECOMPRESS

  First, make sure you have the latest **cpan-mac** distribution (http://www.cpan.org/authors/id/CNANDOR), which has utilities for doing all of the steps. Read the cpan-mac directions carefully and install it. If you choose not to use cpan-mac for some reason, there are alternatives listed here.

  After installing cpan-mac, drop the module archive on the **untarzipme** droplet, which will decompress and unpack for you.

  Or, you can either use the shareware **StuffIt Expander** program (http://my.smithmicro.com/mac/stuffit/) or the freeware **MacGzip** program (http://persephone.cps.unizar.es/general/gente/spd/gzip/gzip.html).

  B. UNPACK

  If you're using untarzipme or StuffIt, the archive should be extracted now. **Or**, you can use the freeware **suntar** or *Tar* (http://hyperarchive.lcs.mit.edu/HyperArchive/Archive/cmp/).

  C. BUILD

  Check the contents of the distribution. Read the module's documentation, looking for reasons why you might have trouble using it with MacPerl. Look for `.xs` and `.c` files, which normally denote that the distribution must be compiled, and you cannot install it "out of the box." (See Section 41.3 [PORTABILITY], page 746.)

  D. INSTALL

  If you are using cpan-mac, just drop the folder on the **installme** droplet, and use the module.

  Or, if you aren't using cpan-mac, do some manual labor.

  Make sure the newlines for the modules are in Mac format, not Unix format. If they are not then you might have decompressed them incorrectly. Check your decompression and unpacking utilities settings to make sure they are translating text files properly.

  As a last resort, you can use the perl one-liner:

  ```
  perl -i.bak -pe 's/(?:\015)?\012/\015/g' <filenames>
  ```

  on the source files.

  Then move the files (probably just the `.pm` files, though there may be some additional ones, too; check the module documentation) to their final destination: This will most likely be in `$ENV{MACPERL}site_lib:` (i.e., `HD:MacPerl folder:site_lib:`). You can add new paths to the default `@INC` in the Preferences menu item in the MacPerl application (`$ENV{MACPERL}site_lib:` is added automagically). Create whatever directory structures are required (i.e., for `Some::Module`, create `$ENV{MACPERL}site_lib:Some:` and put `Module.pm` in that directory).

Then run the following script (or something like it):

```
#!perl -w
use AutoSplit;
my $dir = "${MACPERL}site_perl";
autosplit("$dir:Some:Module.pm", "$dir:auto", 0, 1, 1);
```

- **If you're on the DJGPP port of DOS,**

  ```
     A. DECOMPRESS
  ```

  djtarx (ftp://ftp.delorie.com/pub/djgpp/current/v2/) will both uncompress and unpack.

  ```
     B. UNPACK
  ```

  See above.

  ```
     C. BUILD
  ```

  Go into the newly-created directory and type:

  ```
      perl Makefile.PL
      make test
  ```

  You will need the packages mentioned in `README.dos` in the Perl distribution.

  ```
     D. INSTALL
  ```

  While still in that directory, type:

  ```
      make install
  ```

  You will need the packages mentioned in `README.dos` in the Perl distribution.

- **If you're on OS/2,**

  Get the EMX development suite and gzip/tar, from either Hobbes (http://hobbes.nmsu.edu) or Leo (http://www.leo.org), and then follow the instructions for Unix.

- **If you're on VMS,**

  When downloading from CPAN, save your file with a `.tgz` extension instead of `.tar.gz`. All other periods in the filename should be replaced with underscores. For example, `Your-Module-1.33.tar.gz` should be downloaded as `Your-Module-1_33.tgz`.

  A. DECOMPRESS

  Type

  ```
  gzip -d Your-Module.tgz
  ```

  or, for zipped modules, type

  ```
  unzip Your-Module.zip
  ```

  Executables for gzip, zip, and VMStar:

  ```
  http://www.hp.com/go/openvms/freeware/
  ```

  and their source code:

```
http://www.fsf.org/order/ftp.html
```

Note that GNU's gzip/gunzip is not the same as Info-ZIP's zip/unzip package. The former is a simple compression tool; the latter permits creation of multi-file archives.

B. UNPACK

If you're using VMStar:

```
VMStar xf Your-Module.tar
```

Or, if you're fond of VMS command syntax:

```
tar/extract/verbose Your_Module.tar
```

C. BUILD

Make sure you have MMS (from Digital) or the freeware MMK (available from MadGoat at http://www.madgoat.com). Then type this to create the DESCRIP.MMS for the module:

```
perl Makefile.PL
```

Now you're ready to build:

```
mms test
```

Substitute `mmk` for `mms` above if you're using MMK.

D. INSTALL

Type

```
mms install
```

Substitute `mmk` for `mms` above if you're using MMK.

- **If you're on MVS**,

  Introduce the `.tar.gz` file into an HFS as binary; don't translate from ASCII to EBCDIC.

  A. DECOMPRESS

  Decompress the file with `gzip -d yourmodule.tar.gz`

  You can get gzip from http://www.s390.ibm.com/products/oe/bpxqp1.html

  B. UNPACK

  Unpack the result with

  ```
  pax -o to=IBM-1047,from=ISO8859-1 -r < yourmodule.tar
  ```

  The BUILD and INSTALL steps are identical to those for Unix. Some modules generate Makefiles that work better with GNU make, which is available from http://www.mks.com/s390/gnu/

41.3 PORTABILITY

Note that not all modules will work with on all platforms. See Section 56.1 [perlport NAME], page 951 for more information on portability issues. Read the documentation to see if the module will work on your system. There are basically three categories of modules that will not work "out of the box" with all platforms (with some possibility of overlap):

- **Those that should, but don't.** These need to be fixed; consider contacting the author and possibly writing a patch.

- **Those that need to be compiled, where the target platform doesn't have compilers readily available.** (These modules contain `.xs` or `.c` files, usually.) You might be able to find existing binaries on the CPAN or elsewhere, or you might want to try getting compilers and building it yourself, and then release the binary for other poor souls to use.
- **Those that are targeted at a specific platform.** (Such as the Win32:: modules.) If the module is targeted specifically at a platform other than yours, you're out of luck, most likely.

Check the CPAN Testers if a module should work with your platform but it doesn't behave as you'd expect, or you aren't sure whether or not a module will work under your platform. If the module you want isn't listed there, you can test it yourself and let CPAN Testers know, you can join CPAN Testers, or you can request it be tested.

```
http://testers.cpan.org/
```

41.4 HEY

If you have any suggested changes for this page, let me know. Please don't send me mail asking for help on how to install your modules. There are too many modules, and too few Orwants, for me to be able to answer or even acknowledge all your questions. Contact the module author instead, or post to comp.lang.perl.modules, or ask someone familiar with Perl on your operating system.

41.5 AUTHOR

Jon Orwant

orwant@medita.mit.edu

with invaluable help from Chris Nandor, and valuable help from Brandon Allbery, Charles Bailey, Graham Barr, Dominic Dunlop, Jarkko Hietaniemi, Ben Holzman, Tom Horsley, Nick Ing-Simmons, Tuomas J. Lukka, Laszlo Molnar, Alan Olsen, Peter Prymmer, Gurusamy Sarathy, Christoph Spalinger, Dan Sugalski, Larry Virden, and Ilya Zakharevich.

First version July 22, 1998; last revised November 21, 2001.

41.6 COPYRIGHT

42 perlmodstyle

42.1 NAME

perlmodstyle - Perl module style guide

42.2 INTRODUCTION

This document attempts to describe the Perl Community's "best practice" for writing Perl modules. It extends the recommendations found in Section 72.1 [perlstyle NAME], page 1212 , which should be considered required reading before reading this document.

While this document is intended to be useful to all module authors, it is particularly aimed at authors who wish to publish their modules on CPAN.

The focus is on elements of style which are visible to the users of a module, rather than those parts which are only seen by the module's developers. However, many of the guidelines presented in this document can be extrapolated and applied successfully to a module's internals.

This document differs from Section 44.1 [perlnewmod NAME], page 760 in that it is a style guide rather than a tutorial on creating CPAN modules. It provides a checklist against which modules can be compared to determine whether they conform to best practice, without necessarily describing in detail how to achieve this.

All the advice contained in this document has been gleaned from extensive conversations with experienced CPAN authors and users. Every piece of advice given here is the result of previous mistakes. This information is here to help you avoid the same mistakes and the extra work that would inevitably be required to fix them.

The first section of this document provides an itemized checklist; subsequent sections provide a more detailed discussion of the items on the list. The final section, "Common Pitfalls", describes some of the most popular mistakes made by CPAN authors.

42.3 QUICK CHECKLIST

For more detail on each item in this checklist, see below.

42.3.1 Before you start

- Don't re-invent the wheel
- Patch, extend or subclass an existing module where possible
- Do one thing and do it well
- Choose an appropriate name
- Get feedback before publishing

42.3.2 The API

- API should be understandable by the average programmer
- Simple methods for simple tasks
- Separate functionality from output

- Consistent naming of subroutines or methods
- Use named parameters (a hash or hashref) when there are more than two parameters

42.3.3 Stability

- Ensure your module works under `use strict` and `-w`
- Stable modules should maintain backwards compatibility

42.3.4 Documentation

- Write documentation in POD
- Document purpose, scope and target applications
- Document each publically accessible method or subroutine, including params and return values
- Give examples of use in your documentation
- Provide a README file and perhaps also release notes, changelog, etc
- Provide links to further information (URL, email)

42.3.5 Release considerations

- Specify pre-requisites in Makefile.PL or Build.PL
- Specify Perl version requirements with `use`
- Include tests with your module
- Choose a sensible and consistent version numbering scheme (X.YY is the common Perl module numbering scheme)
- Increment the version number for every change, no matter how small
- Package the module using "make dist"
- Choose an appropriate license (GPL/Artistic is a good default)

42.4 BEFORE YOU START WRITING A MODULE

Try not to launch headlong into developing your module without spending some time thinking first. A little forethought may save you a vast amount of effort later on.

42.4.1 Has it been done before?

You may not even need to write the module. Check whether it's already been done in Perl, and avoid re-inventing the wheel unless you have a good reason.

Good places to look for pre-existing modules include `http://search.cpan.org/` and `https://metacpan.org` and asking on `module-authors@perl.org` (`http://lists.perl.org/list/module-authors.html`).

If an existing module **almost** does what you want, consider writing a patch, writing a subclass, or otherwise extending the existing module rather than rewriting it.

42.4.2 Do one thing and do it well

At the risk of stating the obvious, modules are intended to be modular. A Perl developer should be able to use modules to put together the building blocks of their application.

However, it's important that the blocks are the right shape, and that the developer shouldn't have to use a big block when all they need is a small one.

Your module should have a clearly defined scope which is no longer than a single sentence. Can your module be broken down into a family of related modules?

Bad example:

"FooBar.pm provides an implementation of the FOO protocol and the related BAR standard."

Good example:

"Foo.pm provides an implementation of the FOO protocol. Bar.pm implements the related BAR protocol."

This means that if a developer only needs a module for the BAR standard, they should not be forced to install libraries for FOO as well.

42.4.3 What's in a name?

Make sure you choose an appropriate name for your module early on. This will help people find and remember your module, and make programming with your module more intuitive.

When naming your module, consider the following:

- Be descriptive (i.e. accurately describes the purpose of the module).
- Be consistent with existing modules.
- Reflect the functionality of the module, not the implementation.
- Avoid starting a new top-level hierarchy, especially if a suitable hierarchy already exists under which you could place your module.

42.4.4 Get feedback before publishing

If you have never uploaded a module to CPAN before (and even if you have), you are strongly encouraged to get feedback on PrePAN (`http://prepan.org`). PrePAN is a site dedicated to discussing ideas for CPAN modules with other Perl developers and is a great resource for new (and experienced) Perl developers.

You should also try to get feedback from people who are already familiar with the module's application domain and the CPAN naming system. Authors of similar modules, or modules with similar names, may be a good place to start, as are community sites like Perl Monks (`http://www.perlmonks.org`).

42.5 DESIGNING AND WRITING YOUR MODULE

Considerations for module design and coding:

42.5.1 To OO or not to OO?

Your module may be object oriented (OO) or not, or it may have both kinds of interfaces available. There are pros and cons of each technique, which should be considered when you design your API.

In *Perl Best Practices* (copyright 2004, Published by O'Reilly Media, Inc.), Damian Conway provides a list of criteria to use when deciding if OO is the right fit for your problem:

- The system being designed is large, or is likely to become large.
- The data can be aggregated into obvious structures, especially if there's a large amount of data in each aggregate.
- The various types of data aggregate form a natural hierarchy that facilitates the use of inheritance and polymorphism.
- You have a piece of data on which many different operations are applied.
- You need to perform the same general operations on related types of data, but with slight variations depending on the specific type of data the operations are applied to.
- It's likely you'll have to add new data types later.
- The typical interactions between pieces of data are best represented by operators.
- The implementation of individual components of the system is likely to change over time.
- The system design is already object-oriented.
- Large numbers of other programmers will be using your code modules.

Think carefully about whether OO is appropriate for your module. Gratuitous object orientation results in complex APIs which are difficult for the average module user to understand or use.

42.5.2 Designing your API

Your interfaces should be understandable by an average Perl programmer. The following guidelines may help you judge whether your API is sufficiently straightforward:

Write simple routines to do simple things.

It's better to have numerous simple routines than a few monolithic ones. If your routine changes its behaviour significantly based on its arguments, it's a sign that you should have two (or more) separate routines.

Separate functionality from output.

Return your results in the most generic form possible and allow the user to choose how to use them. The most generic form possible is usually a Perl data structure which can then be used to generate a text report, HTML, XML, a database query, or whatever else your users require.

If your routine iterates through some kind of list (such as a list of files, or records in a database) you may consider providing a callback so that users can manipulate each element of the list in turn. File::Find provides an example of this with its `find(\&wanted, $dir)` syntax.

Provide sensible shortcuts and defaults.

Don't require every module user to jump through the same hoops to achieve a simple result. You can always include optional parameters or routines for more complex or non-standard behaviour. If most of your users have to type a few almost identical lines of code when they start using your module, it's a sign that you should have made that behaviour a default. Another good indicator that you should use defaults is if most of your users call your routines with the same arguments.

Naming conventions

Your naming should be consistent. For instance, it's better to have:

```
display_day();
display_week();
display_year();
```

than

```
display_day();
week_display();
show_year();
```

This applies equally to method names, parameter names, and anything else which is visible to the user (and most things that aren't!)

Parameter passing

Use named parameters. It's easier to use a hash like this:

```
$obj->do_something(
        name => "wibble",
        type => "text",
        size => 1024,
);
```

... than to have a long list of unnamed parameters like this:

```
$obj->do_something("wibble", "text", 1024);
```

While the list of arguments might work fine for one, two or even three arguments, any more arguments become hard for the module user to remember, and hard for the module author to manage. If you want to add a new parameter you will have to add it to the end of the list for backward compatibility, and this will probably make your list order unintuitive. Also, if many elements may be undefined you may see the following unattractive method calls:

```
$obj->do_something(undef, undef, undef, undef, undef, 1024);
```

Provide sensible defaults for parameters which have them. Don't make your users specify parameters which will almost always be the same.

The issue of whether to pass the arguments in a hash or a hashref is largely a matter of personal style.

The use of hash keys starting with a hyphen (`-name`) or entirely in upper case (`NAME`) is a relic of older versions of Perl in which ordinary lower case strings were not handled correctly by the `=>` operator. While some modules retain uppercase or hyphenated argument keys for historical reasons or as a matter of personal style, most new modules should use simple lower case keys. Whatever you choose, be consistent!

42.5.3 Strictness and warnings

Your module should run successfully under the strict pragma and should run without generating any warnings. Your module should also handle taint-checking where appropriate, though this can cause difficulties in many cases.

42.5.4 Backwards compatibility

Modules which are "stable" should not break backwards compatibility without at least a long transition phase and a major change in version number.

42.5.5 Error handling and messages

When your module encounters an error it should do one or more of:

- Return an undefined value.
- set `$Module::errstr` or similar (`errstr` is a common name used by DBI and other popular modules; if you choose something else, be sure to document it clearly).
- `warn()` or `carp()` a message to STDERR.
- `croak()` only when your module absolutely cannot figure out what to do. (`croak()` is a better version of `die()` for use within modules, which reports its errors from the perspective of the caller. See `Carp` for details of `croak()`, `carp()` and other useful routines.)
- As an alternative to the above, you may prefer to throw exceptions using the Error module.

Configurable error handling can be very useful to your users. Consider offering a choice of levels for warning and debug messages, an option to send messages to a separate file, a way to specify an error-handling routine, or other such features. Be sure to default all these options to the commonest use.

42.6 DOCUMENTING YOUR MODULE

42.6.1 POD

Your module should include documentation aimed at Perl developers. You should use Perl's "plain old documentation" (POD) for your general technical documentation, though you may wish to write additional documentation (white papers, tutorials, etc) in some other format. You need to cover the following subjects:

- A synopsis of the common uses of the module
- The purpose, scope and target applications of your module
- Use of each publically accessible method or subroutine, including parameters and return values
- Examples of use
- Sources of further information
- A contact email address for the author/maintainer

The level of detail in Perl module documentation generally goes from less detailed to more detailed. Your SYNOPSIS section should contain a minimal example of use (perhaps as little as one line of code; skip the unusual use cases or anything not needed by most users); the DESCRIPTION should describe your module in broad terms, generally in just a few paragraphs; more detail of the module's routines or methods, lengthy code examples, or other in-depth material should be given in subsequent sections.

Ideally, someone who's slightly familiar with your module should be able to refresh their memory without hitting "page down". As your reader continues through the document, they should receive a progressively greater amount of knowledge.

The recommended order of sections in Perl module documentation is:

- NAME
- SYNOPSIS
- DESCRIPTION
- One or more sections or subsections giving greater detail of available methods and routines and any other relevant information.
- BUGS/CAVEATS/etc
- AUTHOR
- SEE ALSO
- COPYRIGHT and LICENSE

Keep your documentation near the code it documents ("inline" documentation). Include POD for a given method right above that method's subroutine. This makes it easier to keep the documentation up to date, and avoids having to document each piece of code twice (once in POD and once in comments).

42.6.2 README, INSTALL, release notes, changelogs

Your module should also include a README file describing the module and giving pointers to further information (website, author email).

An INSTALL file should be included, and should contain simple installation instructions. When using ExtUtils::MakeMaker this will usually be:

```
perl Makefile.PL
make

make test

make install
```

When using Module::Build, this will usually be:

```
perl Build.PL
perl Build

perl Build test
perl Build install
```

Release notes or changelogs should be produced for each release of your software describing user-visible changes to your module, in terms relevant to the user.

Unless you have good reasons for using some other format (for example, a format used within your company), the convention is to name your changelog file `Changes`, and to follow the simple format described in `CPAN-Changes-Spec`.

42.7 RELEASE CONSIDERATIONS

42.7.1 Version numbering

Version numbers should indicate at least major and minor releases, and possibly sub-minor releases. A major release is one in which most of the functionality has changed, or in which major new functionality is added. A minor release is one in which a small amount of functionality has been added or changed. Sub-minor version numbers are usually used for changes which do not affect functionality, such as documentation patches.

The most common CPAN version numbering scheme looks like this:

```
  1.00, 1.10, 1.11, 1.20, 1.30, 1.31, 1.32
```

A correct CPAN version number is a floating point number with at least 2 digits after the decimal. You can test whether it conforms to CPAN by using

```
  perl -MExtUtils::MakeMaker -le 'print MM->parse_version(shift)' 'Foo.pm'
```

If you want to release a 'beta' or 'alpha' version of a module but don't want CPAN.pm to list it as most recent use an '_' after the regular version number followed by at least 2 digits, eg. 1.20_01. If you do this, the following idiom is recommended:

```
our $VERSION = "1.12_01"; # so CPAN distribution will have
                          # right filename
our $XS_VERSION = $VERSION; # only needed if you have XS code
$VERSION = eval $VERSION; # so "use Module 0.002" won't warn on
                          # underscore
```

With that trick MakeMaker will only read the first line and thus read the underscore, while the perl interpreter will evaluate the $VERSION and convert the string into a number. Later operations that treat $VERSION as a number will then be able to do so without provoking a warning about $VERSION not being a number.

Never release anything (even a one-word documentation patch) without incrementing the number. Even a one-word documentation patch should result in a change in version at the sub-minor level.

Once picked, it is important to stick to your version scheme, without reducing the number of digits. This is because "downstream" packagers, such as the FreeBSD ports system, interpret the version numbers in various ways. If you change the number of digits in your version scheme, you can confuse these systems so they get the versions of your module out of order, which is obviously bad.

42.7.2 Pre-requisites

Module authors should carefully consider whether to rely on other modules, and which modules to rely on.

Most importantly, choose modules which are as stable as possible. In order of preference:

- Core Perl modules
- Stable CPAN modules
- Unstable CPAN modules
- Modules not available from CPAN

Specify version requirements for other Perl modules in the pre-requisites in your Makefile.PL or Build.PL.

Be sure to specify Perl version requirements both in Makefile.PL or Build.PL and with `require 5.6.1` or similar. See the section on `use VERSION` of [perlfunc require], page 437 for details.

42.7.3 Testing

All modules should be tested before distribution (using "make disttest"), and the tests should also be available to people installing the modules (using "make test"). For Module::Build you would use the `make test` equivalent `perl Build test`.

The importance of these tests is proportional to the alleged stability of a module. A module which purports to be stable or which hopes to achieve wide use should adhere to as strict a testing regime as possible.

Useful modules to help you write tests (with minimum impact on your development process or your time) include Test::Simple, Carp::Assert and Test::Inline. For more sophisticated test suites there are Test::More and Test::MockObject.

42.7.4 Packaging

Modules should be packaged using one of the standard packaging tools. Currently you have the choice between ExtUtils::MakeMaker and the more platform independent Module::Build, allowing modules to be installed in a consistent manner. When using ExtUtils::MakeMaker, you can use "make dist" to create your package. Tools exist to help you to build your module in a MakeMaker-friendly style. These include ExtUtils::ModuleMaker and h2xs. See also Section 44.1 [perlnewmod NAME], page 760.

42.7.5 Licensing

Make sure that your module has a license, and that the full text of it is included in the distribution (unless it's a common one and the terms of the license don't require you to include it).

If you don't know what license to use, dual licensing under the GPL and Artistic licenses (the same as Perl itself) is a good idea. See Section 27.1 [perlgpl NAME], page 506 and Section 3.1 [perlartistic NAME], page 18.

42.8 COMMON PITFALLS

42.8.1 Reinventing the wheel

There are certain application spaces which are already very, very well served by CPAN. One example is templating systems, another is date and time modules, and there are many more. While it is a rite of passage to write your own version of these things, please consider carefully whether the Perl world really needs you to publish it.

42.8.2 Trying to do too much

Your module will be part of a developer's toolkit. It will not, in itself, form the **entire** toolkit. It's tempting to add extra features until your code is a monolithic system rather than a set of modular building blocks.

42.8.3 Inappropriate documentation

Don't fall into the trap of writing for the wrong audience. Your primary audience is a reasonably experienced developer with at least a moderate understanding of your module's application domain, who's just downloaded your module and wants to start using it as quickly as possible.

Tutorials, end-user documentation, research papers, FAQs etc are not appropriate in a module's main documentation. If you really want to write these, include them as sub-documents such as `My::Module::Tutorial` or `My::Module::FAQ` and provide a link in the SEE ALSO section of the main documentation.

42.9 SEE ALSO

Section 72.1 [perlstyle NAME], page 1212
: General Perl style guide

Section 44.1 [perlnewmod NAME], page 760
: How to create a new module

Section 52.1 [perlpod NAME], page 900
: POD documentation

`podchecker`
: Verifies your POD's correctness

Packaging Tools
: `ExtUtils-MakeMaker`, `Module-Build`

Testing tools
: `Test-Simple`, `Test-Inline`, `Carp-Assert`, `Test-More`, `Test-MockObject`

http://pause.perl.org/
: Perl Authors Upload Server. Contains links to information for module authors.

Any good book on software engineering

42.10 AUTHOR

Kirrily "Skud" Robert <skud@cpan.org>

43 perlmroapi

43.1 NAME

perlmroapi - Perl method resolution plugin interface

43.2 DESCRIPTION

As of Perl 5.10.1 there is a new interface for plugging and using method resolution orders other than the default (linear depth first search). The C3 method resolution order added in 5.10.0 has been re-implemented as a plugin, without changing its Perl-space interface.

Each plugin should register itself by providing the following structure

```
struct mro_alg {
    AV *(*resolve)(pTHX_ HV *stash, U32 level);
    const char *name;
    U16 length;
    U16 kflags;
    U32 hash;
};
```

and calling `Perl_mro_register`:

```
Perl_mro_register(aTHX_ &my_mro_alg);
```

resolve

Pointer to the linearisation function, described below.

name

Name of the MRO, either in ISO-8859-1 or UTF-8.

length

Length of the name.

kflags

If the name is given in UTF-8, set this to `HVhek_UTF8`. The value is passed direct as the parameter *kflags* to `hv_common()`.

hash

A precomputed hash value for the MRO's name, or 0.

43.3 Callbacks

The `resolve` function is called to generate a linearised ISA for the given stash, using this MRO. It is called with a pointer to the stash, and a *level* of 0. The core always sets *level* to 0 when it calls your function - the parameter is provided to allow your implementation to track depth if it needs to recurse.

The function should return a reference to an array containing the parent classes in order. The names of the classes should be the result of calling `HvENAME()` on the stash. In those cases where `HvENAME()` returns null, `HvNAME()` should be used instead.

The caller is responsible for incrementing the reference count of the array returned if it wants to keep the structure. Hence, if you have created a temporary value that you keep no pointer to, sv_2mortal() to ensure that it is disposed of correctly. If you have cached your return value, then return a pointer to it without changing the reference count.

43.4 Caching

Computing MROs can be expensive. The implementation provides a cache, in which you can store a single SV *, or anything that can be cast to SV *, such as AV *. To read your private value, use the macro MRO_GET_PRIVATE_DATA(), passing it the mro_meta structure from the stash, and a pointer to your mro_alg structure:

```
meta = HvMROMETA(stash);
private_sv = MRO_GET_PRIVATE_DATA(meta, &my_mro_alg);
```

To set your private value, call Perl_mro_set_private_data():

```
Perl_mro_set_private_data(aTHX_ meta, &c3_alg, private_sv);
```

The private data cache will take ownership of a reference to private_sv, much the same way that hv_store() takes ownership of a reference to the value that you pass it.

43.5 Examples

For examples of MRO implementations, see S_mro_get_linear_isa_c3() and the BOOT: section of mro/mro.xs, and S_mro_get_linear_isa_dfs() in mro.c

43.6 AUTHORS

The implementation of the C3 MRO and switchable MROs within the perl core was written by Brandon L Black. Nicholas Clark created the pluggable interface, refactored Brandon's implementation to work with it, and wrote this document.

44 perlnewmod

44.1 NAME

perlnewmod - preparing a new module for distribution

44.2 DESCRIPTION

This document gives you some suggestions about how to go about writing Perl modules, preparing them for distribution, and making them available via CPAN.

One of the things that makes Perl really powerful is the fact that Perl hackers tend to want to share the solutions to problems they've faced, so you and I don't have to battle with the same problem again.

The main way they do this is by abstracting the solution into a Perl module. If you don't know what one of these is, the rest of this document isn't going to be much use to you. You're also missing out on an awful lot of useful code; consider having a look at Section 40.1 [perlmod NAME], page 732, `perlmodlib` and Section 41.1 [perlmodinstall NAME], page 742 before coming back here.

When you've found that there isn't a module available for what you're trying to do, and you've had to write the code yourself, consider packaging up the solution into a module and uploading it to CPAN so that others can benefit.

You should also take a look at Section 42.1 [perlmodstyle NAME], page 748 for best practices in making a module.

44.2.1 Warning

We're going to primarily concentrate on Perl-only modules here, rather than XS modules. XS modules serve a rather different purpose, and you should consider different things before distributing them - the popularity of the library you are gluing, the portability to other operating systems, and so on. However, the notes on preparing the Perl side of the module and packaging and distributing it will apply equally well to an XS module as a pure-Perl one.

44.2.2 What should I make into a module?

You should make a module out of any code that you think is going to be useful to others. Anything that's likely to fill a hole in the communal library and which someone else can slot directly into their program. Any part of your code which you can isolate and extract and plug into something else is a likely candidate.

Let's take an example. Suppose you're reading in data from a local format into a hash-of-hashes in Perl, turning that into a tree, walking the tree and then piping each node to an Acme Transmogrifier Server.

Now, quite a few people have the Acme Transmogrifier, and you've had to write something to talk the protocol from scratch - you'd almost certainly want to make that into a module. The level at which you pitch it is up to you: you might want protocol-level modules analogous to `Net-SMTP` which then talk to higher level modules analogous to `Mail-Send`. The choice is yours, but you do want to get a module out for that server protocol.

Nobody else on the planet is going to talk your local data format, so we can ignore that. But what about the thing in the middle? Building tree structures from Perl variables and then traversing them is a nice, general problem, and if nobody's already written a module that does that, you might want to modularise that code too.

So hopefully you've now got a few ideas about what's good to modularise. Let's now see how it's done.

44.2.3 Step-by-step: Preparing the ground

Before we even start scraping out the code, there are a few things we'll want to do in advance.

Look around

> Dig into a bunch of modules to see how they're written. I'd suggest starting with `Text-Tabs`, since it's in the standard library and is nice and simple, and then looking at something a little more complex like `File-Copy`. For object oriented code, `WWW::Mechanize` or the `Email::*` modules provide some good examples.
>
> These should give you an overall feel for how modules are laid out and written.

Check it's new

> There are a lot of modules on CPAN, and it's easy to miss one that's similar to what you're planning on contributing. Have a good plough through the `http://search.cpan.org` and make sure you're not the one reinventing the wheel!

Discuss the need

> You might love it. You might feel that everyone else needs it. But there might not actually be any real demand for it out there. If you're unsure about the demand your module will have, consider sending out feelers on the `comp.lang.perl.modules` newsgroup, or as a last resort, ask the modules list at `modules@perl.org`. Remember that this is a closed list with a very long turn-around time - be prepared to wait a good while for a response from them.

Choose a name

> Perl modules included on CPAN have a naming hierarchy you should try to fit in with. See `perlmodlib` for more details on how this works, and browse around CPAN and the modules list to get a feel of it. At the very least, remember this: modules should be title capitalised, (This::Thing) fit in with a category, and explain their purpose succinctly.

Check again

> While you're doing that, make really sure you haven't missed a module similar to the one you're about to write.
>
> When you've got your name sorted out and you're sure that your module is wanted and not currently available, it's time to start coding.

44.2.4 Step-by-step: Making the module

Start with `module-starter` or `h2xs`

The `module-starter` utility is distributed as part of the `Module-Starter` CPAN package. It creates a directory with stubs of all the necessary files to start a new module, according to recent "best practice" for module development, and is invoked from the command line, thus:

```
module-starter --module=Foo::Bar \
   --author="Your Name" --email=yourname@cpan.org
```

If you do not wish to install the `Module-Starter` package from CPAN, `h2xs` is an older tool, originally intended for the development of XS modules, which comes packaged with the Perl distribution.

A typical invocation of `h2xs` for a pure Perl module is:

```
h2xs -AX --skip-exporter --use-new-tests -n Foo::Bar
```

The `-A` omits the Autoloader code, `-X` omits XS elements, `--skip-exporter` omits the Exporter code, `--use-new-tests` sets up a modern testing environment, and `-n` specifies the name of the module.

Use `strict` and `warnings`

A module's code has to be warning and strict-clean, since you can't guarantee the conditions that it'll be used under. Besides, you wouldn't want to distribute code that wasn't warning or strict-clean anyway, right?

Use `Carp`

The `Carp` module allows you to present your error messages from the caller's perspective; this gives you a way to signal a problem with the caller and not your module. For instance, if you say this:

```
warn "No hostname given";
```

the user will see something like this:

```
No hostname given at /usr/local/lib/perl5/site_perl/5.6.0/Net/Acme.pm
line 123.
```

which looks like your module is doing something wrong. Instead, you want to put the blame on the user, and say this:

```
No hostname given at bad_code, line 10.
```

You do this by using `Carp` and replacing your `warn`s with `carp`s. If you need to `die`, say `croak` instead. However, keep `warn` and `die` in place for your sanity checks - where it really is your module at fault.

Use `Exporter` - wisely!

`Exporter` gives you a standard way of exporting symbols and subroutines from your module into the caller's namespace. For instance, saying `use Net::Acme qw(&frob)` would import the `frob` subroutine.

The package variable `@EXPORT` will determine which symbols will get exported when the caller simply says `use Net::Acme` - you will hardly ever want to put anything in there. `@EXPORT_OK`, on the other hand, specifies which symbols you're willing to export. If you do want to export a bunch of symbols, use the

%EXPORT_TAGS and define a standard export set - look at Exporter for more details.

Use Section 52.1 [plain old documentation], page 900

The work isn't over until the paperwork is done, and you're going to need to put in some time writing some documentation for your module. module-starter or h2xs will provide a stub for you to fill in; if you're not sure about the format, look at Section 52.1 [perlpod NAME], page 900 for an introduction. Provide a good synopsis of how your module is used in code, a description, and then notes on the syntax and function of the individual subroutines or methods. Use Perl comments for developer notes and POD for end-user notes.

Write tests

You're encouraged to create self-tests for your module to ensure it's working as intended on the myriad platforms Perl supports; if you upload your module to CPAN, a host of testers will build your module and send you the results of the tests. Again, module-starter and h2xs provide a test framework which you can extend - you should do something more than just checking your module will compile. Test-Simple and Test-More are good places to start when writing a test suite.

Write the README

If you're uploading to CPAN, the automated gremlins will extract the README file and place that in your CPAN directory. It'll also appear in the main by-module and by-category directories if you make it onto the modules list. It's a good idea to put here what the module actually does in detail, and the user-visible changes since the last release.

44.2.5 Step-by-step: Distributing your module

Get a CPAN user ID

Every developer publishing modules on CPAN needs a CPAN ID. Visit http://pause.perl.org/, select "Request PAUSE Account", and wait for your request to be approved by the PAUSE administrators.

`perl Makefile.PL; make test; make dist`

Once again, module-starter or h2xs has done all the work for you. They produce the standard Makefile.PL you see when you download and install modules, and this produces a Makefile with a dist target.

Once you've ensured that your module passes its own tests - always a good thing to make sure - you can make dist, and the Makefile will hopefully produce you a nice tarball of your module, ready for upload.

Upload the tarball

The email you got when you received your CPAN ID will tell you how to log in to PAUSE, the Perl Authors Upload SErver. From the menus there, you can upload your module to CPAN.

Announce to the modules list

Once uploaded, it'll sit unnoticed in your author directory. If you want it connected to the rest of the CPAN, you'll need to go to "Register Namespace" on PAUSE. Once registered, your module will appear in the by-module and by-category listings on CPAN.

Announce to clpa

If you have a burning desire to tell the world about your release, post an announcement to the moderated `comp.lang.perl.announce` newsgroup.

Fix bugs!

Once you start accumulating users, they'll send you bug reports. If you're lucky, they'll even send you patches. Welcome to the joys of maintaining a software project...

44.3 AUTHOR

Simon Cozens, `simon@cpan.org`

Updated by Kirrily "Skud" Robert, `skud@cpan.org`

44.4 SEE ALSO

Section 40.1 [perlmod NAME], page 732, `perlmodlib`, Section 41.1 [perlmodinstall NAME], page 742, `h2xs`, `strict`, `Carp`, `Exporter`, Section 52.1 [perlpod NAME], page 900, `Test-Simple`, `Test-More` `ExtUtils-MakeMaker`, `Module-Build`, `Module-Starter` http://www.cpan.org/ , Ken Williams's tutorial on building your own module at http://mathforum.org/~ken/perl_modules.html

45 perlnumber

45.1 NAME

perlnumber - semantics of numbers and numeric operations in Perl

45.2 SYNOPSIS

```
    $n = 1234;              # decimal integer
    $n = 0b1110011;         # binary integer
    $n = 01234;             # octal integer
    $n = 0x1234;            # hexadecimal integer
    $n = 12.34e-56;         # exponential notation
    $n = "-12.34e56";       # number specified as a string
    $n = "1234";            # number specified as a string
```

45.3 DESCRIPTION

This document describes how Perl internally handles numeric values.

Perl's operator overloading facility is completely ignored here. Operator overloading allows user-defined behaviors for numbers, such as operations over arbitrarily large integers, floating points numbers with arbitrary precision, operations over "exotic" numbers such as modular arithmetic or p-adic arithmetic, and so on. See `overload` for details.

45.4 Storing numbers

Perl can internally represent numbers in 3 different ways: as native integers, as native floating point numbers, and as decimal strings. Decimal strings may have an exponential notation part, as in `"12.34e-56"`. *Native* here means "a format supported by the C compiler which was used to build perl".

The term "native" does not mean quite as much when we talk about native integers, as it does when native floating point numbers are involved. The only implication of the term "native" on integers is that the limits for the maximal and the minimal supported true integral quantities are close to powers of 2. However, "native" floats have a most fundamental restriction: they may represent only those numbers which have a relatively "short" representation when converted to a binary fraction. For example, 0.9 cannot be represented by a native float, since the binary fraction for 0.9 is infinite:

```
  binary0.1110011001100...
```

with the sequence `1100` repeating again and again. In addition to this limitation, the exponent of the binary number is also restricted when it is represented as a floating point number. On typical hardware, floating point values can store numbers with up to 53 binary digits, and with binary exponents between -1024 and 1024. In decimal representation this is close to 16 decimal digits and decimal exponents in the range of -304..304. The upshot of all this is that Perl cannot store a number like 12345678901234567 as a floating point number on such architectures without loss of information.

Similarly, decimal strings can represent only those numbers which have a finite decimal expansion. Being strings, and thus of arbitrary length, there is no practical limit for the exponent or number of decimal digits for these numbers. (But realize that what we are discussing the rules for just the *storage* of these numbers. The fact that you can store such "large" numbers does not mean that the *operations* over these numbers will use all of the significant digits. See Section 45.5 [Numeric operators and numeric conversions], page 766 for details.)

In fact numbers stored in the native integer format may be stored either in the signed native form, or in the unsigned native form. Thus the limits for Perl numbers stored as native integers would typically be -2**31..2**32-1, with appropriate modifications in the case of 64-bit integers. Again, this does not mean that Perl can do operations only over integers in this range: it is possible to store many more integers in floating point format.

Summing up, Perl numeric values can store only those numbers which have a finite decimal expansion or a "short" binary expansion.

45.5 Numeric operators and numeric conversions

As mentioned earlier, Perl can store a number in any one of three formats, but most operators typically understand only one of those formats. When a numeric value is passed as an argument to such an operator, it will be converted to the format understood by the operator.

Six such conversions are possible:

```
native integer        --> native floating point       (*)
native integer        --> decimal string
native floating_point --> native integer              (*)
native floating_point --> decimal string              (*)
decimal string        --> native integer
decimal string        --> native floating point       (*)
```

These conversions are governed by the following general rules:

- If the source number can be represented in the target form, that representation is used.
- If the source number is outside of the limits representable in the target form, a representation of the closest limit is used. (*Loss of information*)
- If the source number is between two numbers representable in the target form, a representation of one of these numbers is used. (*Loss of information*)
- In `native floating point --> native integer` conversions the magnitude of the result is less than or equal to the magnitude of the source. ("*Rounding to zero*".)
- If the `decimal string --> native integer` conversion cannot be done without loss of information, the result is compatible with the conversion sequence `decimal_string --> native_floating_point --> native_integer`. In particular, rounding is strongly biased to 0, though a number like `"0.99999999999999999999"` has a chance of being rounded to 1.

RESTRICTION: The conversions marked with `(*)` above involve steps performed by the C compiler. In particular, bugs/features of the compiler used may lead to breakage of some of the above rules.

45.6 Flavors of Perl numeric operations

Perl operations which take a numeric argument treat that argument in one of four different ways: they may force it to one of the integer/floating/ string formats, or they may behave differently depending on the format of the operand. Forcing a numeric value to a particular format does not change the number stored in the value.

All the operators which need an argument in the integer format treat the argument as in modular arithmetic, e.g., `mod 2**32` on a 32-bit architecture. `sprintf "%u", -1` therefore provides the same result as `sprintf "%u", ~0`.

Arithmetic operators

The binary operators `+ - * / % == != > < >= <=` and the unary operators `- abs` and `--` will attempt to convert arguments to integers. If both conversions are possible without loss of precision, and the operation can be performed without loss of precision then the integer result is used. Otherwise arguments are converted to floating point format and the floating point result is used. The caching of conversions (as described above) means that the integer conversion does not throw away fractional parts on floating point numbers.

`++`

`++` behaves as the other operators above, except that if it is a string matching the format `/^[a-zA-Z]*[0-9]*\z/` the string increment described in Section 48.1 [perlop NAME], page 798 is used.

Arithmetic operators during `use integer`

In scopes where `use integer;` is in force, nearly all the operators listed above will force their argument(s) into integer format, and return an integer result. The exceptions, `abs`, `++` and `--`, do not change their behavior with `use integer;`

Other mathematical operators

Operators such as `**`, `sin` and `exp` force arguments to floating point format.

Bitwise operators

Arguments are forced into the integer format if not strings.

Bitwise operators during `use integer`

forces arguments to integer format. Also shift operations internally use signed integers rather than the default unsigned.

Operators which expect an integer

force the argument into the integer format. This is applicable to the third and fourth arguments of `sysread`, for example.

Operators which expect a string

force the argument into the string format. For example, this is applicable to `printf "%s", $value`.

Though forcing an argument into a particular form does not change the stored number, Perl remembers the result of such conversions. In particular, though the first such conversion may be time-consuming, repeated operations will not need to redo the conversion.

45.7 AUTHOR

Ilya Zakharevich `ilya@math.ohio-state.edu`

Editorial adjustments by Gurusamy Sarathy <gsar@ActiveState.com>

Updates for 5.8.0 by Nicholas Clark <nick@ccl4.org>

45.8 SEE ALSO

`overload`, Section 48.1 [perlop NAME], page 798

46 perlobj

46.1 NAME

perlobj - Perl object reference

46.2 DESCRIPTION

This document provides a reference for Perl's object orientation features. If you're looking for an introduction to object-oriented programming in Perl, please see Section 47.1 [perlootut NAME], page 786.

In order to understand Perl objects, you first need to understand references in Perl. See Section 62.1 [perlref NAME], page 1077 for details.

This document describes all of Perl's object-oriented (OO) features from the ground up. If you're just looking to write some object-oriented code of your own, you are probably better served by using one of the object systems from CPAN described in Section 47.1 [perlootut NAME], page 786.

If you're looking to write your own object system, or you need to maintain code which implements objects from scratch then this document will help you understand exactly how Perl does object orientation.

There are a few basic principles which define object oriented Perl:

1. An object is simply a data structure that knows to which class it belongs.
2. A class is simply a package. A class provides methods that expect to operate on objects.
3. A method is simply a subroutine that expects a reference to an object (or a package name, for class methods) as the first argument.

Let's look at each of these principles in depth.

46.2.1 An Object is Simply a Data Structure

Unlike many other languages which support object orientation, Perl does not provide any special syntax for constructing an object. Objects are merely Perl data structures (hashes, arrays, scalars, filehandles, etc.) that have been explicitly associated with a particular class.

That explicit association is created by the built-in `bless` function, which is typically used within the *constructor* subroutine of the class.

Here is a simple constructor:

```
package File;

sub new {
    my $class = shift;

    return bless {}, $class;
}
```

The name `new` isn't special. We could name our constructor something else:

```
package File;

sub load {
    my $class = shift;

    return bless {}, $class;
}
```

The modern convention for OO modules is to always use `new` as the name for the constructor, but there is no requirement to do so. Any subroutine that blesses a data structure into a class is a valid constructor in Perl.

In the previous examples, the `{}` code creates a reference to an empty anonymous hash. The `bless` function then takes that reference and associates the hash with the class in `$class`. In the simplest case, the `$class` variable will end up containing the string "File".

We can also use a variable to store a reference to the data structure that is being blessed as our object:

```
sub new {
    my $class = shift;

    my $self = {};
    bless $self, $class;

    return $self;
}
```

Once we've blessed the hash referred to by `$self` we can start calling methods on it. This is useful if you want to put object initialization in its own separate method:

```
sub new {
    my $class = shift;

    my $self = {};
    bless $self, $class;

    $self->_initialize();

    return $self;
}
```

Since the object is also a hash, you can treat it as one, using it to store data associated with the object. Typically, code inside the class can treat the hash as an accessible data structure, while code outside the class should always treat the object as opaque. This is called **encapsulation**. Encapsulation means that the user of an object does not have to know how it is implemented. The user simply calls documented methods on the object.

Note, however, that (unlike most other OO languages) Perl does not ensure or enforce encapsulation in any way. If you want objects to actually *be* opaque you need to arrange for that yourself. This can be done in a variety of ways, including using Section 46.2.16 [Inside-Out objects], page 784 or modules from CPAN.

46.2.1.1 Objects Are Blessed; Variables Are Not

When we bless something, we are not blessing the variable which contains a reference to that thing, nor are we blessing the reference that the variable stores; we are blessing the thing that the variable refers to (sometimes known as the *referent*). This is best demonstrated with this code:

```
use Scalar::Util 'blessed';

my $foo = {};
my $bar = $foo;

bless $foo, 'Class';
print blessed( $bar );      # prints "Class"

$bar = "some other value";
print blessed( $bar );      # prints undef
```

When we call `bless` on a variable, we are actually blessing the underlying data structure that the variable refers to. We are not blessing the reference itself, nor the variable that contains that reference. That's why the second call to `blessed( $bar )` returns false. At that point `$bar` is no longer storing a reference to an object.

You will sometimes see older books or documentation mention "blessing a reference" or describe an object as a "blessed reference", but this is incorrect. It isn't the reference that is blessed as an object; it's the thing the reference refers to (i.e. the referent).

46.2.2 A Class is Simply a Package

Perl does not provide any special syntax for class definitions. A package is simply a namespace containing variables and subroutines. The only difference is that in a class, the subroutines may expect a reference to an object or the name of a class as the first argument. This is purely a matter of convention, so a class may contain both methods and subroutines which *don't* operate on an object or class.

Each package contains a special array called `@ISA`. The `@ISA` array contains a list of that class's parent classes, if any. This array is examined when Perl does method resolution, which we will cover later.

It is possible to manually set `@ISA`, and you may see this in older Perl code. Much older code also uses the `base` pragma. For new code, we recommend that you use the `parent` pragma to declare your parents. This pragma will take care of setting `@ISA`. It will also load the parent classes and make sure that the package doesn't inherit from itself.

However the parent classes are set, the package's `@ISA` variable will contain a list of those parents. This is simply a list of scalars, each of which is a string that corresponds to a package name.

All classes inherit from the `UNIVERSAL` class implicitly. The `UNIVERSAL` class is implemented by the Perl core, and provides several default methods, such as `isa()`, `can()`, and `VERSION()`. The `UNIVERSAL` class will *never* appear in a package's `@ISA` variable.

Perl *only* provides method inheritance as a built-in feature. Attribute inheritance is left up the class to implement. See the Section 46.2.7.1 [Writing Accessors], page 777 section for details.

46.2.3 A Method is Simply a Subroutine

Perl does not provide any special syntax for defining a method. A method is simply a regular subroutine, and is declared with `sub`. What makes a method special is that it expects to receive either an object or a class name as its first argument.

Perl *does* provide special syntax for method invocation, the `->` operator. We will cover this in more detail later.

Most methods you write will expect to operate on objects:

```
sub save {
    my $self = shift;

    open my $fh, '>', $self->path() or die $!;
    print {$fh} $self->data()       or die $!;
    close $fh                       or die $!;
}
```

46.2.4 Method Invocation >>

Calling a method on an object is written as `$object->method`.

The left hand side of the method invocation (or arrow) operator is the object (or class name), and the right hand side is the method name.

```
my $pod = File->new( 'perlobj.pod', $data );
$pod->save();
```

The `->` syntax is also used when dereferencing a reference. It looks like the same operator, but these are two different operations.

When you call a method, the thing on the left side of the arrow is passed as the first argument to the method. That means when we call `Critter->new()`, the `new()` method receives the string `"Critter"` as its first argument. When we call `$fred->speak()`, the `$fred` variable is passed as the first argument to `speak()`.

Just as with any Perl subroutine, all of the arguments passed in `@_` are aliases to the original argument. This includes the object itself. If you assign directly to `$_[0]` you will change the contents of the variable that holds the reference to the object. We recommend that you don't do this unless you know exactly what you're doing.

Perl knows what package the method is in by looking at the left side of the arrow. If the left hand side is a package name, it looks for the method in that package. If the left hand side is an object, then Perl looks for the method in the package that the object has been blessed into.

If the left hand side is neither a package name nor an object, then the method call will cause an error, but see the section on Section 46.2.9 [Method Call Variations], page 778 for more nuances.

46.2.5 Inheritance

We already talked about the special `@ISA` array and the `parent` pragma.

When a class inherits from another class, any methods defined in the parent class are available to the child class. If you attempt to call a method on an object that isn't defined in its own class, Perl will also look for that method in any parent classes it may have.

```
package File::MP3;
use parent 'File';    # sets @File::MP3::ISA = ('File');

my $mp3 = File::MP3->new( 'Andvari.mp3', $data );
$mp3->save();
```

Since we didn't define a `save()` method in the `File::MP3` class, Perl will look at the `File::MP3` class's parent classes to find the `save()` method. If Perl cannot find a `save()` method anywhere in the inheritance hierarchy, it will die.

In this case, it finds a `save()` method in the `File` class. Note that the object passed to `save()` in this case is still a `File::MP3` object, even though the method is found in the `File` class.

We can override a parent's method in a child class. When we do so, we can still call the parent class's method with the `SUPER` pseudo-class.

```
sub save {
    my $self = shift;

    say 'Prepare to rock';
    $self->SUPER::save();
}
```

The `SUPER` modifier can *only* be used for method calls. You can't use it for regular subroutine calls or class methods:

```
SUPER::save($thing);     # FAIL: looks for save() sub in package SUPER

SUPER->save($thing);     # FAIL: looks for save() method in class
                         #       SUPER

$thing->SUPER::save();   # Okay: looks for save() method in parent
                         #       classes
```

46.2.5.1 How SUPER is Resolved

The `SUPER` pseudo-class is resolved from the package where the call is made. It is *not* resolved based on the object's class. This is important, because it lets methods at different levels within a deep inheritance hierarchy each correctly call their respective parent methods.

```
package A;

sub new {
    return bless {}, shift;
}

sub speak {
    my $self = shift;

    say 'A';
}
```

```
package B;

use parent -norequire, 'A';

sub speak {
    my $self = shift;

    $self->SUPER::speak();

    say 'B';
}

package C;

use parent -norequire, 'B';

sub speak {
    my $self = shift;

    $self->SUPER::speak();

    say 'C';
}

my $c = C->new();
$c->speak();
```

In this example, we will get the following output:

```
A
B
C
```

This demonstrates how `SUPER` is resolved. Even though the object is blessed into the `C` class, the `speak()` method in the `B` class can still call `SUPER::speak()` and expect it to correctly look in the parent class of `B` (i.e the class the method call is in), not in the parent class of `C` (i.e. the class the object belongs to).

There are rare cases where this package-based resolution can be a problem. If you copy a subroutine from one package to another, `SUPER` resolution will be done based on the original package.

46.2.5.2 Multiple Inheritance

Multiple inheritance often indicates a design problem, but Perl always gives you enough rope to hang yourself with if you ask for it.

To declare multiple parents, you simply need to pass multiple class names to `use parent`:

```
package MultiChild;

use parent 'Parent1', 'Parent2';
```

46.2.5.3 Method Resolution Order

Method resolution order only matters in the case of multiple inheritance. In the case of single inheritance, Perl simply looks up the inheritance chain to find a method:

```
Grandparent
  |
Parent
  |
Child
```

If we call a method on a `Child` object and that method is not defined in the `Child` class, Perl will look for that method in the `Parent` class and then, if necessary, in the `Grandparent` class.

If Perl cannot find the method in any of these classes, it will die with an error message.

When a class has multiple parents, the method lookup order becomes more complicated.

By default, Perl does a depth-first left-to-right search for a method. That means it starts with the first parent in the `@ISA` array, and then searches all of its parents, grandparents, etc. If it fails to find the method, it then goes to the next parent in the original class's `@ISA` array and searches from there.

```
          SharedGreatGrandParent
          /                    \
PaternalGrandparent       MaternalGrandparent
          \                    /
           Father        Mother
                 \      /
                  Child
```

So given the diagram above, Perl will search `Child`, `Father`, `PaternalGrandparent`, `SharedGreatGrandParent`, `Mother`, and finally `MaternalGrandparent`. This may be a problem because now we're looking in `SharedGreatGrandParent` *before* we've checked all its derived classes (i.e. before we tried `Mother` and `MaternalGrandparent`).

It is possible to ask for a different method resolution order with the `mro` pragma.

```
package Child;

use mro 'c3';
use parent 'Father', 'Mother';
```

This pragma lets you switch to the "C3" resolution order. In simple terms, "C3" order ensures that shared parent classes are never searched before child classes, so Perl will now search: `Child`, `Father`, `PaternalGrandparent`, `Mother` `MaternalGrandparent`, and finally `SharedGreatGrandParent`. Note however that this is not "breadth-first" searching: All the `Father` ancestors (except the common ancestor) are searched before any of the `Mother` ancestors are considered.

The C3 order also lets you call methods in sibling classes with the `next` pseudo-class. See the `mro` documentation for more details on this feature.

46.2.5.4 Method Resolution Caching

When Perl searches for a method, it caches the lookup so that future calls to the method do not need to search for it again. Changing a class's parent class or adding subroutines to a class will invalidate the cache for that class.

The mro pragma provides some functions for manipulating the method cache directly.

46.2.6 Writing Constructors

As we mentioned earlier, Perl provides no special constructor syntax. This means that a class must implement its own constructor. A constructor is simply a class method that returns a reference to a new object.

The constructor can also accept additional parameters that define the object. Let's write a real constructor for the File class we used earlier:

```
package File;

sub new {
    my $class = shift;
    my ( $path, $data ) = @_;

    my $self = bless {
        path => $path,
        data => $data,
    }, $class;

    return $self;
}
```

As you can see, we've stored the path and file data in the object itself. Remember, under the hood, this object is still just a hash. Later, we'll write accessors to manipulate this data.

For our File::MP3 class, we can check to make sure that the path we're given ends with ".mp3":

```
package File::MP3;

sub new {
    my $class = shift;
    my ( $path, $data ) = @_;

    die "You cannot create a File::MP3 without an mp3 extension\n"
        unless $path =~ /\.mp3\z/;

    return $class->SUPER::new(@_);
}
```

This constructor lets its parent class do the actual object construction.

46.2.7 Attributes

An attribute is a piece of data belonging to a particular object. Unlike most object-oriented languages, Perl provides no special syntax or support for declaring and manipulating attributes.

Attributes are often stored in the object itself. For example, if the object is an anonymous hash, we can store the attribute values in the hash using the attribute name as the key.

While it's possible to refer directly to these hash keys outside of the class, it's considered a best practice to wrap all access to the attribute with accessor methods.

This has several advantages. Accessors make it easier to change the implementation of an object later while still preserving the original API.

An accessor lets you add additional code around attribute access. For example, you could apply a default to an attribute that wasn't set in the constructor, or you could validate that a new value for the attribute is acceptable.

Finally, using accessors makes inheritance much simpler. Subclasses can use the accessors rather than having to know how a parent class is implemented internally.

46.2.7.1 Writing Accessors

As with constructors, Perl provides no special accessor declaration syntax, so classes must provide explicitly written accessor methods. There are two common types of accessors, read-only and read-write.

A simple read-only accessor simply gets the value of a single attribute:

```
sub path {
    my $self = shift;

    return $self->{path};
}
```

A read-write accessor will allow the caller to set the value as well as get it:

```
sub path {
    my $self = shift;

    if (@_) {
        $self->{path} = shift;
    }

    return $self->{path};
}
```

46.2.8 An Aside About Smarter and Safer Code

Our constructor and accessors are not very smart. They don't check that a `$path` is defined, nor do they check that a `$path` is a valid filesystem path.

Doing these checks by hand can quickly become tedious. Writing a bunch of accessors by hand is also incredibly tedious. There are a lot of modules on CPAN that can help you write safer and more concise code, including the modules we recommend in Section 47.1 [perlootut NAME], page 786.

46.2.9 Method Call Variations

Perl supports several other ways to call methods besides the `$object->method()` usage we've seen so far.

46.2.9.1 Method Names as Strings

Perl lets you use a scalar variable containing a string as a method name:

```
my $file = File->new( $path, $data );

my $method = 'save';
$file->$method();
```

This works exactly like calling `$file->save()`. This can be very useful for writing dynamic code. For example, it allows you to pass a method name to be called as a parameter to another method.

46.2.9.2 Class Names as Strings

Perl also lets you use a scalar containing a string as a class name:

```
my $class = 'File';

my $file = $class->new( $path, $data );
```

Again, this allows for very dynamic code.

46.2.9.3 Subroutine References as Methods

You can also use a subroutine reference as a method:

```
my $sub = sub {
    my $self = shift;

    $self->save();
};

$file->$sub();
```

This is exactly equivalent to writing `$sub->($file)`. You may see this idiom in the wild combined with a call to `can`:

```
if ( my $meth = $object->can('foo') ) {
    $object->$meth();
}
```

46.2.9.4 Deferencing Method Call

Perl also lets you use a dereferenced scalar reference in a method call. That's a mouthful, so let's look at some code:

```
$file->${ \'save' };
$file->${ returns_scalar_ref() };
$file->${ \( returns_scalar() ) };
$file->${ returns_ref_to_sub_ref() };
```

This works if the dereference produces a string *or* a subroutine reference.

46.2.9.5 Method Calls on Filehandles

Under the hood, Perl filehandles are instances of the `IO::Handle` or `IO::File` class. Once you have an open filehandle, you can call methods on it. Additionally, you can call methods on the `STDIN`, `STDOUT`, and `STDERR` filehandles.

```
open my $fh, '>', 'path/to/file';
$fh->autoflush();
$fh->print('content');

STDOUT->autoflush();
```

46.2.10 Invoking Class Methods

Because Perl allows you to use barewords for package names and subroutine names, it sometimes interprets a bareword's meaning incorrectly. For example, the construct `Class->new()` can be interpreted as either `'Class'->new()` or `Class()->new()`. In English, that second interpretation reads as "call a subroutine named Class(), then call new() as a method on the return value of Class()". If there is a subroutine named `Class()` in the current namespace, Perl will always interpret `Class->new()` as the second alternative: a call to `new()` on the object returned by a call to `Class()`

You can force Perl to use the first interpretation (i.e. as a method call on the class named "Class") in two ways. First, you can append a `::` to the class name:

```
Class::->new()
```

Perl will always interpret this as a method call.

Alternatively, you can quote the class name:

```
'Class'->new()
```

Of course, if the class name is in a scalar Perl will do the right thing as well:

```
my $class = 'Class';
$class->new();
```

46.2.10.1 Indirect Object Syntax

Outside of the file handle case, use of this syntax is discouraged as it can confuse the Perl interpreter. See below for more details.

Perl supports another method invocation syntax called "indirect object" notation. This syntax is called "indirect" because the method comes before the object it is being invoked on.

This syntax can be used with any class or object method:

```
my $file = new File $path, $data;
save $file;
```

We recommend that you avoid this syntax, for several reasons.

First, it can be confusing to read. In the above example, it's not clear if `save` is a method provided by the `File` class or simply a subroutine that expects a file object as its first argument.

When used with class methods, the problem is even worse. Because Perl allows subroutine names to be written as barewords, Perl has to guess whether the bareword after the method is a class name or subroutine name. In other words, Perl can resolve the syntax as either `File->new( $path, $data )` **or** `new( File( $path, $data ) )`.

To parse this code, Perl uses a heuristic based on what package names it has seen, what subroutines exist in the current package, what barewords it has previously seen, and other input. Needless to say, heuristics can produce very surprising results!

Older documentation (and some CPAN modules) encouraged this syntax, particularly for constructors, so you may still find it in the wild. However, we encourage you to avoid using it in new code.

You can force Perl to interpret the bareword as a class name by appending "::" to it, like we saw earlier:

```
my $file = new File:: $path, $data;
```

46.2.11 `bless`, `blessed`, and `ref`

As we saw earlier, an object is simply a data structure that has been blessed into a class via the `bless` function. The `bless` function can take either one or two arguments:

```
my $object = bless {}, $class;
my $object = bless {};
```

In the first form, the anonymous hash is being blessed into the class in `$class`. In the second form, the anonymous hash is blessed into the current package.

The second form is strongly discouraged, because it breaks the ability of a subclass to reuse the parent's constructor, but you may still run across it in existing code.

If you want to know whether a particular scalar refers to an object, you can use the `blessed` function exported by `Scalar-Util`, which is shipped with the Perl core.

```
use Scalar::Util 'blessed';

if ( defined blessed($thing) ) { ... }
```

If `$thing` refers to an object, then this function returns the name of the package the object has been blessed into. If `$thing` doesn't contain a reference to a blessed object, the `blessed` function returns `undef`.

Note that `blessed($thing)` will also return false if `$thing` has been blessed into a class named `"0"`. This is a possible, but quite pathological. Don't create a class named `"0"` unless you know what you're doing.

Similarly, Perl's built-in `ref` function treats a reference to a blessed object specially. If you call `ref($thing)` and `$thing` holds a reference to an object, it will return the name of the class that the object has been blessed into.

If you simply want to check that a variable contains an object reference, we recommend that you use `defined blessed($object)`, since `ref` returns true values for all references, not just objects.

46.2.12 The UNIVERSAL Class

All classes automatically inherit from the `UNIVERSAL` class, which is built-in to the Perl core. This class provides a number of methods, all of which can be called on either a class or an object. You can also choose to override some of these methods in your class. If you do so, we recommend that you follow the built-in semantics described below.

isa($class)

The isa method returns *true* if the object is a member of the class in $class, or a member of a subclass of $class.

If you override this method, it should never throw an exception.

DOES($role)

The DOES method returns *true* if its object claims to perform the role $role. By default, this is equivalent to isa. This method is provided for use by object system extensions that implement roles, like Moose and Role::Tiny.

You can also override DOES directly in your own classes. If you override this method, it should never throw an exception.

can($method)

The can method checks to see if the class or object it was called on has a method named $method. This checks for the method in the class and all of its parents. If the method exists, then a reference to the subroutine is returned. If it does not then undef is returned.

If your class responds to method calls via AUTOLOAD, you may want to overload can to return a subroutine reference for methods which your AUTOLOAD method handles.

If you override this method, it should never throw an exception.

VERSION($need)

The VERSION method returns the version number of the class (package).

If the $need argument is given then it will check that the current version (as defined by the $VERSION variable in the package) is greater than or equal to $need; it will die if this is not the case. This method is called automatically by the VERSION form of use.

```
use Package 1.2 qw(some imported subs);
# implies:
Package->VERSION(1.2);
```

We recommend that you use this method to access another package's version, rather than looking directly at $Package::VERSION. The package you are looking at could have overridden the VERSION method.

We also recommend using this method to check whether a module has a sufficient version. The internal implementation uses the version module to make sure that different types of version numbers are compared correctly.

46.2.13 AUTOLOAD

If you call a method that doesn't exist in a class, Perl will throw an error. However, if that class or any of its parent classes defines an AUTOLOAD method, that AUTOLOAD method is called instead.

AUTOLOAD is called as a regular method, and the caller will not know the difference. Whatever value your AUTOLOAD method returns is returned to the caller.

The fully qualified method name that was called is available in the $AUTOLOAD package global for your class. Since this is a global, if you want to refer to do it without a package name prefix under strict 'vars', you need to declare it.

```
# XXX - this is a terrible way to implement accessors, but it makes
# for a simple example.
our $AUTOLOAD;
sub AUTOLOAD {
    my $self = shift;

    # Remove qualifier from original method name...
    my $called =  $AUTOLOAD =~ s/.*:://r;

    # Is there an attribute of that name?
    die "No such attribute: $called"
        unless exists $self->{$called};

    # If so, return it...
    return $self->{$called};
}

sub DESTROY { } # see below
```

Without the `our $AUTOLOAD` declaration, this code will not compile under the `strict` pragma.

As the comment says, this is not a good way to implement accessors. It's slow and too clever by far. However, you may see this as a way to provide accessors in older Perl code. See Section 47.1 [perlootut NAME], page 786 for recommendations on OO coding in Perl.

If your class does have an `AUTOLOAD` method, we strongly recommend that you override `can` in your class as well. Your overridden `can` method should return a subroutine reference for any method that your `AUTOLOAD` responds to.

46.2.14 Destructors

When the last reference to an object goes away, the object is destroyed. If you only have one reference to an object stored in a lexical scalar, the object is destroyed when that scalar goes out of scope. If you store the object in a package global, that object may not go out of scope until the program exits.

If you want to do something when the object is destroyed, you can define a `DESTROY` method in your class. This method will always be called by Perl at the appropriate time, unless the method is empty.

This is called just like any other method, with the object as the first argument. It does not receive any additional arguments. However, the `$_[0]` variable will be read-only in the destructor, so you cannot assign a value to it.

If your `DESTROY` method throws an error, this error will be ignored. It will not be sent to `STDERR` and it will not cause the program to die. However, if your destructor is running inside an `eval {}` block, then the error will change the value of `$@`.

Because `DESTROY` methods can be called at any time, you should localize any global variables you might update in your `DESTROY`. In particular, if you use `eval {}` you should localize `$@`, and if you use `system` or backticks you should localize `$?`.

If you define an AUTOLOAD in your class, then Perl will call your AUTOLOAD to handle the DESTROY method. You can prevent this by defining an empty DESTROY, like we did in the autoloading example. You can also check the value of $AUTOLOAD and return without doing anything when called to handle DESTROY.

46.2.14.1 Global Destruction

The order in which objects are destroyed during the global destruction before the program exits is unpredictable. This means that any objects contained by your object may already have been destroyed. You should check that a contained object is defined before calling a method on it:

```
sub DESTROY {
    my $self = shift;

    $self->{handle}->close() if $self->{handle};
}
```

You can use the ${^GLOBAL_PHASE} variable to detect if you are currently in the global destruction phase:

```
sub DESTROY {
    my $self = shift;

    return if ${^GLOBAL_PHASE} eq 'DESTRUCT';

    $self->{handle}->close();
}
```

Note that this variable was added in Perl 5.14.0. If you want to detect the global destruction phase on older versions of Perl, you can use the Devel::GlobalDestruction module on CPAN.

If your DESTROY method issues a warning during global destruction, the Perl interpreter will append the string " during global destruction" to the warning.

During global destruction, Perl will always garbage collect objects before unblessed references. See Section 30.8.1 [perlhacktips PERL_DESTRUCT_LEVEL], page 596 for more information about global destruction.

46.2.15 Non-Hash Objects

All the examples so far have shown objects based on a blessed hash. However, it's possible to bless any type of data structure or referent, including scalars, globs, and subroutines. You may see this sort of thing when looking at code in the wild.

Here's an example of a module as a blessed scalar:

```
package Time;

use strict;
use warnings;

sub new {
    my $class = shift;
    my $time = time;
    return bless \$time, $class;
}

sub epoch {
    my $self = shift;
    return ${ $self };
}

my $time = Time->new();
print $time->epoch();
```

46.2.16 Inside-Out objects

In the past, the Perl community experimented with a technique called "inside-out objects". An inside-out object stores its data outside of the object's reference, indexed on a unique property of the object, such as its memory address, rather than in the object itself. This has the advantage of enforcing the encapsulation of object attributes, since their data is not stored in the object itself.

This technique was popular for a while (and was recommended in Damian Conway's *Perl Best Practices*), but never achieved universal adoption. The `Object-InsideOut` module on CPAN provides a comprehensive implementation of this technique, and you may see it or other inside-out modules in the wild.

Here is a simple example of the technique, using the `Hash-Util-FieldHash` core module. This module was added to the core to support inside-out object implementations.

```
package Time;

use strict;
use warnings;

use Hash::Util::FieldHash 'fieldhash';

fieldhash my %time_for;

sub new {
    my $class = shift;

    my $self = bless \( my $object ), $class;

    $time_for{$self} = time;

    return $self;
}

sub epoch {
    my $self = shift;
    return $time_for{$self};
}

my $time = Time->new;
print $time->epoch;
```

46.2.17 Pseudo-hashes

The pseudo-hash feature was an experimental feature introduced in earlier versions of Perl and removed in 5.10.0. A pseudo-hash is an array reference which can be accessed using named keys like a hash. You may run in to some code in the wild which uses it. See the `fields` pragma for more information.

46.3 SEE ALSO

A kinder, gentler tutorial on object-oriented programming in Perl can be found in Section 47.1 [perlootut NAME], page 786. You should also check out `perlmodlib` for some style guides on constructing both modules and classes.

47 perlootut

47.1 NAME

perlootut - Object-Oriented Programming in Perl Tutorial

47.2 DATE

This document was created in February, 2011, and the last major revision was in February, 2013.

If you are reading this in the future then it's possible that the state of the art has changed. We recommend you start by reading the perlootut document in the latest stable release of Perl, rather than this version.

47.3 DESCRIPTION

This document provides an introduction to object-oriented programming in Perl. It begins with a brief overview of the concepts behind object oriented design. Then it introduces several different OO systems from CPAN (`http://search.cpan.org`) which build on top of what Perl provides.

By default, Perl's built-in OO system is very minimal, leaving you to do most of the work. This minimalism made a lot of sense in 1994, but in the years since Perl 5.0 we've seen a number of common patterns emerge in Perl OO. Fortunately, Perl's flexibility has allowed a rich ecosystem of Perl OO systems to flourish.

If you want to know how Perl OO works under the hood, the Section 46.1 [perlobj NAME], page 769 document explains the nitty gritty details.

This document assumes that you already understand the basics of Perl syntax, variable types, operators, and subroutine calls. If you don't understand these concepts yet, please read Section 34.1 [perlintro NAME], page 639 first. You should also read the Section 74.1 [perlsyn NAME], page 1249, Section 48.1 [perlop NAME], page 798, and Section 73.1 [perlsub NAME], page 1216 documents.

47.4 OBJECT-ORIENTED FUNDAMENTALS

Most object systems share a number of common concepts. You've probably heard terms like "class", "object, "method", and "attribute" before. Understanding the concepts will make it much easier to read and write object-oriented code. If you're already familiar with these terms, you should still skim this section, since it explains each concept in terms of Perl's OO implementation.

Perl's OO system is class-based. Class-based OO is fairly common. It's used by Java, C++, C#, Python, Ruby, and many other languages. There are other object orientation paradigms as well. JavaScript is the most popular language to use another paradigm. JavaScript's OO system is prototype-based.

47.4.1 Object

An **object** is a data structure that bundles together data and subroutines which operate on that data. An object's data is called **attributes**, and its subroutines are called **methods**. An object can be thought of as a noun (a person, a web service, a computer).

An object represents a single discrete thing. For example, an object might represent a file. The attributes for a file object might include its path, content, and last modification time. If we created an object to represent `/etc/hostname` on a machine named "foo.example.com", that object's path would be "/etc/hostname", its content would be "foo\n", and it's last modification time would be 1304974868 seconds since the beginning of the epoch.

The methods associated with a file might include `rename()` and `write()`.

In Perl most objects are hashes, but the OO systems we recommend keep you from having to worry about this. In practice, it's best to consider an object's internal data structure opaque.

47.4.2 Class

A **class** defines the behavior of a category of objects. A class is a name for a category (like "File"), and a class also defines the behavior of objects in that category.

All objects belong to a specific class. For example, our `/etc/hostname` object belongs to the `File` class. When we want to create a specific object, we start with its class, and **construct** or **instantiate** an object. A specific object is often referred to as an **instance** of a class.

In Perl, any package can be a class. The difference between a package which is a class and one which isn't is based on how the package is used. Here's our "class declaration" for the `File` class:

```
package File;
```

In Perl, there is no special keyword for constructing an object. However, most OO modules on CPAN use a method named `new()` to construct a new object:

```
my $hostname = File->new(
    path          => '/etc/hostname',
    content       => "foo\n",
    last_mod_time => 1304974868,
);
```

(Don't worry about that `->` operator, it will be explained later.)

47.4.2.1 Blessing

As we said earlier, most Perl objects are hashes, but an object can be an instance of any Perl data type (scalar, array, etc.). Turning a plain data structure into an object is done by **blessing** that data structure using Perl's `bless` function.

While we strongly suggest you don't build your objects from scratch, you should know the term **bless**. A **blessed** data structure (aka "a referent") is an object. We sometimes say that an object has been "blessed into a class".

Once a referent has been blessed, the `blessed` function from the `Scalar-Util` core module can tell us its class name. This subroutine returns an object's class when passed an object, and false otherwise.

```
use Scalar::Util 'blessed';

print blessed($hash);      # undef
print blessed($hostname);  # File
```

47.4.2.2 Constructor

A **constructor** creates a new object. In Perl, a class's constructor is just another method, unlike some other languages, which provide syntax for constructors. Most Perl classes use `new` as the name for their constructor:

```
my $file = File->new(...);
```

47.4.3 Methods

You already learned that a **method** is a subroutine that operates on an object. You can think of a method as the things that an object can *do*. If an object is a noun, then methods are its verbs (save, print, open).

In Perl, methods are simply subroutines that live in a class's package. Methods are always written to receive the object as their first argument:

```
sub print_info {
    my $self = shift;

    print "This file is at ", $self->path, "\n";
}

$file->print_info;
# The file is at /etc/hostname
```

What makes a method special is *how it's called*. The arrow operator (`->`) tells Perl that we are calling a method.

When we make a method call, Perl arranges for the method's **invocant** to be passed as the first argument. **Invocant** is a fancy name for the thing on the left side of the arrow. The invocant can either be a class name or an object. We can also pass additional arguments to the method:

```
sub print_info {
    my $self   = shift;
    my $prefix = shift // "This file is at ";

    print $prefix, ", ", $self->path, "\n";
}

$file->print_info("The file is located at ");
# The file is located at /etc/hostname
```

47.4.4 Attributes

Each class can define its **attributes**. When we instantiate an object, we assign values to those attributes. For example, every `File` object has a path. Attributes are sometimes called **properties**.

Perl has no special syntax for attributes. Under the hood, attributes are often stored as keys in the object's underlying hash, but don't worry about this.

We recommend that you only access attributes via **accessor** methods. These are methods that can get or set the value of each attribute. We saw this earlier in the `print_info()` example, which calls `$self->path`.

You might also see the terms **getter** and **setter**. These are two types of accessors. A getter gets the attribute's value, while a setter sets it. Another term for a setter is **mutator**

Attributes are typically defined as read-only or read-write. Read-only attributes can only be set when the object is first created, while read-write attributes can be altered at any time.

The value of an attribute may itself be another object. For example, instead of returning its last mod time as a number, the `File` class could return a `DateTime` object representing that value.

It's possible to have a class that does not expose any publicly settable attributes. Not every class has attributes and methods.

47.4.5 Polymorphism

Polymorphism is a fancy way of saying that objects from two different classes share an API. For example, we could have `File` and `WebPage` classes which both have a `print_content()` method. This method might produce different output for each class, but they share a common interface.

While the two classes may differ in many ways, when it comes to the `print_content()` method, they are the same. This means that we can try to call the `print_content()` method on an object of either class, and **we don't have to know what class the object belongs to!**

Polymorphism is one of the key concepts of object-oriented design.

47.4.6 Inheritance

Inheritance lets you create a specialized version of an existing class. Inheritance lets the new class reuse the methods and attributes of another class.

For example, we could create an `File::MP3` class which **inherits** from `File`. An `File::MP3` **is-a** *more specific* type of `File`. All mp3 files are files, but not all files are mp3 files.

We often refer to inheritance relationships as **parent-child** or `superclass/subclass` relationships. Sometimes we say that the child has an **is-a** relationship with its parent class.

`File` is a **superclass** of `File::MP3`, and `File::MP3` is a **subclass** of `File`.

```
package File::MP3;

use parent 'File';
```

The `parent` module is one of several ways that Perl lets you define inheritance relationships.

Perl allows multiple inheritance, which means that a class can inherit from multiple parents. While this is possible, we strongly recommend against it. Generally, you can use **roles** to do everything you can do with multiple inheritance, but in a cleaner way.

Note that there's nothing wrong with defining multiple subclasses of a given class. This is both common and safe. For example, we might define `File::MP3::FixedBitrate` and `File::MP3::VariableBitrate` classes to distinguish between different types of mp3 file.

47.4.6.1 Overriding methods and method resolution

Inheritance allows two classes to share code. By default, every method in the parent class is also available in the child. The child can explicitly **override** a parent's method to provide its own implementation. For example, if we have an `File::MP3` object, it has the `print_info()` method from `File`:

```
my $cage = File::MP3->new(
    path          => 'mp3s/My-Body-Is-a-Cage.mp3',
    content       => $mp3_data,
    last_mod_time => 1304974868,
    title         => 'My Body Is a Cage',
);

$cage->print_info;
# The file is at mp3s/My-Body-Is-a-Cage.mp3
```

If we wanted to include the mp3's title in the greeting, we could override the method:

```
package File::MP3;

use parent 'File';

sub print_info {
    my $self = shift;

    print "This file is at ", $self->path, "\n";
    print "Its title is ", $self->title, "\n";
}

$cage->print_info;
# The file is at mp3s/My-Body-Is-a-Cage.mp3
# Its title is My Body Is a Cage
```

The process of determining what method should be used is called **method resolution**. What Perl does is look at the object's class first (`File::MP3` in this case). If that class defines the method, then that class's version of the method is called. If not, Perl looks at each parent class in turn. For `File::MP3`, its only parent is `File`. If `File::MP3` does not define the method, but `File` does, then Perl calls the method in `File`.

If `File` inherited from `DataSource`, which inherited from `Thing`, then Perl would keep looking "up the chain" if necessary.

It is possible to explicitly call a parent method from a child:

```
package File::MP3;

use parent 'File';
```

```
  sub print_info {
      my $self = shift;

      $self->SUPER::print_info();
      print "Its title is ", $self->title, "\n";
  }
```

The SUPER:: bit tells Perl to look for the print_info() in the File::MP3 class's inheritance chain. When it finds the parent class that implements this method, the method is called.

We mentioned multiple inheritance earlier. The main problem with multiple inheritance is that it greatly complicates method resolution. See Section 46.1 [perlobj NAME], page 769 for more details.

47.4.7 Encapsulation

Encapsulation is the idea that an object is opaque. When another developer uses your class, they don't need to know *how* it is implemented, they just need to know *what* it does.

Encapsulation is important for several reasons. First, it allows you to separate the public API from the private implementation. This means you can change that implementation without breaking the API.

Second, when classes are well encapsulated, they become easier to subclass. Ideally, a subclass uses the same APIs to access object data that its parent class uses. In reality, subclassing sometimes involves violating encapsulation, but a good API can minimize the need to do this.

We mentioned earlier that most Perl objects are implemented as hashes under the hood. The principle of encapsulation tells us that we should not rely on this. Instead, we should use accessor methods to access the data in that hash. The object systems that we recommend below all automate the generation of accessor methods. If you use one of them, you should never have to access the object as a hash directly.

47.4.8 Composition

In object-oriented code, we often find that one object references another object. This is called **composition**, or a **has-a** relationship.

Earlier, we mentioned that the File class's last_mod_time accessor could return a DateTime object. This is a perfect example of composition. We could go even further, and make the path and content accessors return objects as well. The File class would then be **composed** of several other objects.

47.4.9 Roles

Roles are something that a class *does*, rather than something that it *is*. Roles are relatively new to Perl, but have become rather popular. Roles are **applied** to classes. Sometimes we say that classes **consume** roles.

Roles are an alternative to inheritance for providing polymorphism. Let's assume we have two classes, Radio and Computer. Both of these things have on/off switches. We want to model that in our class definitions.

We could have both classes inherit from a common parent, like `Machine`, but not all machines have on/off switches. We could create a parent class called `HasOnOffSwitch`, but that is very artificial. Radios and computers are not specializations of this parent. This parent is really a rather ridiculous creation.

This is where roles come in. It makes a lot of sense to create a `HasOnOffSwitch` role and apply it to both classes. This role would define a known API like providing `turn_on()` and `turn_off()` methods.

Perl does not have any built-in way to express roles. In the past, people just bit the bullet and used multiple inheritance. Nowadays, there are several good choices on CPAN for using roles.

47.4.10 When to Use OO

Object Orientation is not the best solution to every problem. In *Perl Best Practices* (copyright 2004, Published by O'Reilly Media, Inc.), Damian Conway provides a list of criteria to use when deciding if OO is the right fit for your problem:

- The system being designed is large, or is likely to become large.
- The data can be aggregated into obvious structures, especially if there's a large amount of data in each aggregate.
- The various types of data aggregate form a natural hierarchy that facilitates the use of inheritance and polymorphism.
- You have a piece of data on which many different operations are applied.
- You need to perform the same general operations on related types of data, but with slight variations depending on the specific type of data the operations are applied to.
- It's likely you'll have to add new data types later.
- The typical interactions between pieces of data are best represented by operators.
- The implementation of individual components of the system is likely to change over time.
- The system design is already object-oriented.
- Large numbers of other programmers will be using your code modules.

47.5 PERL OO SYSTEMS

As we mentioned before, Perl's built-in OO system is very minimal, but also quite flexible. Over the years, many people have developed systems which build on top of Perl's built-in system to provide more features and convenience.

We strongly recommend that you use one of these systems. Even the most minimal of them eliminates a lot of repetitive boilerplate. There's really no good reason to write your classes from scratch in Perl.

If you are interested in the guts underlying these systems, check out Section 46.1 [perlobj NAME], page 769.

47.5.1 Moose

`Moose` bills itself as a "postmodern object system for Perl 5". Don't be scared, the "postmodern" label is a callback to Larry's description of Perl as "the first postmodern computer language".

Moose provides a complete, modern OO system. Its biggest influence is the Common Lisp Object System, but it also borrows ideas from Smalltalk and several other languages. Moose was created by Stevan Little, and draws heavily from his work on the Perl 6 OO design.

Here is our File class using Moose:

```
package File;
use Moose;

has path          => ( is => 'ro' );
has content       => ( is => 'ro' );
has last_mod_time => ( is => 'ro' );

sub print_info {
    my $self = shift;

    print "This file is at ", $self->path, "\n";
}
```

Moose provides a number of features:

- Declarative sugar

  Moose provides a layer of declarative "sugar" for defining classes. That sugar is just a set of exported functions that make declaring how your class works simpler and more palatable. This lets you describe *what* your class is, rather than having to tell Perl *how* to implement your class.

  The has() subroutine declares an attribute, and Moose automatically creates accessors for these attributes. It also takes care of creating a new() method for you. This constructor knows about the attributes you declared, so you can set them when creating a new File.

- Roles built-in

  Moose lets you define roles the same way you define classes:

  ```
  package HasOnOfSwitch;
  use Moose::Role;

  has is_on => (
      is  => 'rw',
      isa => 'Bool',
  );

  sub turn_on {
      my $self = shift;
      $self->is_on(1);
  }

  sub turn_off {
      my $self = shift;
      $self->is_on(0);
  ```

```
    }
```

- A miniature type system

  In the example above, you can see that we passed `isa => 'Bool'` to `has()` when creating our `is_on` attribute. This tells `Moose` that this attribute must be a boolean value. If we try to set it to an invalid value, our code will throw an error.

- Full introspection and manipulation

  Perl's built-in introspection features are fairly minimal. `Moose` builds on top of them and creates a full introspection layer for your classes. This lets you ask questions like "what methods does the File class implement?" It also lets you modify your classes programmatically.

- Self-hosted and extensible

  `Moose` describes itself using its own introspection API. Besides being a cool trick, this means that you can extend `Moose` using `Moose` itself.

- Rich ecosystem

  There is a rich ecosystem of `Moose` extensions on CPAN under the MooseX (`http://search.cpan.org/search?query=MooseX&mode=dist`) namespace. In addition, many modules on CPAN already use `Moose`, providing you with lots of examples to learn from.

- Many more features

  `Moose` is a very powerful tool, and we can't cover all of its features here. We encourage you to learn more by reading the `Moose` documentation, starting with Moose::Manual (`http://search.cpan.org/perldoc?Moose::Manual`).

Of course, `Moose` isn't perfect.

`Moose` can make your code slower to load. `Moose` itself is not small, and it does a *lot* of code generation when you define your class. This code generation means that your runtime code is as fast as it can be, but you pay for this when your modules are first loaded.

This load time hit can be a problem when startup speed is important, such as with a command-line script or a "plain vanilla" CGI script that must be loaded each time it is executed.

Before you panic, know that many people do use `Moose` for command-line tools and other startup-sensitive code. We encourage you to try `Moose` out first before worrying about startup speed.

`Moose` also has several dependencies on other modules. Most of these are small standalone modules, a number of which have been spun off from `Moose`. `Moose` itself, and some of its dependencies, require a compiler. If you need to install your software on a system without a compiler, or if having *any* dependencies is a problem, then `Moose` may not be right for you.

47.5.1.1 Moo

If you try `Moose` and find that one of these issues is preventing you from using `Moose`, we encourage you to consider `Moo` next. `Moo` implements a subset of `Moose`'s functionality in a simpler package. For most features that it does implement, the end-user API is *identical* to `Moose`, meaning you can switch from `Moo` to `Moose` quite easily.

Moo does not implement most of Moose's introspection API, so it's often faster when loading your modules. Additionally, none of its dependencies require XS, so it can be installed on machines without a compiler.

One of Moo's most compelling features is its interoperability with Moose. When someone tries to use Moose's introspection API on a Moo class or role, it is transparently inflated into a Moose class or role. This makes it easier to incorporate Moo-using code into a Moose code base and vice versa.

For example, a Moose class can subclass a Moo class using extends or consume a Moo role using with.

The Moose authors hope that one day Moo can be made obsolete by improving Moose enough, but for now it provides a worthwhile alternative to Moose.

47.5.2 Class::Accessor

Class-Accessor is the polar opposite of Moose. It provides very few features, nor is it self-hosting.

It is, however, very simple, pure Perl, and it has no non-core dependencies. It also provides a "Moose-like" API on demand for the features it supports.

Even though it doesn't do much, it is still preferable to writing your own classes from scratch.

Here's our File class with Class::Accessor:

```
package File;
use Class::Accessor 'antlers';

has path          => ( is => 'ro' );
has content       => ( is => 'ro' );
has last_mod_time => ( is => 'ro' );

sub print_info {
    my $self = shift;

    print "This file is at ", $self->path, "\n";
}
```

The antlers import flag tells Class::Accessor that you want to define your attributes using Moose-like syntax. The only parameter that you can pass to has is is. We recommend that you use this Moose-like syntax if you choose Class::Accessor since it means you will have a smoother upgrade path if you later decide to move to Moose.

Like Moose, Class::Accessor generates accessor methods and a constructor for your class.

47.5.3 Class::Tiny

Finally, we have Class-Tiny. This module truly lives up to its name. It has an incredibly minimal API and absolutely no dependencies on any recent Perl. Still, we think it's a lot easier to use than writing your own OO code from scratch.

Here's our File class once more:

```
package File;
use Class::Tiny qw( path content last_mod_time );

sub print_info {
    my $self = shift;

    print "This file is at ", $self->path, "\n";
}
```

That's it!

With `Class::Tiny`, all accessors are read-write. It generates a constructor for you, as well as the accessors you define.

You can also use `Class-Tiny-Antlers` for `Moose`-like syntax.

47.5.4 Role::Tiny

As we mentioned before, roles provide an alternative to inheritance, but Perl does not have any built-in role support. If you choose to use Moose, it comes with a full-fledged role implementation. However, if you use one of our other recommended OO modules, you can still use roles with `Role-Tiny`

`Role::Tiny` provides some of the same features as Moose's role system, but in a much smaller package. Most notably, it doesn't support any sort of attribute declaration, so you have to do that by hand. Still, it's useful, and works well with `Class::Accessor` and `Class::Tiny`

47.5.5 OO System Summary

Here's a brief recap of the options we covered:

- `Moose`

  `Moose` is the maximal option. It has a lot of features, a big ecosystem, and a thriving user base. We also covered `Moo` briefly. `Moo` is `Moose` lite, and a reasonable alternative when Moose doesn't work for your application.

- `Class-Accessor`

  `Class::Accessor` does a lot less than `Moose`, and is a nice alternative if you find `Moose` overwhelming. It's been around a long time and is well battle-tested. It also has a minimal `Moose` compatibility mode which makes moving from `Class::Accessor` to `Moose` easy.

- `Class-Tiny`

  `Class::Tiny` is the absolute minimal option. It has no dependencies, and almost no syntax to learn. It's a good option for a super minimal environment and for throwing something together quickly without having to worry about details.

- `Role-Tiny`

  Use `Role::Tiny` with `Class::Accessor` or `Class::Tiny` if you find yourself considering multiple inheritance. If you go with `Moose`, it comes with its own role implementation.

47.5.6 Other OO Systems

There are literally dozens of other OO-related modules on CPAN besides those covered here, and you're likely to run across one or more of them if you work with other people's code.

In addition, plenty of code in the wild does all of its OO "by hand", using just the Perl built-in OO features. If you need to maintain such code, you should read Section 46.1 [perlobj NAME], page 769 to understand exactly how Perl's built-in OO works.

47.6 CONCLUSION

As we said before, Perl's minimal OO system has led to a profusion of OO systems on CPAN. While you can still drop down to the bare metal and write your classes by hand, there's really no reason to do that with modern Perl.

For small systems, `Class-Tiny` and `Class-Accessor` both provide minimal object systems that take care of basic boilerplate for you.

For bigger projects, `Moose` provides a rich set of features that will let you focus on implementing your business logic.

We encourage you to play with and evaluate `Moose`, `Class-Accessor`, and `Class-Tiny` to see which OO system is right for you.

48 perlop

48.1 NAME

perlop - Perl operators and precedence

48.2 DESCRIPTION

In Perl, the operator determines what operation is performed, independent of the type of the operands. For example `$x + $y` is always a numeric addition, and if `$x` or `$y` do not contain numbers, an attempt is made to convert them to numbers first.

This is in contrast to many other dynamic languages, where the operation is determined by the type of the first argument. It also means that Perl has two versions of some operators, one for numeric and one for string comparison. For example `$x == $y` compares two numbers for equality, and `$x eq $y` compares two strings.

There are a few exceptions though: `x` can be either string repetition or list repetition, depending on the type of the left operand, and `&`, `|`, `^` and `~` can be either string or numeric bit operations.

48.2.1 Operator Precedence and Associativity

Operator precedence and associativity work in Perl more or less like they do in mathematics.

Operator precedence means some operators are evaluated before others. For example, in `2 + 4 * 5`, the multiplication has higher precedence so `4 * 5` is evaluated first yielding `2 + 20 == 22` and not `6 * 5 == 30`.

Operator associativity defines what happens if a sequence of the same operators is used one after another: whether the evaluator will evaluate the left operations first, or the right first. For example, in `8 - 4 - 2`, subtraction is left associative so Perl evaluates the expression left to right. `8 - 4` is evaluated first making the expression `4 - 2 == 2` and not `8 - 2 == 6`.

Perl operators have the following associativity and precedence, listed from highest precedence to lowest. Operators borrowed from C keep the same precedence relationship with each other, even where C's precedence is slightly screwy. (This makes learning Perl easier for C folks.) With very few exceptions, these all operate on scalar values only, not array values.

```
    left        terms and list operators (leftward)
    left        ->
    nonassoc    ++ --
    right       **
    right       ! ~ \ and unary + and -
    left        =~ !~
    left        * / % x
    left        + - .
    left        << >>
    nonassoc    named unary operators
    nonassoc    < > <= >= lt gt le ge
```

```
nonassoc    == != <=> eq ne cmp ~~
left        &
left        | ^
left        &&
left        || //
nonassoc    ..  ...
right       ?:
right       = += -= *= etc. goto last next redo dump
left        , =>
nonassoc    list operators (rightward)
right       not
left        and
left        or xor
```

In the following sections, these operators are covered in precedence order.

Many operators can be overloaded for objects. See `overload`.

48.2.2 Terms and List Operators (Leftward)

A TERM has the highest precedence in Perl. They include variables, quote and quote-like operators, any expression in parentheses, and any function whose arguments are parenthesized. Actually, there aren't really functions in this sense, just list operators and unary operators behaving as functions because you put parentheses around the arguments. These are all documented in Section 25.1 [perlfunc NAME], page 351.

If any list operator (`print()`, etc.) or any unary operator (`chdir()`, etc.) is followed by a left parenthesis as the next token, the operator and arguments within parentheses are taken to be of highest precedence, just like a normal function call.

In the absence of parentheses, the precedence of list operators such as `print`, `sort`, or `chmod` is either very high or very low depending on whether you are looking at the left side or the right side of the operator. For example, in

```
@ary = (1, 3, sort 4, 2);
print @ary;         # prints 1324
```

the commas on the right of the `sort` are evaluated before the `sort`, but the commas on the left are evaluated after. In other words, list operators tend to gobble up all arguments that follow, and then act like a simple TERM with regard to the preceding expression. Be careful with parentheses:

```
# These evaluate exit before doing the print:
print($foo, exit);  # Obviously not what you want.
print $foo, exit;   # Nor is this.

# These do the print before evaluating exit:
(print $foo), exit; # This is what you want.
print($foo), exit;  # Or this.
print ($foo), exit; # Or even this.
```

Also note that

```
print ($foo & 255) + 1, "\n";
```

probably doesn't do what you expect at first glance. The parentheses enclose the argument list for `print` which is evaluated (printing the result of `$foo & 255`). Then one is added to the return value of `print` (usually 1). The result is something like this:

```
1 + 1, "\n";    # Obviously not what you meant.
```

To do what you meant properly, you must write:

```
print(($foo & 255) + 1, "\n");
```

See Section 48.2.11 [Named Unary Operators], page 804 for more discussion of this.

Also parsed as terms are the `do {}` and `eval {}` constructs, as well as subroutine and method calls, and the anonymous constructors `[]` and `{}`.

See also Section 48.2.29 [Quote and Quote-like Operators], page 818 toward the end of this section, as well as Section 48.2.33 [I/O Operators], page 844.

48.2.3 The Arrow Operator >>

`"->"` is an infix dereference operator, just as it is in C and C++. If the right side is either a `[...]`, `{...}`, or a `(...)` subscript, then the left side must be either a hard or symbolic reference to an array, a hash, or a subroutine respectively. (Or technically speaking, a location capable of holding a hard reference, if it's an array or hash reference being used for assignment.) See Section 63.1 [perlreftut NAME], page 1092 and Section 62.1 [perlref NAME], page 1077.

Otherwise, the right side is a method name or a simple scalar variable containing either the method name or a subroutine reference, and the left side must be either an object (a blessed reference) or a class name (that is, a package name). See Section 46.1 [perlobj NAME], page 769.

The dereferencing cases (as opposed to method-calling cases) are somewhat extended by the experimental `postderef` feature. For the details of that feature, consult Section 62.5 [perlref Postfix Dereference Syntax], page 1088.

48.2.4 Auto-increment and Auto-decrement

`"++"` and `"--"` work as in C. That is, if placed before a variable, they increment or decrement the variable by one before returning the value, and if placed after, increment or decrement after returning the value.

```
$i = 0;  $j = 0;
print $i++;  # prints 0
print ++$j;  # prints 1
```

Note that just as in C, Perl doesn't define **when** the variable is incremented or decremented. You just know it will be done sometime before or after the value is returned. This also means that modifying a variable twice in the same statement will lead to undefined behavior. Avoid statements like:

```
$i = $i ++;
print ++ $i + $i ++;
```

Perl will not guarantee what the result of the above statements is.

The auto-increment operator has a little extra builtin magic to it. If you increment a variable that is numeric, or that has ever been used in a numeric context, you get a normal increment. If, however, the variable has been used in only string contexts since it was set, and has a value that is not the empty string and matches the pattern /^[a-zA-Z]*[0-9]*\z/, the increment is done as a string, preserving each character within its range, with carry:

```
    print ++($foo = "99");      # prints "100"
    print ++($foo = "a0");      # prints "a1"
    print ++($foo = "Az");      # prints "Ba"
    print ++($foo = "zz");      # prints "aaa"
```

undef is always treated as numeric, and in particular is changed to 0 before incrementing (so that a post-increment of an undef value will return 0 rather than undef).

The auto-decrement operator is not magical.

48.2.5 Exponentiation

Binary "**" is the exponentiation operator. It binds even more tightly than unary minus, so -2**4 is -(2**4), not (-2)**4. (This is implemented using C's pow(3) function, which actually works on doubles internally.)

Note that certain exponentiation expressions are ill-defined: these include 0**0, 1**Inf, and Inf**0. Do not expect any particular results from these special cases, the results are platform-dependent.

48.2.6 Symbolic Unary Operators

Unary "!" performs logical negation, that is, "not". See also not for a lower precedence version of this.

Unary "-" performs arithmetic negation if the operand is numeric, including any string that looks like a number. If the operand is an identifier, a string consisting of a minus sign concatenated with the identifier is returned. Otherwise, if the string starts with a plus or minus, a string starting with the opposite sign is returned. One effect of these rules is that -bareword is equivalent to the string "-bareword". If, however, the string begins with a non-alphabetic character (excluding "+" or "-"), Perl will attempt to convert the string to a numeric, and the arithmetic negation is performed. If the string cannot be cleanly converted to a numeric, Perl will give the warning **Argument "the string" isn't numeric in negation (-) at ...**.

Unary "~" performs bitwise negation, that is, 1's complement. For example, 0666 & ~027 is 0640. (See also Section 48.2.37 [Integer Arithmetic], page 849 and Section 48.2.36 [Bitwise String Operators], page 848.) Note that the width of the result is platform-dependent: ~0 is 32 bits wide on a 32-bit platform, but 64 bits wide on a 64-bit platform, so if you are expecting a certain bit width, remember to use the "&" operator to mask off the excess bits.

When complementing strings, if all characters have ordinal values under 256, then their complements will, also. But if they do not, all characters will be in either 32- or 64-bit complements, depending on your architecture. So for example, ~"\x{3B1}" is "\x{FFFF_FC4E}" on 32-bit machines and "\x{FFFF_FFFF_FFFF_FC4E}" on 64-bit machines.

If the experimental "bitwise" feature is enabled via use feature 'bitwise', then unary "~" always treats its argument as a number, and an alternate form of the operator, "~.", always treats its argument as a string. So ~0 and ~"0" will both give 2**32-1 on 32-bit platforms, whereas ~.0 and ~."0" will both yield "\xff". This feature produces a warning unless you use no warnings 'experimental::bitwise'.

Unary "+" has no effect whatsoever, even on strings. It is useful syntactically for separating a function name from a parenthesized expression that would otherwise be interpreted as the complete list of function arguments. (See examples above under Terms and List Operators (Leftward).)

Unary "\" creates a reference to whatever follows it. See Section 63.1 [perlreftut NAME], page 1092 and Section 62.1 [perlref NAME], page 1077. Do not confuse this behavior with the behavior of backslash within a string, although both forms do convey the notion of protecting the next thing from interpolation.

48.2.7 Binding Operators

Binary "=~" binds a scalar expression to a pattern match. Certain operations search or modify the string $_ by default. This operator makes that kind of operation work on some other string. The right argument is a search pattern, substitution, or transliteration. The left argument is what is supposed to be searched, substituted, or transliterated instead of the default $_. When used in scalar context, the return value generally indicates the success of the operation. The exceptions are substitution (s///) and transliteration (y///) with the /r (non-destructive) option, which cause the **r**eturn value to be the result of the substitution. Behavior in list context depends on the particular operator. See Section 48.2.30 [Regexp Quote-Like Operators], page 823 for details and Section 68.1 [perlretut NAME], page 1131 for examples using these operators.

If the right argument is an expression rather than a search pattern, substitution, or transliteration, it is interpreted as a search pattern at run time. Note that this means that its contents will be interpolated twice, so

```
 '\\' =~ q'\\';
```

is not ok, as the regex engine will end up trying to compile the pattern \, which it will consider a syntax error.

Binary "!~" is just like "=~" except the return value is negated in the logical sense.

Binary "!~" with a non-destructive substitution (s///r) or transliteration (y///r) is a syntax error.

48.2.8 Multiplicative Operators

Binary "*" multiplies two numbers.

Binary "/" divides two numbers.

Binary "%" is the modulo operator, which computes the division remainder of its first argument with respect to its second argument. Given integer operands $m and $n: If $n is positive, then $m % $n is $m minus the largest multiple of $n less than or equal to $m. If $n is negative, then $m % $n is $m minus the smallest multiple of $n that is not less than $m (that is, the result will be less than or equal to zero). If the operands $m and $n are floating point values and the absolute value of $n (that is abs($n)) is less than (UV_MAX + 1), only the integer portion of $m and $n will be used in the operation (Note: here UV_MAX means the maximum of the unsigned integer type). If the absolute value of the right operand (abs($n)) is greater than or equal to (UV_MAX + 1), "%" computes the floating-point remainder $r in the equation ($r = $m - $i*$n) where $i is a certain integer that makes $r have the same sign as the right operand $n (**not** as the left operand $m like C function fmod()) and the absolute value less than that of $n. Note that when use integer is in scope, "%" gives you direct access to the modulo operator as implemented by your C compiler. This operator is not as well defined for negative operands, but it will execute faster.

Binary "x" is the repetition operator. In scalar context or if the left operand is not enclosed in parentheses, it returns a string consisting of the left operand repeated the number of times specified by the right operand. In list context, if the left operand is enclosed in parentheses or is a list formed by qw/*STRING*/, it repeats the list. If the right operand is zero or negative (raising a warning on negative), it returns an empty string or an empty list, depending on the context.

```
print ’-’ x 80;             # print row of dashes

print "\t" x ($tab/8), ’ ’ x ($tab%8);      # tab over

@ones = (1) x 80;           # a list of 80 1’s
@ones = (5) x @ones;        # set all elements to 5
```

48.2.9 Additive Operators

Binary "+" returns the sum of two numbers.

Binary "-" returns the difference of two numbers.

Binary "." concatenates two strings.

48.2.10 Shift Operators > >>>

Binary "<<" returns the value of its left argument shifted left by the number of bits specified by the right argument. Arguments should be integers. (See also Section 48.2.37 [Integer Arithmetic], page 849.)

Binary ">>" returns the value of its left argument shifted right by the number of bits specified by the right argument. Arguments should be integers. (See also Section 48.2.37 [Integer Arithmetic], page 849.)

Note that both << and >> in Perl are implemented directly using << and >> in C. If use integer (see Section 48.2.37 [Integer Arithmetic], page 849) is in force then signed C integers are used, else unsigned C integers are used. Either way, the implementation isn’t going to generate results larger than the size of the integer type Perl was built with (32 bits or 64 bits).

The result of overflowing the range of the integers is undefined because it is undefined also in C. In other words, using 32-bit integers, 1 << 32 is undefined. Shifting by a negative number of bits is also undefined.

If you get tired of being subject to your platform’s native integers, the use bigint pragma neatly sidesteps the issue altogether:

```
print 20 << 20;  # 20971520
print 20 << 40;  # 5120 on 32-bit machines,
                 # 21990232555520 on 64-bit machines
use bigint;
print 20 << 100; # 25353012004564588029934064107520
```

48.2.11 Named Unary Operators

The various named unary operators are treated as functions with one argument, with optional parentheses.

If any list operator (`print()`, etc.) or any unary operator (`chdir()`, etc.) is followed by a left parenthesis as the next token, the operator and arguments within parentheses are taken to be of highest precedence, just like a normal function call. For example, because named unary operators are higher precedence than `||`:

```
    chdir $foo    || die;       # (chdir $foo) || die
    chdir($foo)   || die;       # (chdir $foo) || die
    chdir ($foo)  || die;       # (chdir $foo) || die
    chdir +($foo) || die;       # (chdir $foo) || die
```

but, because `"*"` is higher precedence than named operators:

```
    chdir $foo * 20;    # chdir ($foo * 20)
    chdir($foo) * 20;   # (chdir $foo) * 20
    chdir ($foo) * 20;  # (chdir $foo) * 20
    chdir +($foo) * 20; # chdir ($foo * 20)

    rand 10 * 20;       # rand (10 * 20)
    rand(10) * 20;      # (rand 10) * 20
    rand (10) * 20;     # (rand 10) * 20
    rand +(10) * 20;    # rand (10 * 20)
```

Regarding precedence, the filetest operators, like `-f`, `-M`, etc. are treated like named unary operators, but they don't follow this functional parenthesis rule. That means, for example, that `-f($file).".bak"` is equivalent to `-f "$file.bak"`.

See also Section 48.2.2 [Terms and List Operators (Leftward)], page 799.

48.2.12 Relational Operators

Perl operators that return true or false generally return values that can be safely used as numbers. For example, the relational operators in this section and the equality operators in the next one return `1` for true and a special version of the defined empty string, `""`, which counts as a zero but is exempt from warnings about improper numeric conversions, just as `"0 but true"` is.

Binary `"<"` returns true if the left argument is numerically less than the right argument.

Binary `">"` returns true if the left argument is numerically greater than the right argument. `>>`

Binary `"<="` returns true if the left argument is numerically less than or equal to the right argument.

Binary `">="` returns true if the left argument is numerically greater than or equal to the right argument. = `>>`

Binary `"lt"` returns true if the left argument is stringwise less than the right argument.

Binary `"gt"` returns true if the left argument is stringwise greater than the right argument.

Binary `"le"` returns true if the left argument is stringwise less than or equal to the right argument.

Binary "ge" returns true if the left argument is stringwise greater than or equal to the right argument.

48.2.13 Equality Operators

Binary "==" returns true if the left argument is numerically equal to the right argument.

Binary "!=" returns true if the left argument is numerically not equal to the right argument.

Binary "<=>" returns -1, 0, or 1 depending on whether the left argument is numerically less than, equal to, or greater than the right argument. If your platform supports NaN's (not-a-numbers) as numeric values, using them with "<=>" returns undef. NaN is not "<", "==", ">", "<=" or ">=" anything (even NaN), so those 5 return false. NaN != NaN returns true, as does NaN != *anything else*. If your platform doesn't support NaN's then NaN is just a string with numeric value 0. >>

```
$ perl -le '$x = "NaN"; print "No NaN support here" if $x == $x'
$ perl -le '$x = "NaN"; print "NaN support here" if $x != $x'
```

(Note that the bigint, bigrat, and bignum pragmas all support "NaN".)

Binary "eq" returns true if the left argument is stringwise equal to the right argument.

Binary "ne" returns true if the left argument is stringwise not equal to the right argument.

Binary "cmp" returns -1, 0, or 1 depending on whether the left argument is stringwise less than, equal to, or greater than the right argument.

Binary "~~" does a smartmatch between its arguments. Smart matching is described in the next section.

"lt", "le", "ge", "gt" and "cmp" use the collation (sort) order specified by the current LC_COLLATE locale if a use locale form that includes collation is in effect. See Section 38.1 [perllocale NAME], page 701. Do not mix these with Unicode, only use them with legacy 8-bit locale encodings. The standard Unicode-Collate and Unicode-Collate-Locale modules offer much more powerful solutions to collation issues.

For case-insensitive comparisions, look at the [perlfunc fc], page 383 case-folding function, available in Perl v5.16 or later:

```
if ( fc($x) eq fc($y) ) { ... }
```

48.2.14 Smartmatch Operator

First available in Perl 5.10.1 (the 5.10.0 version behaved differently), binary ~~ does a "smartmatch" between its arguments. This is mostly used implicitly in the when construct described in Section 74.1 [perlsyn NAME], page 1249, although not all when clauses call the smartmatch operator. Unique among all of Perl's operators, the smartmatch operator can recurse. The smartmatch operator is [experimental], page 945 and its behavior is subject to change.

It is also unique in that all other Perl operators impose a context (usually string or numeric context) on their operands, autoconverting those operands to those imposed contexts. In contrast, smartmatch *infers* contexts from the actual types of its operands and uses that type information to select a suitable comparison mechanism.

The ~~ operator compares its operands "polymorphically", determining how to compare them according to their actual types (numeric, string, array, hash, etc.) Like the equality operators with which it shares the same precedence, ~~ returns 1 for true and "" for false. It is often best read aloud as "in", "inside of", or "is contained in", because the left operand is often looked for *inside* the right operand. That makes the order of the operands to the smartmatch operand often opposite that of the regular match operator. In other words, the "smaller" thing is usually placed in the left operand and the larger one in the right.

The behavior of a smartmatch depends on what type of things its arguments are, as determined by the following table. The first row of the table whose types apply determines the smartmatch behavior. Because what actually happens is mostly determined by the type of the second operand, the table is sorted on the right operand instead of on the left.

```
Left      Right      Description and pseudocode
===============================================================
Any       undef      check whether Any is undefined
               like: !defined Any

Any       Object     invoke ~~ overloading on Object, or die

Right operand is an ARRAY:

Left      Right      Description and pseudocode
===============================================================
ARRAY1    ARRAY2     recurse on paired elements of ARRAY1 and ARRAY2[2]
               like: (ARRAY1[0] ~~ ARRAY2[0])
                       && (ARRAY1[1] ~~ ARRAY2[1]) && ...
HASH      ARRAY      any ARRAY elements exist as HASH keys
               like: grep { exists HASH->{$_} } ARRAY
Regexp    ARRAY      any ARRAY elements pattern match Regexp
               like: grep { /Regexp/ } ARRAY
undef     ARRAY      undef in ARRAY
               like: grep { !defined } ARRAY
Any       ARRAY      smartmatch each ARRAY element[3]
               like: grep { Any ~~ $_ } ARRAY

Right operand is a HASH:

Left      Right      Description and pseudocode
===============================================================
HASH1     HASH2      all same keys in both HASHes
               like: keys HASH1 ==
                        grep { exists HASH2->{$_} } keys HASH1
ARRAY     HASH       any ARRAY elements exist as HASH keys
               like: grep { exists HASH->{$_} } ARRAY
Regexp    HASH       any HASH keys pattern match Regexp
               like: grep { /Regexp/ } keys HASH
undef     HASH       always false (undef can't be a key)
```

```
               like: 0 == 1
Any       HASH       HASH key existence
               like: exists HASH->{Any}

Right operand is CODE:

Left      Right      Description and pseudocode
===============================================================
ARRAY     CODE       sub returns true on all ARRAY elements[1]
               like: !grep { !CODE->($_) } ARRAY
HASH      CODE       sub returns true on all HASH keys[1]
               like: !grep { !CODE->($_) } keys HASH
Any       CODE       sub passed Any returns true
               like: CODE->(Any)
```

Right operand is a Regexp:

```
Left      Right      Description and pseudocode
===============================================================
ARRAY     Regexp     any ARRAY elements match Regexp
               like: grep { /Regexp/ } ARRAY
HASH      Regexp     any HASH keys match Regexp
               like: grep { /Regexp/ } keys HASH
Any       Regexp     pattern match
               like: Any =~ /Regexp/

Other:

Left      Right      Description and pseudocode
===============================================================
Object    Any        invoke ~~ overloading on Object,
                     or fall back to...

Any       Num        numeric equality
                like: Any == Num
Num       nummy[4]    numeric equality
                like: Num == nummy
undef     Any        check whether undefined
                like: !defined(Any)
Any       Any        string equality
                like: Any eq Any
```

Notes:

1. Empty hashes or arrays match.
2. That is, each element smartmatches the element of the same index in the other array.[3]
3. If a circular reference is found, fall back to referential equality.
4. Either an actual number, or a string that looks like one.

The smartmatch implicitly dereferences any non-blessed hash or array reference, so the *HASH* and *ARRAY* entries apply in those cases. For blessed references, the *Object* entries apply. Smartmatches involving hashes only consider hash keys, never hash values.

The "like" code entry is not always an exact rendition. For example, the smartmatch operator short-circuits whenever possible, but `grep` does not. Also, `grep` in scalar context returns the number of matches, but `~~` returns only true or false.

Unlike most operators, the smartmatch operator knows to treat `undef` specially:

```
use v5.10.1;
@array = (1, 2, 3, undef, 4, 5);
say "some elements undefined" if undef ~~ @array;
```

Each operand is considered in a modified scalar context, the modification being that array and hash variables are passed by reference to the operator, which implicitly dereferences them. Both elements of each pair are the same:

```
use v5.10.1;

my %hash = (red    => 1, blue   => 2, green  => 3,
            orange => 4, yellow => 5, purple => 6,
            black  => 7, grey   => 8, white  => 9);

my @array = qw(red blue green);

say "some array elements in hash keys" if  @array ~~  %hash;
say "some array elements in hash keys" if \@array ~~ \%hash;

say "red in array" if "red" ~~  @array;
say "red in array" if "red" ~~ \@array;

say "some keys end in e" if /e$/ ~~  %hash;
say "some keys end in e" if /e$/ ~~ \%hash;
```

Two arrays smartmatch if each element in the first array smartmatches (that is, is "in") the corresponding element in the second array, recursively.

```
use v5.10.1;
my @little = qw(red blue green);
my @bigger = ("red", "blue", [ "orange", "green" ] );
if (@little ~~ @bigger) {  # true!
    say "little is contained in bigger";
}
```

Because the smartmatch operator recurses on nested arrays, this will still report that "red" is in the array.

```
use v5.10.1;
my @array = qw(red blue green);
my $nested_array = [[[[[[[ @array ]]]]]]];
say "red in array" if "red" ~~ $nested_array;
```

If two arrays smartmatch each other, then they are deep copies of each others' values, as this example reports:

```
use v5.12.0;
my @a = (0, 1, 2, [3, [4, 5], 6], 7);
my @b = (0, 1, 2, [3, [4, 5], 6], 7);

if (@a ~~ @b && @b ~~ @a) {
    say "a and b are deep copies of each other";
}
elsif (@a ~~ @b) {
    say "a smartmatches in b";
}
elsif (@b ~~ @a) {
    say "b smartmatches in a";
}
else {
    say "a and b don't smartmatch each other at all";
}
```

If you were to set `$b[3] = 4`, then instead of reporting that "a and b are deep copies of each other", it now reports that `"b smartmatches in a"`. That's because the corresponding position in `@a` contains an array that (eventually) has a 4 in it.

Smartmatching one hash against another reports whether both contain the same keys, no more and no less. This could be used to see whether two records have the same field names, without caring what values those fields might have. For example:

```
use v5.10.1;
sub make_dogtag {
    state $REQUIRED_FIELDS = { name=>1, rank=>1, serial_num=>1 };

    my ($class, $init_fields) = @_;

    die "Must supply (only) name, rank, and serial number"
        unless $init_fields ~~ $REQUIRED_FIELDS;

    ...
}
```

or, if other non-required fields are allowed, use ARRAY ~~ HASH:

```
use v5.10.1;
sub make_dogtag {
    state $REQUIRED_FIELDS = { name=>1, rank=>1, serial_num=>1 };

    my ($class, $init_fields) = @_;

    die "Must supply (at least) name, rank, and serial number"
```

```
            unless [keys %{$init_fields}] ~~ $REQUIRED_FIELDS;

        ...
    }
```

The smartmatch operator is most often used as the implicit operator of a `when` clause. See the section on "Switch Statements" in Section 74.1 [perlsyn NAME], page 1249.

48.2.14.1 Smartmatching of Objects

To avoid relying on an object's underlying representation, if the smartmatch's right operand is an object that doesn't overload `~~`, it raises the exception `"Smartmatching a non-overloaded object breaks encapsulation"`. That's because one has no business digging around to see whether something is "in" an object. These are all illegal on objects without a `~~` overload:

```
    %hash ~~ $object
       42 ~~ $object
   "fred" ~~ $object
```

However, you can change the way an object is smartmatched by overloading the `~~` operator. This is allowed to extend the usual smartmatch semantics. For objects that do have an `~~` overload, see `overload`.

Using an object as the left operand is allowed, although not very useful. Smartmatching rules take precedence over overloading, so even if the object in the left operand has smartmatch overloading, this will be ignored. A left operand that is a non-overloaded object falls back on a string or numeric comparison of whatever the `ref` operator returns. That means that

```
    $object ~~ X
```

does *not* invoke the overload method with *X* as an argument. Instead the above table is consulted as normal, and based on the type of *X*, overloading may or may not be invoked. For simple strings or numbers, "in" becomes equivalent to this:

```
    $object ~~ $number          ref($object) == $number
    $object ~~ $string          ref($object) eq $string
```

For example, this reports that the handle smells IOish (but please don't really do this!):

```
    use IO::Handle;
    my $fh = IO::Handle->new();
    if ($fh ~~ /\bIO\b/) {
        say "handle smells IOish";
    }
```

That's because it treats `$fh` as a string like `"IO::Handle=GLOB(0x8039e0)"`, then pattern matches against that.

48.2.15 Bitwise And

Binary "&" returns its operands ANDed together bit by bit. Although no warning is currently raised, the result is not well defined when this operation is performed on operands that aren't either numbers (see Section 48.2.37 [Integer Arithmetic], page 849) nor bitstrings (see Section 48.2.36 [Bitwise String Operators], page 848).

Note that "&" has lower priority than relational operators, so for example the parentheses are essential in a test like

```
print "Even\n" if ($x & 1) == 0;
```

If the experimental "bitwise" feature is enabled via `use feature 'bitwise'`, then this operator always treats its operand as numbers. This feature produces a warning unless you also use `no warnings 'experimental::bitwise'`.

48.2.16 Bitwise Or and Exclusive Or

Binary "|" returns its operands ORed together bit by bit.

Binary "^" returns its operands XORed together bit by bit.

Although no warning is currently raised, the results are not well defined when these operations are performed on operands that aren't either numbers (see Section 48.2.37 [Integer Arithmetic], page 849) nor bitstrings (see Section 48.2.36 [Bitwise String Operators], page 848).

Note that "|" and "^" have lower priority than relational operators, so for example the parentheses are essential in a test like

```
print "false\n" if (8 | 2) != 10;
```

If the experimental "bitwise" feature is enabled via `use feature 'bitwise'`, then this operator always treats its operand as numbers. This feature produces a warning unless you also use `no warnings 'experimental::bitwise'`.

48.2.17 C-style Logical And

Binary "&&" performs a short-circuit logical AND operation. That is, if the left operand is false, the right operand is not even evaluated. Scalar or list context propagates down to the right operand if it is evaluated.

48.2.18 C-style Logical Or

Binary "||" performs a short-circuit logical OR operation. That is, if the left operand is true, the right operand is not even evaluated. Scalar or list context propagates down to the right operand if it is evaluated.

48.2.19 Logical Defined-Or

Although it has no direct equivalent in C, Perl's `//` operator is related to its C-style "or". In fact, it's exactly the same as `||`, except that it tests the left hand side's definedness instead of its truth. Thus, `EXPR1 // EXPR2` returns the value of `EXPR1` if it's defined, otherwise, the value of `EXPR2` is returned. (`EXPR1` is evaluated in scalar context, `EXPR2` in the context of `//` itself). Usually, this is the same result as `defined(EXPR1) ? EXPR1 : EXPR2` (except that the ternary-operator form can be used as a lvalue, while `EXPR1 // EXPR2` cannot). This is very useful for providing default values for variables. If you actually want to test if at least one of `$x` and `$y` is defined, use `defined($x // $y)`.

The `||`, `//` and `&&` operators return the last value evaluated (unlike C's `||` and `&&`, which return 0 or 1). Thus, a reasonably portable way to find out the home directory might be:

```
$home =  $ENV{HOME}
      // $ENV{LOGDIR}
```

```
        // (getpwuid($<))[7]
        // die "You're homeless!\n";
```

In particular, this means that you shouldn't use this for selecting between two aggregates for assignment:

```
@a = @b || @c;              # this is wrong
@a = scalar(@b) || @c;      # really meant this
@a = @b ? @b : @c;          # this works fine, though
```

As alternatives to `&&` and `||` when used for control flow, Perl provides the `and` and `or` operators (see below). The short-circuit behavior is identical. The precedence of `"and"` and `"or"` is much lower, however, so that you can safely use them after a list operator without the need for parentheses:

```
unlink "alpha", "beta", "gamma"
        or gripe(), next LINE;
```

With the C-style operators that would have been written like this:

```
unlink("alpha", "beta", "gamma")
        || (gripe(), next LINE);
```

It would be even more readable to write that this way:

```
unless(unlink("alpha", "beta", "gamma")) {
    gripe();
    next LINE;
}
```

Using `"or"` for assignment is unlikely to do what you want; see below.

48.2.20 Range Operators

Binary `".."` is the range operator, which is really two different operators depending on the context. In list context, it returns a list of values counting (up by ones) from the left value to the right value. If the left value is greater than the right value then it returns the empty list. The range operator is useful for writing `foreach (1..10)` loops and for doing slice operations on arrays. In the current implementation, no temporary array is created when the range operator is used as the expression in `foreach` loops, but older versions of Perl might burn a lot of memory when you write something like this:

```
for (1 .. 1_000_000) {
    # code
}
```

The range operator also works on strings, using the magical auto-increment, see below.

In scalar context, `".."` returns a boolean value. The operator is bistable, like a flip-flop, and emulates the line-range (comma) operator of **sed**, **awk**, and various editors. Each `".."` operator maintains its own boolean state, even across calls to a subroutine that contains it. It is false as long as its left operand is false. Once the left operand is true, the range operator stays true until the right operand is true, *AFTER* which the range operator becomes false again. It doesn't become false till the next time the range operator is evaluated. It can test the right operand and become false on the same evaluation it became true (as in **awk**), but it still returns true once. If you don't want it to test the right operand until the next evaluation, as in **sed**, just use three dots ("...") instead of two. In all other regards, "..." behaves just like ".." does.

The right operand is not evaluated while the operator is in the "false" state, and the left operand is not evaluated while the operator is in the "true" state. The precedence is a little lower than || and &&. The value returned is either the empty string for false, or a sequence number (beginning with 1) for true. The sequence number is reset for each range encountered. The final sequence number in a range has the string "E0" appended to it, which doesn't affect its numeric value, but gives you something to search for if you want to exclude the endpoint. You can exclude the beginning point by waiting for the sequence number to be greater than 1.

If either operand of scalar ".." is a constant expression, that operand is considered true if it is equal (==) to the current input line number (the $. variable).

To be pedantic, the comparison is actually int(EXPR) == int(EXPR), but that is only an issue if you use a floating point expression; when implicitly using $. as described in the previous paragraph, the comparison is int(EXPR) == int($.) which is only an issue when $. is set to a floating point value and you are not reading from a file. Furthermore, "span" .. "spat" or 2.18 .. 3.14 will not do what you want in scalar context because each of the operands are evaluated using their integer representation.

Examples:

As a scalar operator:

```
if (101 .. 200) { print; } # print 2nd hundred lines, short for
                           #  if ($. == 101 .. $. == 200) { print; }

next LINE if (1 .. /^$/);  # skip header lines, short for
                           #   next LINE if ($. == 1 .. /^$/);
                           # (typically in a loop labeled LINE)

s/^/> / if (/^$/ .. eof());  # quote body

# parse mail messages
while (<>) {
    $in_header =   1  .. /^$/;
    $in_body   = /^$/ .. eof;
    if ($in_header) {
        # do something
    } else { # in body
        # do something else
    }
} continue {
    close ARGV if eof;             # reset $. each file
}
```

Here's a simple example to illustrate the difference between the two range operators:

```
@lines = ("   - Foo",
          "01 - Bar",
          "1  - Baz",
            "   - Quux");

foreach (@lines) {
    if (/0/ .. /1/) {
        print "$_\n";
    }
}
```

This program will print only the line containing "Bar". If the range operator is changed to ..., it will also print the "Baz" line.

And now some examples as a list operator:

```
for (101 .. 200) { print }      # print $_ 100 times
@foo = @foo[0 .. $#foo];        # an expensive no-op
@foo = @foo[$#foo-4 .. $#foo];  # slice last 5 items
```

The range operator (in list context) makes use of the magical auto-increment algorithm if the operands are strings. You can say

```
@alphabet = ("A" .. "Z");
```

to get all normal letters of the English alphabet, or

```
$hexdigit = (0 .. 9, "a" .. "f")[$num & 15];
```

to get a hexadecimal digit, or

```
@z2 = ("01" .. "31");
print $z2[$mday];
```

to get dates with leading zeros.

If the final value specified is not in the sequence that the magical increment would produce, the sequence goes until the next value would be longer than the final value specified.

If the initial value specified isn't part of a magical increment sequence (that is, a non-empty string matching `/^[a-zA-Z]*[0-9]*\z/`), only the initial value will be returned. So the following will only return an alpha:

```
use charnames "greek";
my @greek_small =  ("\N{alpha}" .. "\N{omega}");
```

To get the 25 traditional lowercase Greek letters, including both sigmas, you could use this instead:

```
use charnames "greek";
my @greek_small =  map { chr } ( ord("\N{alpha}")
                                        ..
                                   ord("\N{omega}")
                                 );
```

However, because there are *many* other lowercase Greek characters than just those, to match lowercase Greek characters in a regular expression, you could use the pattern `/(?:(?=\p{Greek})\p{Lower})+/` (or the Section 61.2.3.9 [experimental feature], page 1073 `/(?[ \p{Greek} & \p{Lower} ])+/`).

Because each operand is evaluated in integer form, `2.18 .. 3.14` will return two elements in list context.

```
@list = (2.18 .. 3.14); # same as @list = (2 .. 3);
```

48.2.21 Conditional Operator

Ternary "?:" is the conditional operator, just as in C. It works much like an if-then-else. If the argument before the ? is true, the argument before the : is returned, otherwise the argument after the : is returned. For example:

```
printf "I have %d dog%s.\n", $n,
        ($n == 1) ? "" : "s";
```

Scalar or list context propagates downward into the 2nd or 3rd argument, whichever is selected.

```
$x = $ok ? $y : $z;  # get a scalar
@x = $ok ? @y : @z;  # get an array
$x = $ok ? @y : @z;  # oops, that's just a count!
```

The operator may be assigned to if both the 2nd and 3rd arguments are legal lvalues (meaning that you can assign to them):

```
($x_or_y ? $x : $y) = $z;
```

Because this operator produces an assignable result, using assignments without parentheses will get you in trouble. For example, this:

```
$x % 2 ? $x += 10 : $x += 2
```

Really means this:

```
(($x % 2) ? ($x += 10) : $x) += 2
```

Rather than this:

```
($x % 2) ? ($x += 10) : ($x += 2)
```

That should probably be written more simply as:

```
$x += ($x % 2) ? 10 : 2;
```

48.2.22 Assignment Operators

> = >>>

"=" is the ordinary assignment operator.

Assignment operators work as in C. That is,

```
$x += 2;
```

is equivalent to

```
$x = $x + 2;
```

although without duplicating any side effects that dereferencing the lvalue might trigger, such as from tie(). Other assignment operators work similarly. The following are recognized:

```
**=    +=    *=    &=    &.=    <<=    &&=
       -=    /=    |=    |.=    >>=    ||=
       .=    %=    ^=    ^.=           //=
             x=
```

Although these are grouped by family, they all have the precedence of assignment. These combined assignment operators can only operate on scalars, whereas the ordinary assignment operator can assign to arrays, hashes, lists and even references. (See Section 11.2.3

["Context"], page 73 and Section 11.2.6 [perldata List value constructors], page 79, and Section 62.6 [perlref Assigning to References], page 1089.)

Unlike in C, the scalar assignment operator produces a valid lvalue. Modifying an assignment is equivalent to doing the assignment and then modifying the variable that was assigned to. This is useful for modifying a copy of something, like this:

```
($tmp = $global) =~ tr/13579/24680/;
```

Although as of 5.14, that can be also be accomplished this way:

```
use v5.14;
$tmp = ($global =~  tr/13579/24680/r);
```

Likewise,

```
($x += 2) *= 3;
```

is equivalent to

```
$x += 2;
$x *= 3;
```

Similarly, a list assignment in list context produces the list of lvalues assigned to, and a list assignment in scalar context returns the number of elements produced by the expression on the right hand side of the assignment.

The three dotted bitwise assignment operators (`&.= |.= ^.=`) are new in Perl 5.22 and experimental. See Section 48.2.36 [Bitwise String Operators], page 848.

48.2.23 Comma Operator

Binary "," is the comma operator. In scalar context it evaluates its left argument, throws that value away, then evaluates its right argument and returns that value. This is just like C's comma operator.

In list context, it's just the list argument separator, and inserts both its arguments into the list. These arguments are also evaluated from left to right.

The => operator (sometimes pronounced "fat comma") is a synonym for the comma except that it causes a word on its left to be interpreted as a string if it begins with a letter or underscore and is composed only of letters, digits and underscores. This includes operands that might otherwise be interpreted as operators, constants, single number v-strings or function calls. If in doubt about this behavior, the left operand can be quoted explicitly.

Otherwise, the => operator behaves exactly as the comma operator or list argument separator, according to context.

For example:

```
use constant FOO => "something";

my %h = ( FOO => 23 );
```

is equivalent to:

```
my %h = ("FOO", 23);
```

It is *NOT*:

```
my %h = ("something", 23);
```

The => operator is helpful in documenting the correspondence between keys and values in hashes, and other paired elements in lists.

```
%hash = ( $key => $value );
login( $username => $password );
```

The special quoting behavior ignores precedence, and hence may apply to *part* of the left operand:

```
print time.shift => "bbb";
```

That example prints something like "1314363215shiftbbb", because the => implicitly quotes the shift immediately on its left, ignoring the fact that time.shift is the entire left operand.

48.2.24 List Operators (Rightward)

On the right side of a list operator, the comma has very low precedence, such that it controls all comma-separated expressions found there. The only operators with lower precedence are the logical operators "and", "or", and "not", which may be used to evaluate calls to list operators without the need for parentheses:

```
open HANDLE, "< :utf8", "filename" or die "Can't open: $!\n";
```

However, some people find that code harder to read than writing it with parentheses:

```
open(HANDLE, "< :utf8", "filename") or die "Can't open: $!\n";
```

in which case you might as well just use the more customary "||" operator:

```
open(HANDLE, "< :utf8", "filename") || die "Can't open: $!\n";
```

See also discussion of list operators in Terms and List Operators (Leftward).

48.2.25 Logical Not

Unary "not" returns the logical negation of the expression to its right. It's the equivalent of "!" except for the very low precedence.

48.2.26 Logical And

Binary "and" returns the logical conjunction of the two surrounding expressions. It's equivalent to && except for the very low precedence. This means that it short-circuits: the right expression is evaluated only if the left expression is true.

48.2.27 Logical or and Exclusive Or

Binary "or" returns the logical disjunction of the two surrounding expressions. It's equivalent to || except for the very low precedence. This makes it useful for control flow:

```
print FH $data              or die "Can't write to FH: $!";
```

This means that it short-circuits: the right expression is evaluated only if the left expression is false. Due to its precedence, you must be careful to avoid using it as replacement for the || operator. It usually works out better for flow control than in assignments:

```
$x = $y or $z;              # bug: this is wrong
($x = $y) or $z;            # really means this
$x = $y || $z;              # better written this way
```

However, when it’s a list-context assignment and you’re trying to use `||` for control flow, you probably need `"or"` so that the assignment takes higher precedence.

```
@info = stat($file) || die;     # oops, scalar sense of stat!
@info = stat($file) or die;     # better, now @info gets its due
```

Then again, you could always use parentheses.

Binary `"xor"` returns the exclusive-OR of the two surrounding expressions. It cannot short-circuit (of course).

There is no low precedence operator for defined-OR.

48.2.28 C Operators Missing From Perl

Here is what C has that Perl doesn’t:

unary &

Address-of operator. (But see the `"\"` operator for taking a reference.)

unary *

Dereference-address operator. (Perl’s prefix dereferencing operators are typed: `$`, `@`, `%`, and `&`.)

(TYPE)

Type-casting operator.

48.2.29 Quote and Quote-like Operators

```
>
```

While we usually think of quotes as literal values, in Perl they function as operators, providing various kinds of interpolating and pattern matching capabilities. Perl provides customary quote characters for these behaviors, but also provides a way for you to choose your quote character for any of them. In the following table, a `{}` represents any pair of delimiters you choose.

Customary	Generic	Meaning	Interpolates
’’	q{}	Literal	no
""	qq{}	Literal	yes
‘‘	qx{}	Command	yes*
	qw{}	Word list	no
//	m{}	Pattern match	yes*
	qr{}	Pattern	yes*
	s{}{}	Substitution	yes*
	tr{}{}	Transliteration	no (but see below)
	y{}{}	Transliteration	no (but see below)
<<EOF		here-doc	yes*

```
* unless the delimiter is ’’.
```

Non-bracketing delimiters use the same character fore and aft, but the four sorts of ASCII brackets (round, angle, square, curly) all nest, which means that

```
q{foo{bar}baz}
```

is the same as

```
'foo{bar}baz'
```

Note, however, that this does not always work for quoting Perl code:

```
$s = q{ if($x eq "}") ... }; # WRONG
```

is a syntax error. The `Text-Balanced` module (standard as of v5.8, and from CPAN before then) is able to do this properly.

There can be whitespace between the operator and the quoting characters, except when `#` is being used as the quoting character. `q#foo#` is parsed as the string `foo`, while `q #foo#` is the operator `q` followed by a comment. Its argument will be taken from the next line. This allows you to write:

```
s {foo}  # Replace foo
  {bar}  # with bar.
```

The following escape sequences are available in constructs that interpolate, and in transliterations:

```
Sequence     Note  Description
\t                  tab               (HT, TAB)
\n                  newline           (NL)
\r                  return            (CR)
\f                  form feed         (FF)
\b                  backspace         (BS)
\a                  alarm (bell)      (BEL)
\e                  escape            (ESC)
\x{263A}     [1,8]  hex char          (example: SMILEY)
\x1b         [2,8]  restricted range hex char (example: ESC)
\N{name}     [3]    named Unicode character or character sequence
\N{U+263D}   [4,8]  Unicode character (example: FIRST QUARTER MOON)
\c[          [5]    control char      (example: chr(27))
\o{23072}    [6,8]  octal char        (example: SMILEY)
\033         [7,8]  restricted range octal char  (example: ESC)
```

[1]

The result is the character specified by the hexadecimal number between the braces. See [[8]], page 821 below for details on which character.

Only hexadecimal digits are valid between the braces. If an invalid character is encountered, a warning will be issued and the invalid character and all subsequent characters (valid or invalid) within the braces will be discarded.

If there are no valid digits between the braces, the generated character is the NULL character (`\x{00}`). However, an explicit empty brace (`\x{}`) will not cause a warning (currently).

[2]

The result is the character specified by the hexadecimal number in the range 0x00 to 0xFF. See [[8]], page 821 below for details on which character.

Only hexadecimal digits are valid following `\x`. When `\x` is followed by fewer than two valid digits, any valid digits will be zero-padded. This means that `\x7` will be interpreted as `\x07`, and a lone `"\x"` will be interpreted as `\x00`. Except at the end of a string, having fewer than two valid digits will result in a warning. Note that although the warning says the illegal character is ignored, it is only ignored as part of the escape and will still be used as the subsequent character in the string. For example:

```
  Original    Result    Warns?
  "\x7"       "\x07"    no
  "\x"        "\x00"    no
  "\x7q"      "\x07q"   yes
  "\xq"       "\x00q"   yes
```

[3]

The result is the Unicode character or character sequence given by *name*. See `charnames`.

[4]

`\N{U+`*`hexadecimal number`*`}` means the Unicode character whose Unicode code point is *hexadecimal number*.

[5]

The character following `\c` is mapped to some other character as shown in the table:

```
 Sequence   Value
   \c@      chr(0)
   \cA      chr(1)
   \ca      chr(1)
   \cB      chr(2)
   \cb      chr(2)
   ...
   \cZ      chr(26)
   \cz      chr(26)
   \c[      chr(27)
                     # See below for chr(28)
   \c]      chr(29)
   \c^      chr(30)
   \c_      chr(31)
   \c?      chr(127) # (on ASCII platforms; see below for link to
                     #  EBCDIC discussion)
```

In other words, it's the character whose code point has had 64 xor'd with its uppercase. `\c?` is DELETE on ASCII platforms because `ord("?") ^ 64` is 127, and `\c@` is NULL because the ord of `"@"` is 64, so xor'ing 64 itself produces 0.

Also, `\c\`*`X`* yields `chr(28) . "`*`X`*`"` for any *X*, but cannot come at the end of a string, because the backslash would be parsed as escaping the end quote.

On ASCII platforms, the resulting characters from the list above are the complete set of ASCII controls. This isn't the case on EBCDIC platforms; see

Section 19.7 [perlebcdic OPERATOR DIFFERENCES], page 286 for a full discussion of the differences between these for ASCII versus EBCDIC platforms.

Use of any other character following the "c" besides those listed above is discouraged, and as of Perl v5.20, the only characters actually allowed are the printable ASCII ones, minus the left brace "{". What happens for any of the allowed other characters is that the value is derived by xor'ing with the seventh bit, which is 64, and a warning raised if enabled. Using the non-allowed characters generates a fatal error.

To get platform independent controls, you can use \N{...}.

[6]

The result is the character specified by the octal number between the braces. See [[8]], page 821 below for details on which character.

If a character that isn't an octal digit is encountered, a warning is raised, and the value is based on the octal digits before it, discarding it and all following characters up to the closing brace. It is a fatal error if there are no octal digits at all.

[7]

The result is the character specified by the three-digit octal number in the range 000 to 777 (but best to not use above 077, see next paragraph). See [[8]], page 821 below for details on which character.

Some contexts allow 2 or even 1 digit, but any usage without exactly three digits, the first being a zero, may give unintended results. (For example, in a regular expression it may be confused with a backreference; see Section 60.2.3.7 [perlrebackslash Octal escapes], page 1049.) Starting in Perl 5.14, you may use \o{} instead, which avoids all these problems. Otherwise, it is best to use this construct only for ordinals \077 and below, remembering to pad to the left with zeros to make three digits. For larger ordinals, either use \o{}, or convert to something else, such as to hex and use \N{U+} (which is portable between platforms with different character sets) or \x{} instead.

[8]

Several constructs above specify a character by a number. That number gives the character's position in the character set encoding (indexed from 0). This is called synonymously its ordinal, code position, or code point. Perl works on platforms that have a native encoding currently of either ASCII/Latin1 or EBCDIC, each of which allow specification of 256 characters. In general, if the number is 255 (0xFF, 0377) or below, Perl interprets this in the platform's native encoding. If the number is 256 (0x100, 0400) or above, Perl interprets it as a Unicode code point and the result is the corresponding Unicode character. For example \x{50} and \o{120} both are the number 80 in decimal, which is less than 256, so the number is interpreted in the native character set encoding. In ASCII the character in the 80th position (indexed from 0) is the letter "P", and in EBCDIC it is the ampersand symbol "&". \x{100} and \o{400} are both 256 in decimal, so the number is interpreted as a Unicode code point no matter what the native encoding is. The name of the character in the 256th position (indexed by 0) in Unicode is LATIN CAPITAL LETTER A WITH MACRON.

There are a couple of exceptions to the above rule. `\N{U+`*hex number*`}` is always interpreted as a Unicode code point, so that `\N{U+0050}` is `"P"` even on EBCDIC platforms. And if `encoding` is in effect, the number is considered to be in that encoding, and is translated from that into the platform's native encoding if there is a corresponding native character; otherwise to Unicode.

NOTE: Unlike C and other languages, Perl has no `\v` escape sequence for the vertical tab (VT, which is 11 in both ASCII and EBCDIC), but you may use `\N{VT}`, `\ck`, `\N{U+0b}`, or `\x0b`. (`\v` does have meaning in regular expression patterns in Perl, see Section 58.1 [perlre NAME], page 989.)

The following escape sequences are available in constructs that interpolate, but not in transliterations.

```
    \l          lowercase next character only
    \u          titlecase (not uppercase!) next character only
    \L          lowercase all characters till \E or end of string
    \U          uppercase all characters till \E or end of string
    \F          foldcase all characters till \E or end of string
    \Q          quote (disable) pattern metacharacters till \E or
                end of string
    \E          end either case modification or quoted section
                (whichever was last seen)
```

See [perlfunc quotemeta], page 430 for the exact definition of characters that are quoted by `\Q`.

`\L`, `\U`, `\F`, and `\Q` can stack, in which case you need one `\E` for each. For example:

```
 say"This \Qquoting \ubusiness \Uhere isn't quite\E done yet,\E is it?";
 This quoting\ Business\ HERE\ ISN\'T\ QUITE\ done\ yet\, is it?
```

If a `use locale` form that includes `LC_CTYPE` is in effect (see Section 38.1 [perllocale NAME], page 701), the case map used by `\l`, `\L`, `\u`, and `\U` is taken from the current locale. If Unicode (for example, `\N{}` or code points of 0x100 or beyond) is being used, the case map used by `\l`, `\L`, `\u`, and `\U` is as defined by Unicode. That means that case-mapping a single character can sometimes produce a sequence of several characters. Under `use locale`, `\F` produces the same results as `\L` for all locales but a UTF-8 one, where it instead uses the Unicode definition.

All systems use the virtual `"\n"` to represent a line terminator, called a "newline". There is no such thing as an unvarying, physical newline character. It is only an illusion that the operating system, device drivers, C libraries, and Perl all conspire to preserve. Not all systems read `"\r"` as ASCII CR and `"\n"` as ASCII LF. For example, on the ancient Macs (pre-MacOS X) of yesteryear, these used to be reversed, and on systems without a line terminator, printing `"\n"` might emit no actual data. In general, use `"\n"` when you mean a "newline" for your system, but use the literal ASCII when you need an exact character. For example, most networking protocols expect and prefer a CR+LF (`"\015\012"` or `"\cM\cJ"`) for line terminators, and although they often accept just `"\012"`, they seldom tolerate just `"\015"`. If you get in the habit of using `"\n"` for networking, you may be burned some day.

For constructs that do interpolate, variables beginning with `"$"` or `"@"` are interpolated. Subscripted variables such as `$a[3]` or `$href->{key}[0]` are also interpolated, as are array and hash slices. But method calls such as `$obj->meth` are not.

Interpolating an array or slice interpolates the elements in order, separated by the value of `$"`, so is equivalent to interpolating `join $", @array`. "Punctuation" arrays such as `@*` are usually interpolated only if the name is enclosed in braces `@{*}`, but the arrays `@_`, `@+`, and `@-` are interpolated even without braces.

For double-quoted strings, the quoting from `\Q` is applied after interpolation and escapes are processed.

```
"abc\Qfoo\tbar$s\Exyz"
```

is equivalent to

```
"abc" . quotemeta("foo\tbar$s") . "xyz"
```

For the pattern of regex operators (`qr//`, `m//` and `s///`), the quoting from `\Q` is applied after interpolation is processed, but before escapes are processed. This allows the pattern to match literally (except for `$` and `@`). For example, the following matches:

```
'\s\t' =~ /\Q\s\t/
```

Because `$` or `@` trigger interpolation, you'll need to use something like `/\Quser\E\@\Qhost/` to match them literally.

Patterns are subject to an additional level of interpretation as a regular expression. This is done as a second pass, after variables are interpolated, so that regular expressions may be incorporated into the pattern from the variables. If this is not what you want, use `\Q` to interpolate a variable literally.

Apart from the behavior described above, Perl does not expand multiple levels of interpolation. In particular, contrary to the expectations of shell programmers, back-quotes do *NOT* interpolate within double quotes, nor do single quotes impede evaluation of variables when used within double quotes.

48.2.30 Regexp Quote-Like Operators

Here are the quote-like operators that apply to pattern matching and related activities.

`qr/`*`STRING`*`/msixpodualn`

This operator quotes (and possibly compiles) its *STRING* as a regular expression. *STRING* is interpolated the same way as *PATTERN* in `m/`*`PATTERN`*`/`. If `"'"` is used as the delimiter, no interpolation is done. Returns a Perl value which may be used instead of the corresponding `/`*`STRING`*`/msixpodualn` expression. The returned value is a normalized version of the original pattern. It magically differs from a string containing the same characters: `ref(qr/x/)` returns "Regexp"; however, dereferencing it is not well defined (you currently get the normalized version of the original pattern, but this may change).

For example,

```
$rex = qr/my.STRING/is;
print $rex;                 # prints (?si-xm:my.STRING)
s/$rex/foo/;
```

is equivalent to

```
s/my.STRING/foo/is;
```

The result may be used as a subpattern in a match:

```
$re = qr/$pattern/;
$string =~ /foo${re}bar/;   # can be interpolated in other
                            # patterns
$string =~ $re;             # or used standalone
$string =~ /$re/;           # or this way
```

Since Perl may compile the pattern at the moment of execution of the `qr()` operator, using `qr()` may have speed advantages in some situations, notably if the result of `qr()` is used standalone:

```
sub match {
    my $patterns = shift;
    my @compiled = map qr/$_/i, @$patterns;
    grep {
        my $success = 0;
        foreach my $pat (@compiled) {
            $success = 1, last if /$pat/;
        }
        $success;
    } @_;
}
```

Precompilation of the pattern into an internal representation at the moment of `qr()` avoids the need to recompile the pattern every time a match `/$pat/` is attempted. (Perl has many other internal optimizations, but none would be triggered in the above example if we did not use `qr()` operator.)

Options (specified by the following modifiers) are:

```
m   Treat string as multiple lines.
s   Treat string as single line. (Make . match a newline)
i   Do case-insensitive pattern matching.
x   Use extended regular expressions.
p   When matching preserve a copy of the matched string so
    that ${^PREMATCH}, ${^MATCH}, ${^POSTMATCH} will be
    defined (ignored starting in v5.20) as these are always
    defined starting in that relese
o   Compile pattern only once.
a   ASCII-restrict: Use ASCII for \d, \s, \w; specifying two
    a's further restricts things to that that no ASCII
    character will match a non-ASCII one under /i.
l   Use the current run-time locale's rules.
u   Use Unicode rules.
d   Use Unicode or native charset, as in 5.12 and earlier.
n   Non-capture mode. Don't let () fill in $1, $2, etc...
```

If a precompiled pattern is embedded in a larger pattern then the effect of `"msixpluadn"` will be propagated appropriately. The effect that the `/o` modifier has is not propagated, being restricted to those patterns explicitly using it.

The last four modifiers listed above, added in Perl 5.14, control the character set rules, but `/a` is the only one you are likely to want to specify explicitly; the other three are selected automatically by various pragmas.

See Section 58.1 [perlre NAME], page 989 for additional information on valid syntax for *STRING*, and for a detailed look at the semantics of regular expressions. In particular, all modifiers except the largely obsolete `/o` are further explained in Section 58.2.1 [perlre Modifiers], page 989. `/o` is described in the next section.

`m/`*PATTERN*`/msixpodualngc`
`/`*PATTERN*`/msixpodualngc`

Searches a string for a pattern match, and in scalar context returns true if it succeeds, false if it fails. If no string is specified via the `=~` or `!~` operator, the `$_` string is searched. (The string specified with `=~` need not be an lvalue–it may be the result of an expression evaluation, but remember the `=~` binds rather tightly.) See also Section 58.1 [perlre NAME], page 989.

Options are as described in `qr//` above; in addition, the following match process modifiers are available:

```
g  Match globally, i.e., find all occurrences.
c  Do not reset search position on a failed match when /g is
   in effect.
```

If `"/"` is the delimiter then the initial `m` is optional. With the `m` you can use any pair of non-whitespace (ASCII) characters as delimiters. This is particularly useful for matching path names that contain `"/"`, to avoid LTS (leaning toothpick syndrome). If `"?"` is the delimiter, then a match-only-once rule applies, described in `m?`*PATTERN*`?` below. If `"'"` (single quote) is the delimiter, no interpolation is performed on the *PATTERN*. When using a delimiter character valid in an identifier, whitespace is required after the `m`.

PATTERN may contain variables, which will be interpolated every time the pattern search is evaluated, except for when the delimiter is a single quote. (Note that `$(`, `$)`, and `$|` are not interpolated because they look like end-of-string tests.) Perl will not recompile the pattern unless an interpolated variable that it contains changes. You can force Perl to skip the test and never recompile by adding a `/o` (which stands for `"once"`) after the trailing delimiter. Once upon a time, Perl would recompile regular expressions unnecessarily, and this modifier was useful to tell it not to do so, in the interests of speed. But now, the only reasons to use `/o` are one of:

1. The variables are thousands of characters long and you know that they don't change, and you need to wring out the last little bit of speed by having Perl skip testing for that. (There is a maintenance penalty for doing this, as mentioning `/o` constitutes a promise that you won't change the variables in the pattern. If you do change them, Perl won't even notice.)
2. you want the pattern to use the initial values of the variables regardless of whether they change or not. (But there are saner ways of accomplishing this than using `/o`.)
3. If the pattern contains embedded code, such as

```
    use re 'eval';
    $code = 'foo(?{ $x })';
    /$code/
```

then perl will recompile each time, even though the pattern string hasn't changed, to ensure that the current value of `$x` is seen each time. Use `/o` if you want to avoid this.

The bottom line is that using `/o` is almost never a good idea.

The empty pattern `//`

If the *PATTERN* evaluates to the empty string, the last *successfully* matched regular expression is used instead. In this case, only the `g` and `c` flags on the empty pattern are honored; the other flags are taken from the original pattern. If no match has previously succeeded, this will (silently) act instead as a genuine empty pattern (which will always match).

Note that it's possible to confuse Perl into thinking `//` (the empty regex) is really `//` (the defined-or operator). Perl is usually pretty good about this, but some pathological cases might trigger this, such as `$x///` (is that `($x) / (//)` or `$x // /`?) and `print $fh //` (`print $fh(//` or `print($fh //`?). In all of these examples, Perl will assume you meant defined-or. If you meant the empty regex, just use parentheses or spaces to disambiguate, or even prefix the empty regex with an `m` (so `//` becomes `m//`).

Matching in list context

If the `/g` option is not used, `m//` in list context returns a list consisting of the subexpressions matched by the parentheses in the pattern, that is, (`$1`, `$2`, `$3`...) (Note that here `$1` etc. are also set). When there are no parentheses in the pattern, the return value is the list `(1)` for success. With or without parentheses, an empty list is returned upon failure.

Examples:

```
 open(TTY, "+</dev/tty")
    || die "can't access /dev/tty: $!";

 <TTY> =~ /^y/i && foo();       # do foo if desired

 if (/Version: *([0-9.]*)/) { $version = $1; }

 next if m#^/usr/spool/uucp#;

 # poor man's grep
 $arg = shift;
 while (<>) {
    print if /$arg/o; # compile only once (no longer needed!)
 }

 if (($F1, $F2, $Etc) = ($foo =~ /^(\S+)\s+(\S+)\s*(.*)/))
```

This last example splits `$foo` into the first two words and the remainder of the line, and assigns those three fields to `$F1`, `$F2`, and `$Etc`. The conditional is true if any variables were assigned; that is, if the pattern matched.

The `/g` modifier specifies global pattern matching–that is, matching as many times as possible within the string. How it behaves depends on the context. In list context, it returns a list of the substrings matched by any capturing parentheses in the regular expression. If there are no parentheses, it returns a list of all the matched strings, as if there were parentheses around the whole pattern.

In scalar context, each execution of `m//g` finds the next match, returning true if it matches, and false if there is no further match. The position after the last match can be read or set using the `pos()` function; see [perlfunc pos], page 428. A failed match normally resets the search position to the beginning of the string, but you can avoid that by adding the `/c` modifier (for example, `m//gc`). Modifying the target string also resets the search position.

`\G` *assertion*

You can intermix `m//g` matches with `m/\G.../g`, where `\G` is a zero-width assertion that matches the exact position where the previous `m//g`, if any, left off. Without the `/g` modifier, the `\G` assertion still anchors at `pos()` as it was at the start of the operation (see [perlfunc pos], page 428), but the match is of course only attempted once. Using `\G` without `/g` on a target string that has not previously had a `/g` match applied to it is the same as using the `\A` assertion to match the beginning of the string. Note also that, currently, `\G` is only properly supported when anchored at the very beginning of the pattern.

Examples:

```
    # list context
    ($one,$five,$fifteen) = (‘uptime‘ =˜ /(\d+\.\d+)/g);

    # scalar context
    local $/ = "";
    while ($paragraph = <>) {
        while ($paragraph =˜ /\p{Ll}[’")]*[.!?]+[’")]*\s/g) {
            $sentences++;
        }
    }
    say $sentences;
```

Here’s another way to check for sentences in a paragraph:

```
 my $sentence_rx = qr{
    (?: (?<= ^ ) | (?<= \s ) )  # after start-of-string or
                                # whitespace
    \p{Lu}                      # capital letter
    .*?                         # a bunch of anything
    (?<= \S )                   # that ends in non-
                                # whitespace
    (?<! \b [DMS]r  )           # but isn’t a common abbr.
```

```
   (?<! \b Mrs )
   (?<! \b Sra )
   (?<! \b St  )
   [.?!]                           # followed by a sentence
                                   # ender
   (?= $ | \s )                    # in front of end-of-string
                                   # or whitespace
}sx;
local $/ = "";
while (my $paragraph = <>) {
   say "NEW PARAGRAPH";
   my $count = 0;
   while ($paragraph =~ /($sentence_rx)/g) {
       printf "\tgot sentence %d: <%s>\n", ++$count, $1;
   }
}
```

Here’s how to use `m//gc` with `\G`:

```
$_ = "ppooqppqq";
while ($i++ < 2) {
    print "1: ’";
    print $1 while /(o)/gc; print "’, pos=", pos, "\n";
    print "2: ’";
    print $1 if /\G(q)/gc;  print "’, pos=", pos, "\n";
    print "3: ’";
    print $1 while /(p)/gc; print "’, pos=", pos, "\n";
}
print "Final: ’$1’, pos=",pos,"\n" if /\G(.)/;
```

The last example should print:

```
1: ’oo’, pos=4
2: ’q’, pos=5
3: ’pp’, pos=7
1: ’’, pos=7
2: ’q’, pos=8
3: ’’, pos=8
Final: ’q’, pos=8
```

Notice that the final match matched `q` instead of `p`, which a match without the `\G` anchor would have done. Also note that the final match did not update `pos`. `pos` is only updated on a `/g` match. If the final match did indeed match `p`, it’s a good bet that you’re running a very old (pre-5.6.0) version of Perl.

A useful idiom for `lex`-like scanners is `/\G.../gc`. You can combine several regexps like this to process a string part-by-part, doing different actions depending on which regexp matched. Each regexp tries to match where the previous one leaves off.

```
$_ = <<’EOL’;
   $url = URI::URL->new( "http://example.com/" );
   die if $url eq "xXx";
EOL

LOOP: {
    print(" digits"),       redo LOOP if /\G\d+\b[,.;]?\s*/gc;
    print(" lowercase"),    redo LOOP
                                   if /\G\p{Ll}+\b[,.;]?\s*/gc;
    print(" UPPERCASE"),    redo LOOP
                                   if /\G\p{Lu}+\b[,.;]?\s*/gc;
    print(" Capitalized"),  redo LOOP
                             if /\G\p{Lu}\p{Ll}+\b[,.;]?\s*/gc;
    print(" MiXeD"),        redo LOOP if /\G\pL+\b[,.;]?\s*/gc;
    print(" alphanumeric"), redo LOOP
                           if /\G[\p{Alpha}\pN]+\b[,.;]?\s*/gc;
    print(" line-noise"),   redo LOOP if /\G\W+/gc;
    print ". That's all!\n";
}
```

Here is the output (split into several lines):

```
line-noise lowercase line-noise UPPERCASE line-noise UPPERCASE
line-noise lowercase line-noise lowercase line-noise lowercase
lowercase line-noise lowercase lowercase line-noise lowercase
lowercase line-noise MiXeD line-noise. That's all!
```

m?*PATTERN*?msixpodualngc
?*PATTERN*?msixpodualngc

This is just like the m/*PATTERN*/ search, except that it matches only once between calls to the reset() operator. This is a useful optimization when you want to see only the first occurrence of something in each file of a set of files, for instance. Only m?? patterns local to the current package are reset.

```
while (<>) {
    if (m?^$?) {
                        # blank line between header and body
    }
} continue {
    reset if eof;       # clear m?? status for next file
}
```

Another example switched the first "latin1" encoding it finds to "utf8" in a pod file:

```
s//utf8/ if m? ^ =encoding \h+ \K latin1 ?x;
```

The match-once behavior is controlled by the match delimiter being ?; with any other delimiter this is the normal m// operator.

In the past, the leading m in m?*PATTERN*? was optional, but omitting it would produce a deprecation warning. As of v5.22.0, omitting it produces a syntax error. If you encounter this construct in older code, you can just add m.

`s/`*PATTERN*`/`*REPLACEMENT*`/msixpodualngcer`

Searches a string for a pattern, and if found, replaces that pattern with the replacement text and returns the number of substitutions made. Otherwise it returns false (specifically, the empty string).

If the `/r` (non-destructive) option is used then it runs the substitution on a copy of the string and instead of returning the number of substitutions, it returns the copy whether or not a substitution occurred. The original string is never changed when `/r` is used. The copy will always be a plain string, even if the input is an object or a tied variable.

If no string is specified via the `=~` or `!~` operator, the `$_` variable is searched and modified. Unless the `/r` option is used, the string specified must be a scalar variable, an array element, a hash element, or an assignment to one of those; that is, some sort of scalar lvalue.

If the delimiter chosen is a single quote, no interpolation is done on either the *PATTERN* or the *REPLACEMENT*. Otherwise, if the *PATTERN* contains a `$` that looks like a variable rather than an end-of-string test, the variable will be interpolated into the pattern at run-time. If you want the pattern compiled only once the first time the variable is interpolated, use the `/o` option. If the pattern evaluates to the empty string, the last successfully executed regular expression is used instead. See Section 58.1 [perlre NAME], page 989 for further explanation on these.

Options are as with `m//` with the addition of the following replacement specific options:

```
e   Evaluate the right side as an expression.
ee  Evaluate the right side as a string then eval the
    result.
r   Return substitution and leave the original string
    untouched.
```

Any non-whitespace delimiter may replace the slashes. Add space after the `s` when using a character allowed in identifiers. If single quotes are used, no interpretation is done on the replacement string (the `/e` modifier overrides this, however). Note that Perl treats backticks as normal delimiters; the replacement text is not evaluated as a command. If the *PATTERN* is delimited by bracketing quotes, the *REPLACEMENT* has its own pair of quotes, which may or may not be bracketing quotes, for example, `s(foo)(bar)` or `s<foo>/bar/`. A `/e` will cause the replacement portion to be treated as a full-fledged Perl expression and evaluated right then and there. It is, however, syntax checked at compile-time. A second `e` modifier will cause the replacement portion to be `eval`ed before being run as a Perl expression.

Examples:

```
s/\bgreen\b/mauve/g;              # don't change wintergreen

$path =~ s|/usr/bin|/usr/local/bin|;

s/Login: $foo/Login: $bar/; # run-time pattern
```

```
($foo = $bar) =~ s/this/that/;      # copy first, then
                                    # change
($foo = "$bar") =~ s/this/that/;    # convert to string,
                                    # copy, then change
$foo = $bar =~ s/this/that/r;       # Same as above using /r
$foo = $bar =~ s/this/that/r
            =~ s/that/the other/r;  # Chained substitutes
                                    # using /r
@foo = map { s/this/that/r } @bar   # /r is very useful in
                                    # maps

$count = ($paragraph =~ s/Mister\b/Mr./g);  # get change-cnt

$_ = ’abc123xyz’;
s/\d+/$&*2/e;               # yields ’abc246xyz’
s/\d+/sprintf("%5d",$&)/e;  # yields ’abc  246xyz’
s/\w/$& x 2/eg;             # yields ’aabbcc  224466xxyyzz’

s/%(.)/$percent{$1}/g;      # change percent escapes; no /e
s/%(.)/$percent{$1} || $&/ge;       # expr now, so /e
s/^=(\w+)/pod($1)/ge;       # use function call

$_ = ’abc123xyz’;
$x = s/abc/def/r;           # $x is ’def123xyz’ and
                            # $_ remains ’abc123xyz’.

# expand variables in $_, but dynamics only, using
# symbolic dereferencing
s/\$(\w+)/${$1}/g;

# Add one to the value of any numbers in the string
s/(\d+)/1 + $1/eg;

# Titlecase words in the last 30 characters only
substr($str, -30) =~ s/\b(\p{Alpha}+)\b/\u\L$1/g;

# This will expand any embedded scalar variable
# (including lexicals) in $_ : First $1 is interpolated
# to the variable name, and then evaluated
s/(\$\w+)/$1/eeg;

# Delete (most) C comments.
$program =~ s {
    /\*     # Match the opening delimiter.
    .*?     # Match a minimal number of characters.
    \*/     # Match the closing delimiter.
} []gsx;

s/^\s*(.*?)\s*$/$1/;        # trim whitespace in $_,
                            # expensively

for ($variable) {           # trim whitespace in $variable,
                            # cheap
    s/^\s+//;
    s/\s+$//;
}

s/([^ ]*) *([^ ]*)/$2 $1/;  # reverse 1st two fields
```

Note the use of $ instead of \ in the last example. Unlike **sed**, we use the \<*digit*> form only in the left hand side. Anywhere else it's $<*digit*>.

Occasionally, you can't use just a /g to get all the changes to occur that you might want. Here are two common cases:

```
# put commas in the right places in an integer
1 while s/(\d)(\d\d\d)(?!\d)/$1,$2/g;

# expand tabs to 8-column spacing
1 while s/\t+/’ ’ x (length($&)*8 - length($‘)%8)/e;
```

48.2.31 Quote-Like Operators

q/*STRING/*
’*STRING*’

A single-quoted, literal string. A backslash represents a backslash unless followed by the delimiter or another backslash, in which case the delimiter or backslash is interpolated.

```
$foo = q!I said, "You said, ’She said it.’"!;
$bar = q(’This is it.’);
$baz = ’\n’;                # a two-character string
```

qq/*STRING/*
"*STRING*"

A double-quoted, interpolated string.

```
$_ .= qq
 (*** The previous line contains the naughty word "$1".\n)
            if /\b(tcl|java|python)\b/i;      # :-)
$baz = "\n";                # a one-character string
```

qx/*STRING/*
‘*STRING*‘

A string which is (possibly) interpolated and then executed as a system command with /bin/sh or its equivalent. Shell wildcards, pipes, and redirections will be honored. The collected standard output of the command is returned; standard error is unaffected. In scalar context, it comes back as a single (potentially multi-line) string, or undef if the command failed. In list context, returns a list of lines (however you've defined lines with `$/` or `$INPUT_RECORD_SEPARATOR`), or an empty list if the command failed.

Because backticks do not affect standard error, use shell file descriptor syntax (assuming the shell supports this) if you care to address this. To capture a command's STDERR and STDOUT together:

```
    $output = `cmd 2>&1`;
```

To capture a command's STDOUT but discard its STDERR:

```
    $output = `cmd 2>/dev/null`;
```

To capture a command's STDERR but discard its STDOUT (ordering is important here):

```
    $output = `cmd 2>&1 1>/dev/null`;
```

To exchange a command's STDOUT and STDERR in order to capture the STDERR but leave its STDOUT to come out the old STDERR:

```
    $output = `cmd 3>&1 1>&2 2>&3 3>&-`;
```

To read both a command's STDOUT and its STDERR separately, it's easiest to redirect them separately to files, and then read from those files when the program is done:

```
    system("program args 1>program.stdout 2>program.stderr");
```

The STDIN filehandle used by the command is inherited from Perl's STDIN. For example:

```
    open(SPLAT, "stuff")   || die "can't open stuff: $!";
    open(STDIN, "<&SPLAT") || die "can't dupe SPLAT: $!";
    print STDOUT `sort`;
```

will print the sorted contents of the file named `"stuff"`.

Using single-quote as a delimiter protects the command from Perl's double-quote interpolation, passing it on to the shell instead:

```
    $perl_info  = qx(ps $$);            # that's Perl's $$
    $shell_info = qx'ps $$';            # that's the new shell's $$
```

How that string gets evaluated is entirely subject to the command interpreter on your system. On most platforms, you will have to protect shell metacharacters if you want them treated literally. This is in practice difficult to do, as it's unclear how to escape which characters. See Section 70.1 [perlsec NAME], page 1198 for a clean and safe example of a manual `fork()` and `exec()` to emulate backticks safely.

On some platforms (notably DOS-like ones), the shell may not be capable of dealing with multiline commands, so putting newlines in the string may not get you what you want. You may be able to evaluate multiple commands in a single line by separating them with the command separator character, if your shell supports that (for example, `;` on many Unix shells and `&` on the Windows NT `cmd` shell).

Perl will attempt to flush all files opened for output before starting the child process, but this may not be supported on some platforms (see Section 56.1

[perlport NAME], page 951). To be safe, you may need to set `$|` (`$AUTOFLUSH` in `English`) or call the `autoflush()` method of `IO-Handle` on any open handles.

Beware that some command shells may place restrictions on the length of the command line. You must ensure your strings don't exceed this limit after any necessary interpolations. See the platform-specific release notes for more details about your particular environment.

Using this operator can lead to programs that are difficult to port, because the shell commands called vary between systems, and may in fact not be present at all. As one example, the `type` command under the POSIX shell is very different from the `type` command under DOS. That doesn't mean you should go out of your way to avoid backticks when they're the right way to get something done. Perl was made to be a glue language, and one of the things it glues together is commands. Just understand what you're getting yourself into.

See Section 48.2.33 [I/O Operators], page 844 for more discussion.

`qw/`*`STRING`*`/`

Evaluates to a list of the words extracted out of *STRING*, using embedded whitespace as the word delimiters. It can be understood as being roughly equivalent to:

```
split(" ", q/STRING/);
```

the differences being that it generates a real list at compile time, and in scalar context it returns the last element in the list. So this expression:

```
qw(foo bar baz)
```

is semantically equivalent to the list:

```
"foo", "bar", "baz"
```

Some frequently seen examples:

```
use POSIX qw( setlocale localeconv )
@EXPORT = qw( foo bar baz );
```

A common mistake is to try to separate the words with commas or to put comments into a multi-line `qw`-string. For this reason, the `use warnings` pragma and the **-w** switch (that is, the `$^W` variable) produces warnings if the *STRING* contains the `","` or the `"#"` character.

`tr/`*`SEARCHLIST`*`/`*`REPLACEMENTLIST`*`/cdsr`
`y/`*`SEARCHLIST`*`/`*`REPLACEMENTLIST`*`/cdsr`

Transliterates all occurrences of the characters found in the search list with the corresponding character in the replacement list. It returns the number of characters replaced or deleted. If no string is specified via the `=~` or `!~` operator, the `$_` string is transliterated.

If the `/r` (non-destructive) option is present, a new copy of the string is made and its characters transliterated, and this copy is returned no matter whether it was modified or not: the original string is always left unchanged. The new copy is always a plain string, even if the input string is an object or a tied variable.

Unless the `/r` option is used, the string specified with `=~` must be a scalar variable, an array element, a hash element, or an assignment to one of those; in other words, an lvalue.

A character range may be specified with a hyphen, so `tr/A-J/0-9/` does the same replacement as `tr/ACEGIBDFHJ/0246813579/`. For **sed** devotees, `y` is provided as a synonym for `tr`. If the *SEARCHLIST* is delimited by bracketing quotes, the *REPLACEMENTLIST* has its own pair of quotes, which may or may not be bracketing quotes; for example, `tr[aeiouy][yuoiea]` or `tr(+\-*/)/ABCD/`.

Characters may be literals or any of the escape sequences accepted in double-quoted strings. But there is no interpolation, so `"$"` and `"@"` are treated as literals. A hyphen at the beginning or end, or preceded by a backslash is considered a literal. Escape sequence details are in Section 48.2.29 [the table near the beginning of this section], page 818. It is a bug in Perl v5.22 that something like

```
tr/\N{U+20}-\N{U+7E}foobar//
```

does not treat that range as fully Unicode.

Note that `tr` does **not** do regular expression character classes such as `\d` or `\pL`. The `tr` operator is not equivalent to the `tr(1)` utility. If you want to map strings between lower/upper cases, see [perlfunc lc], page 399 and [perlfunc uc], page 476, and in general consider using the `s` operator if you need regular expressions. The `\U`, `\u`, `\L`, and `\l` string-interpolation escapes on the right side of a substitution operator will perform correct case-mappings, but `tr[a-z][A-Z]` will not (except sometimes on legacy 7-bit data).

Note also that the whole range idea is rather unportable between character sets–and even within character sets they may cause results you probably didn't expect. A sound principle is to use only ranges that begin from and end at either alphabets of equal case (a-e, A-E), or digits (0-4). Anything else is unsafe. If in doubt, spell out the character sets in full.

Options:

```
    c   Complement the SEARCHLIST.
    d   Delete found but unreplaced characters.
    s   Squash duplicate replaced characters.
    r   Return the modified string and leave the original string
        untouched.
```

If the `/c` modifier is specified, the *SEARCHLIST* character set is complemented. If the `/d` modifier is specified, any characters specified by *SEARCHLIST* not found in *REPLACEMENTLIST* are deleted. (Note that this is slightly more flexible than the behavior of some **tr** programs, which delete anything they find in the *SEARCHLIST*, period.) If the `/s` modifier is specified, sequences of characters that were transliterated to the same character are squashed down to a single instance of the character.

If the `/d` modifier is used, the *REPLACEMENTLIST* is always interpreted exactly as specified. Otherwise, if the *REPLACEMENTLIST* is shorter than the *SEARCHLIST*, the final character is replicated till it is long enough. If the *REPLACEMENTLIST* is empty, the *SEARCHLIST* is replicated. This latter is useful for counting characters in a class or for squashing character sequences in a class.

Examples:

```
    $ARGV[1] =~ tr/A-Z/a-z/;    # canonicalize to lower case ASCII

    $cnt = tr/*/*/;             # count the stars in $_

    $cnt = $sky =~ tr/*/*/;     # count the stars in $sky

    $cnt = tr/0-9//;            # count the digits in $_

    tr/a-zA-Z//s;               # bookkeeper -> bokeper

    ($HOST = $host) =~ tr/a-z/A-Z/;
     $HOST = $host  =~ tr/a-z/A-Z/r;   # same thing

    $HOST = $host =~ tr/a-z/A-Z/r    # chained with s///r
                  =~ s/:/ -p/r;

    tr/a-zA-Z/ /cs;             # change non-alphas to single space

    @stripped = map tr/a-zA-Z/ /csr, @original;
                                # /r with map

    tr [\200-\377]
       [\000-\177];             # wickedly delete 8th bit
```

If multiple transliterations are given for a character, only the first one is used:

```
    tr/AAA/XYZ/
```

will transliterate any A to X.

Because the transliteration table is built at compile time, neither the *SEARCHLIST* nor the *REPLACEMENTLIST* are subjected to double quote interpolation. That means that if you want to use variables, you must use an `eval()`:

```
    eval "tr/$oldlist/$newlist/";
    die $@ if $@;

    eval "tr/$oldlist/$newlist/, 1" or die $@;
```

<<*EOF* >

A line-oriented form of quoting is based on the shell "here-document" syntax. Following a `<<` you specify a string to terminate the quoted material, and all lines following the current line down to the terminating string are the value of the item.

The terminating string may be either an identifier (a word), or some quoted text. An unquoted identifier works like double quotes. There may not be a space between the `<<` and the identifier, unless the identifier is explicitly quoted. (If you put a space it will be treated as a null identifier, which is valid, and matches the first empty line.) The terminating string must appear by itself (unquoted and with no surrounding whitespace) on the terminating line.

If the terminating string is quoted, the type of quotes used determine the treatment of the text.

Double Quotes

Double quotes indicate that the text will be interpolated using exactly the same rules as normal double quoted strings.

```
   print <<EOF;
The price is $Price.
EOF

   print << "EOF"; # same as above
The price is $Price.
EOF
```

Single Quotes

Single quotes indicate the text is to be treated literally with no interpolation of its content. This is similar to single quoted strings except that backslashes have no special meaning, with `\\` being treated as two backslashes and not one as they would in every other quoting construct.

Just as in the shell, a backslashed bareword following the `<<` means the same thing as a single-quoted string does:

```
    $cost = <<’VISTA’;  # hasta la ...
That’ll be $10 please, ma’am.
VISTA

    $cost = <<\VISTA;   # Same thing!
That’ll be $10 please, ma’am.
VISTA
```

This is the only form of quoting in perl where there is no need to worry about escaping content, something that code generators can and do make good use of.

Backticks

The content of the here doc is treated just as it would be if the string were embedded in backticks. Thus the content is interpolated as though it were double quoted and then executed via the shell, with the results of the execution returned.

```
   print << ‘EOC‘; # execute command and get results
echo hi there
EOC
```

It is possible to stack multiple here-docs in a row:

```
   print <<"foo", <<"bar"; # you can stack them
I said foo.
foo
I said bar.
bar
```

```
   myfunc(<< "THIS", 23, <<'THAT');
Here's a line
or two.
THIS
and here's another.
THAT
```

Just don't forget that you have to put a semicolon on the end to finish the statement, as Perl doesn't know you're not going to try to do this:

```
   print <<ABC
179231
ABC
   + 20;
```

If you want to remove the line terminator from your here-docs, use `chomp()`.

```
chomp($string = <<'END');
This is a string.
END
```

If you want your here-docs to be indented with the rest of the code, you'll need to remove leading whitespace from each line manually:

```
($quote = <<'FINIS') =~ s/^\s+//gm;
   The Road goes ever on and on,
   down from the door where it began.
FINIS
```

If you use a here-doc within a delimited construct, such as in `s///eg`, the quoted material must still come on the line following the `<<FOO` marker, which means it may be inside the delimited construct:

```
s/this/<<E . 'that'
the other
E
 . 'more '/eg;
```

It works this way as of Perl 5.18. Historically, it was inconsistent, and you would have to write

```
s/this/<<E . 'that'
 . 'more '/eg;
the other
E
```

outside of string evals.

Additionally, quoting rules for the end-of-string identifier are unrelated to Perl's quoting rules. `q()`, `qq()`, and the like are not supported in place of `''` and `""`, and the only interpolation is for backslashing the quoting character:

```
print << "abc\"def";
testing...
abc"def
```

Finally, quoted strings cannot span multiple lines. The general rule is that the identifier must be a string literal. Stick with that, and you should be safe.

48.2.32 Gory details of parsing quoted constructs

When presented with something that might have several different interpretations, Perl uses the **DWIM** (that's "Do What I Mean") principle to pick the most probable interpretation. This strategy is so successful that Perl programmers often do not suspect the ambivalence of what they write. But from time to time, Perl's notions differ substantially from what the author honestly meant.

This section hopes to clarify how Perl handles quoted constructs. Although the most common reason to learn this is to unravel labyrinthine regular expressions, because the initial steps of parsing are the same for all quoting operators, they are all discussed together.

The most important Perl parsing rule is the first one discussed below: when processing a quoted construct, Perl first finds the end of that construct, then interprets its contents. If you understand this rule, you may skip the rest of this section on the first reading. The other rules are likely to contradict the user's expectations much less frequently than this first one.

Some passes discussed below are performed concurrently, but because their results are the same, we consider them individually. For different quoting constructs, Perl performs different numbers of passes, from one to four, but these passes are always performed in the same order.

Finding the end

The first pass is finding the end of the quoted construct. This results in saving to a safe location a copy of the text (between the starting and ending delimiters), normalized as necessary to avoid needing to know what the original delimiters were.

If the construct is a here-doc, the ending delimiter is a line that has a terminating string as the content. Therefore `<<EOF` is terminated by `EOF` immediately followed by `"\n"` and starting from the first column of the terminating line. When searching for the terminating line of a here-doc, nothing is skipped. In other words, lines after the here-doc syntax are compared with the terminating string line by line.

For the constructs except here-docs, single characters are used as starting and ending delimiters. If the starting delimiter is an opening punctuation (that is `(`, `[`, `{`, or `<`), the ending delimiter is the corresponding closing punctuation (that is `)`, `]`, `}`, or `>`). If the starting delimiter is an unpaired character like `/` or a closing punctuation, the ending delimiter is the same as the starting delimiter. Therefore a `/` terminates a `qq//` construct, while a `]` terminates both `qq[]` and `qq]]` constructs.

When searching for single-character delimiters, escaped delimiters and `\\` are skipped. For example, while searching for terminating `/`, combinations of `\\` and `\/` are skipped. If the delimiters are bracketing, nested pairs are also skipped. For example, while searching for a closing `]` paired with the opening `[`, combinations of `\\`, `\]`, and `\[` are all skipped, and nested `[` and `]` are skipped as well. However, when backslashes are used as the delimiters (like `qq\\` and tr\\\), nothing is skipped. During the search for the end, backslashes that escape delimiters or other backslashes are removed (exactly speaking, they are not copied to the safe location).

For constructs with three-part delimiters (s///, y///, and tr///), the search is repeated once more. If the first delimiter is not an opening punctuation, the three delimiters must be the same, such as s!!! and tr))), in which case the second delimiter terminates the left part and starts the right part at once. If the left part is delimited by bracketing punctuation (that is (), [], {}, or <>), the right part needs another pair of delimiters such as s(){} and tr[]//. In these cases, whitespace and comments are allowed between the two parts, although the comment must follow at least one whitespace character; otherwise a character expected as the start of the comment may be regarded as the starting delimiter of the right part.

During this search no attention is paid to the semantics of the construct. Thus:

```
"$hash{"$foo/$bar"}"
```

or:

```
m/
  bar       # NOT a comment, this slash / terminated m//!
 /x
```

do not form legal quoted expressions. The quoted part ends on the first " and /, and the rest happens to be a syntax error. Because the slash that terminated m// was followed by a SPACE, the example above is not m//x, but rather m// with no /x modifier. So the embedded # is interpreted as a literal #.

Also no attention is paid to \c\ (multichar control char syntax) during this search. Thus the second \ in qq/\c\/ is interpreted as a part of \/, and the following / is not recognized as a delimiter. Instead, use \034 or \x1c at the end of quoted constructs.

Interpolation

The next step is interpolation in the text obtained, which is now delimiter-independent. There are multiple cases.

<<’EOF’

No interpolation is performed. Note that the combination \\ is left intact, since escaped delimiters are not available for here-docs.

m’’, the pattern of s’’’

No interpolation is performed at this stage. Any backslashed sequences including \\ are treated at the stage to [parsing regular expressions], page 843.

’’, q//, tr’’’, y’’’, the replacement of s’’’

The only interpolation is removal of \ from pairs of \\. Therefore "-" in tr’’’ and y’’’ is treated literally as a hyphen and no character range is available. \1 in the replacement of s’’’ does not work as $1.

`tr///`, `y///`

No variable interpolation occurs. String modifying combinations for case and quoting such as `\Q`, `\U`, and `\E` are not recognized. The other escape sequences such as `\200` and `\t` and backslashed characters such as `\\` and `\-` are converted to appropriate literals. The character `"-"` is treated specially and therefore `\-` is treated as a literal `"-"`.

`""`, `‘‘`, `qq//`, `qx//`, `<file*glob>`, `<<"EOF"`

`\Q`, `\U`, `\u`, `\L`, `\l`, `\F` (possibly paired with `\E`) are converted to corresponding Perl constructs. Thus, `"$foo\Qbaz$bar"` is converted to `$foo . (quotemeta("baz" . $bar))` internally. The other escape sequences such as `\200` and `\t` and backslashed characters such as `\\` and `\-` are replaced with appropriate expansions.

Let it be stressed that *whatever falls between* `\Q` *and* `\E` is interpolated in the usual way. Something like `"\Q\\E"` has no `\E` inside. Instead, it has `\Q`, `\\`, and `E`, so the result is the same as for `"\\\\E"`. As a general rule, backslashes between `\Q` and `\E` may lead to counterintuitive results. So, `"\Q\t\E"` is converted to `quotemeta("\t")`, which is the same as `"\\\t"` (since TAB is not alphanumeric). Note also that:

```
$str = ’\t’;
return "\Q$str";
```

may be closer to the conjectural *intention* of the writer of `"\Q\t\E"`.

Interpolated scalars and arrays are converted internally to the `join` and `"."` catenation operations. Thus, `"$foo XXX ’@arr’"` becomes:

```
$foo . " XXX ’" . (join $", @arr) . "’";
```

All operations above are performed simultaneously, left to right.

Because the result of `"\Q` *`STRING`* `\E"` has all metacharacters quoted, there is no way to insert a literal `$` or `@` inside a `\Q\E` pair. If protected by `\`, `$` will be quoted to become `"\\\$"`; if not, it is interpreted as the start of an interpolated scalar.

Note also that the interpolation code needs to make a decision on where the interpolated scalar ends. For instance, whether `"a $x -> {c}"` really means:

```
"a " . $x . " -> {c}";
```

or:

```
"a " . $x -> {c};
```

Most of the time, the longest possible text that does not include spaces between components and which contains matching braces or brackets. because the outcome may be determined by voting based on heuristic estimators, the result is not strictly predictable. Fortunately, it’s usually correct for ambiguous cases.

the replacement of `s///`

Processing of `\Q`, `\U`, `\u`, `\L`, `\l`, `\F` and interpolation happens as with `qq//` constructs.

It is at this step that `\1` is begrudgingly converted to `$1` in the replacement text of `s///`, in order to correct the incorrigible *sed* hackers who haven't picked up the saner idiom yet. A warning is emitted if the `use warnings` pragma or the **`-w`** command-line flag (that is, the `$^W` variable) was set.

`RE` in `?RE?`, `/RE/`, `m/RE/`, `s/RE/foo/`,

Processing of `\Q`, `\U`, `\u`, `\L`, `\l`, `\F`, `\E`, and interpolation happens (almost) as with `qq//` constructs.

Processing of `\N{...}` is also done here, and compiled into an intermediate form for the regex compiler. (This is because, as mentioned below, the regex compilation may be done at execution time, and `\N{...}` is a compile-time construct.)

However any other combinations of `\` followed by a character are not substituted but only skipped, in order to parse them as regular expressions at the following step. As `\c` is skipped at this step, `@` of `\c@` in RE is possibly treated as an array symbol (for example `@foo`), even though the same text in `qq//` gives interpolation of `\c@`.

Code blocks such as `(?{BLOCK})` are handled by temporarily passing control back to the perl parser, in a similar way that an interpolated array subscript expression such as `"foo$array[1+f("[xyz")]bar"` would be.

Moreover, inside `(?{BLOCK})`, `(?# comment )`, and a `#`-comment in a `/x`-regular expression, no processing is performed whatsoever. This is the first step at which the presence of the `/x` modifier is relevant.

Interpolation in patterns has several quirks: `$|`, `$(`, `$)`, `@+` and `@-` are not interpolated, and constructs `$var[SOMETHING]` are voted (by several different estimators) to be either an array element or `$var` followed by an RE alternative. This is where the notation `${arr[$bar]}` comes handy: `/${arr[0-9]}/` is interpreted as array element `-9`, not as a regular expression from the variable `$arr` followed by a digit, which would be the interpretation of `/$arr[0-9]/`. Since voting among different estimators may occur, the result is not predictable.

The lack of processing of `\\` creates specific restrictions on the post-processed text. If the delimiter is `/`, one cannot get the combination `\/` into the result of this step. `/` will finish the regular expression, `\/` will be stripped to `/` on the previous step, and `\\/` will be left as is. Because `/` is equivalent to `\/` inside a regular expression, this does not matter unless the delimiter happens to be character special to the RE engine, such as in `s*foo*bar*`, `m[foo]`, or `?foo?`; or an alphanumeric char, as in:

```
m m ^ a \s* b mmx;
```

In the RE above, which is intentionally obfuscated for illustration, the delimiter is `m`, the modifier is `mx`, and after delimiter-removal the RE is the same as for `m/ ^ a \s* b /mx`. There's more than one reason you're encouraged to restrict your delimiters to non-alphanumeric, non-whitespace choices.

This step is the last one for all constructs except regular expressions, which are processed further.

parsing regular expressions

Previous steps were performed during the compilation of Perl code, but this one happens at run time, although it may be optimized to be calculated at compile time if appropriate. After preprocessing described above, and possibly after evaluation if concatenation, joining, casing translation, or metaquoting are involved, the resulting *string* is passed to the RE engine for compilation.

Whatever happens in the RE engine might be better discussed in Section 58.1 [perlre NAME], page 989, but for the sake of continuity, we shall do so here.

This is another step where the presence of the `/x` modifier is relevant. The RE engine scans the string from left to right and converts it into a finite automaton.

Backslashed characters are either replaced with corresponding literal strings (as with `\{`), or else they generate special nodes in the finite automaton (as with `\b`). Characters special to the RE engine (such as `|`) generate corresponding nodes or groups of nodes. `(?#...)` comments are ignored. All the rest is either converted to literal strings to match, or else is ignored (as is whitespace and `#`-style comments if `/x` is present).

Parsing of the bracketed character class construct, `[...]`, is rather different than the rule used for the rest of the pattern. The terminator of this construct is found using the same rules as for finding the terminator of a `{}`-delimited construct, the only exception being that `]` immediately following `[` is treated as though preceded by a backslash.

The terminator of runtime `(?{...})` is found by temporarily switching control to the perl parser, which should stop at the point where the logically balancing terminating `}` is found.

It is possible to inspect both the string given to RE engine and the resulting finite automaton. See the arguments `debug`/`debugcolor` in the `use re` pragma, as well as Perl's **-Dr** command-line switch documented in Section 69.3.3 [perlrun Command Switches], page 1179.

Optimization of regular expressions

This step is listed for completeness only. Since it does not change semantics, details of this step are not documented and are subject to change without notice. This step is performed over the finite automaton that was generated during the previous pass.

It is at this stage that `split()` silently optimizes `/^/` to mean `/^/m`.

48.2.33 I/O Operators

>> >>

There are several I/O operators you should know about.

A string enclosed by backticks (grave accents) first undergoes double-quote interpolation. It is then interpreted as an external command, and the output of that command is the value of the backtick string, like in a shell. In scalar context, a single string consisting of all output is returned. In list context, a list of values is returned, one per line of output. (You can set `$/` to use a different line terminator.) The command is executed each time the pseudo-literal is evaluated. The status value of the command is returned in `$?` (see Section 86.1 [perlvar NAME], page 1375 for the interpretation of `$?`). Unlike in **csh**, no translation is done on the return data–newlines remain newlines. Unlike in any of the shells, single quotes do not hide variable names in the command from interpretation. To pass a literal dollar-sign through to the shell you need to hide it with a backslash. The generalized form of backticks is `qx//`. (Because backticks always undergo shell expansion as well, see Section 70.1 [perlsec NAME], page 1198 for security concerns.)

In scalar context, evaluating a filehandle in angle brackets yields the next line from that file (the newline, if any, included), or `undef` at end-of-file or on error. When `$/` is set to `undef` (sometimes known as file-slurp mode) and the file is empty, it returns `''` the first time, followed by `undef` subsequently.

Ordinarily you must assign the returned value to a variable, but there is one situation where an automatic assignment happens. If and only if the input symbol is the only thing inside the conditional of a `while` statement (even if disguised as a `for(;;)` loop), the value is automatically assigned to the global variable `$_`, destroying whatever was there previously. (This may seem like an odd thing to you, but you'll use the construct in almost every Perl script you write.) The `$_` variable is not implicitly localized. You'll have to put a `local $_;` before the loop if you want that to happen.

The following lines are equivalent:

```
while (defined($_ = <STDIN>)) { print; }
while ($_ = <STDIN>) { print; }
while (<STDIN>) { print; }
for (;<STDIN>;) { print; }
print while defined($_ = <STDIN>);
print while ($_ = <STDIN>);
print while <STDIN>;
```

This also behaves similarly, but assigns to a lexical variable instead of to `$_`:

```
while (my $line = <STDIN>) { print $line }
```

In these loop constructs, the assigned value (whether assignment is automatic or explicit) is then tested to see whether it is defined. The defined test avoids problems where the line has a string value that would be treated as false by Perl; for example a `""` or a `"0"` with no trailing newline. If you really mean for such values to terminate the loop, they should be tested for explicitly:

```
while (($_ = <STDIN>) ne '0') { ... }
while (<STDIN>) { last unless $_; ... }
```

In other boolean contexts, <*FILEHANDLE*> without an explicit `defined` test or comparison elicits a warning if the `use warnings` pragma or the `-w` command-line switch (the `$^W` variable) is in effect.

The filehandles STDIN, STDOUT, and STDERR are predefined. (The filehandles `stdin`, `stdout`, and `stderr` will also work except in packages, where they would be interpreted as local identifiers rather than global.) Additional filehandles may be created with the `open()` function, amongst others. See Section 49.1 [perlopentut NAME], page 852 and ⟨undefined⟩ [perlfunc open], page ⟨undefined⟩ for details on this.

If a <*FILEHANDLE*> is used in a context that is looking for a list, a list comprising all input lines is returned, one line per list element. It's easy to grow to a rather large data space this way, so use with care.

<*FILEHANDLE*> may also be spelled `readline(*`*FILEHANDLE*`)`. See [perlfunc readline], page 433.

The null filehandle `<>` is special: it can be used to emulate the behavior of **sed** and **awk**, and any other Unix filter program that takes a list of filenames, doing the same to each line of input from all of them. Input from `<>` comes either from standard input, or from each file listed on the command line. Here's how it works: the first time `<>` is evaluated, the `@ARGV` array is checked, and if it is empty, `$ARGV[0]` is set to `"-"`, which when opened gives you standard input. The `@ARGV` array is then processed as a list of filenames. The loop

```
while (<>) {
    ...                     # code for each line
}
```

is equivalent to the following Perl-like pseudo code:

```
unshift(@ARGV, '-') unless @ARGV;
while ($ARGV = shift) {
    open(ARGV, $ARGV);
    while (<ARGV>) {
        ...         # code for each line
    }
}
```

except that it isn't so cumbersome to say, and will actually work. It really does shift the `@ARGV` array and put the current filename into the `$ARGV` variable. It also uses filehandle *ARGV* internally. `<>` is just a synonym for `<ARGV>`, which is magical. (The pseudo code above doesn't work because it treats `<ARGV>` as non-magical.)

Since the null filehandle uses the two argument form of ⟨undefined⟩ [perlfunc open], page ⟨undefined⟩ it interprets special characters, so if you have a script like this:

```
while (<>) {
    print;
}
```

and call it with `perl dangerous.pl 'rm -rfv *|'`, it actually opens a pipe, executes the `rm` command and reads `rm`'s output from that pipe. If you want all items in `@ARGV` to be interpreted as file names, you can use the module `ARGV::readonly` from CPAN, or use the double bracket:

```
while (<<>>) {
        print;
    }
```

Using double angle brackets inside of a while causes the open to use the three argument form (with the second argument being `<`), so all arguments in ARGV are treated as literal filenames (including "-"). (Note that for convenience, if you use <<>> and if @ARGV is empty, it will still read from the standard input.)

You can modify @ARGV before the first <> as long as the array ends up containing the list of filenames you really want. Line numbers ($.) continue as though the input were one big happy file. See the example in [perlfunc eof], page 376 for how to reset line numbers on each file.

If you want to set @ARGV to your own list of files, go right ahead. This sets @ARGV to all plain text files if no @ARGV was given:

```
    @ARGV = grep { -f && -T } glob('*') unless @ARGV;
```

You can even set them to pipe commands. For example, this automatically filters compressed arguments through **gzip**:

```
    @ARGV = map { /\.(gz|Z)$/ ? "gzip -dc < $_ |" : $_ } @ARGV;
```

If you want to pass switches into your script, you can use one of the Getopts modules or put a loop on the front like this:

```
    while ($_ = $ARGV[0], /^-/) {
        shift;
        last if /^--$/;
        if (/^-D(.*)/) { $debug = $1 }
        if (/^-v/)     { $verbose++  }
        # ...           # other switches
    }

    while (<>) {
        # ...           # code for each line
    }
```

The <> symbol will return undef for end-of-file only once. If you call it again after this, it will assume you are processing another @ARGV list, and if you haven't set @ARGV, will read input from STDIN.

If what the angle brackets contain is a simple scalar variable (for example, $foo), then that variable contains the name of the filehandle to input from, or its typeglob, or a reference to the same. For example:

```
    $fh = \*STDIN;
    $line = <$fh>;
```

If what's within the angle brackets is neither a filehandle nor a simple scalar variable containing a filehandle name, typeglob, or typeglob reference, it is interpreted as a filename pattern to be globbed, and either a list of filenames or the next filename in the list is returned, depending on context. This distinction is determined on syntactic grounds alone. That means <$x> is always a readline() from an indirect handle, but <$hash{key}> is always a glob(). That's because $x is a simple scalar variable, but $hash{key} is not–it's a hash element. Even <$x > (note the extra space) is treated as glob("$x "), not readline($x).

One level of double-quote interpretation is done first, but you can't say <$foo> because that's an indirect filehandle as explained in the previous paragraph. (In older versions of Perl, programmers would insert curly brackets to force interpretation as a filename glob: <${foo}>. These days, it's considered cleaner to call the internal function directly as glob($foo), which is probably the right way to have done it in the first place.) For example:

```
while (<*.c>) {
    chmod 0644, $_;
}
```

is roughly equivalent to:

```
open(FOO, "echo *.c | tr -s ' \t\r\f' '\\012\\012\\012\\012'|");
while (<FOO>) {
    chomp;
    chmod 0644, $_;
}
```

except that the globbing is actually done internally using the standard File-Glob extension. Of course, the shortest way to do the above is:

```
chmod 0644, <*.c>;
```

A (file)glob evaluates its (embedded) argument only when it is starting a new list. All values must be read before it will start over. In list context, this isn't important because you automatically get them all anyway. However, in scalar context the operator returns the next value each time it's called, or undef when the list has run out. As with filehandle reads, an automatic defined is generated when the glob occurs in the test part of a while, because legal glob returns (for example, a file called 0) would otherwise terminate the loop. Again, undef is returned only once. So if you're expecting a single value from a glob, it is much better to say

```
($file) = <blurch*>;
```

than

```
$file = <blurch*>;
```

because the latter will alternate between returning a filename and returning false.

If you're trying to do variable interpolation, it's definitely better to use the glob() function, because the older notation can cause people to become confused with the indirect filehandle notation.

```
@files = glob("$dir/*.[ch]");
@files = glob($files[$i]);
```

48.2.34 Constant Folding

Like C, Perl does a certain amount of expression evaluation at compile time whenever it determines that all arguments to an operator are static and have no side effects. In particular, string concatenation happens at compile time between literals that don't do variable substitution. Backslash interpolation also happens at compile time. You can say

```
    'Now is the time for all'
. "\n"
.  'good men to come to.'
```

and this all reduces to one string internally. Likewise, if you say

```
foreach $file (@filenames) {
    if (-s $file > 5 + 100 * 2**16) {  }
}
```

the compiler precomputes the number which that expression represents so that the interpreter won't have to.

48.2.35 No-ops

Perl doesn't officially have a no-op operator, but the bare constants `0` and `1` are special-cased not to produce a warning in void context, so you can for example safely do

```
1 while foo();
```

48.2.36 Bitwise String Operators

Bitstrings of any size may be manipulated by the bitwise operators (`~ | & ^`).

If the operands to a binary bitwise op are strings of different sizes, `|` and `^` ops act as though the shorter operand had additional zero bits on the right, while the `&` op acts as though the longer operand were truncated to the length of the shorter. The granularity for such extension or truncation is one or more bytes.

```
# ASCII-based examples
print "j p \n" ^ " a h";            # prints "JAPH\n"
print "JA" | "  ph\n";              # prints "japh\n"
print "japh\nJunk" & '_____';       # prints "JAPH\n";
print 'p N$' ^ " E<H\n";            # prints "Perl\n";
```

If you are intending to manipulate bitstrings, be certain that you're supplying bitstrings: If an operand is a number, that will imply a **numeric** bitwise operation. You may explicitly show which type of operation you intend by using `""` or `0+`, as in the examples below.

```
$foo =  150  |  105;        # yields 255  (0x96 | 0x69 is 0xFF)
$foo = '150' |  105;        # yields 255
$foo =  150  | '105';       # yields 255
$foo = '150' | '105';       # yields string '155' (under ASCII)

$baz = 0+$foo & 0+$bar;     # both ops explicitly numeric
$biz = "$foo" ^ "$bar";     # both ops explicitly stringy
```

This somewhat unpredictable behavior can be avoided with the experimental "bitwise" feature, new in Perl 5.22. You can enable it via `use feature 'bitwise'`. By default, it will warn unless the `"experimental::bitwise"` warnings category has been disabled. (`use experimental 'bitwise'` will enable the feature and disable the warning.) Under this feature, the four standard bitwise operators (`~ | & ^`) are always numeric. Adding a dot after each operator (`~. |. &. ^.`) forces it to treat its operands as strings:

```
use experimental "bitwise";
$foo =  150  |  105;        # yields 255  (0x96 | 0x69 is 0xFF)
$foo = '150' |  105;        # yields 255
$foo =  150  | '105';       # yields 255
$foo = '150' | '105';       # yields 255
$foo =  150  |. 105;        # yields string '155'
```

```
$foo = '150' |. 105;        # yields string '155'
$foo =  150  |.'105';       # yields string '155'
$foo = '150' |.'105';       # yields string '155'

$baz = $foo &  $bar;        # both operands numeric
$biz = $foo ^. $bar;        # both operands stringy
```

The assignment variants of these operators (`&= |= ^= &.= |.= ^.=`) behave likewise under the feature.

The behavior of these operators is problematic (and subject to change) if either or both of the strings are encoded in UTF-8 (see Section 81.2.2 [perlunicode Byte and Character Semantics], page 1318.

See ⟨undefined⟩ [perlfunc vec], page ⟨undefined⟩ for information on how to manipulate individual bits in a bit vector.

48.2.37 Integer Arithmetic

By default, Perl assumes that it must do most of its arithmetic in floating point. But by saying

```
use integer;
```

you may tell the compiler to use integer operations (see `integer` for a detailed explanation) from here to the end of the enclosing BLOCK. An inner BLOCK may countermand this by saying

```
no integer;
```

which lasts until the end of that BLOCK. Note that this doesn't mean everything is an integer, merely that Perl will use integer operations for arithmetic, comparison, and bitwise operators. For example, even under `use integer`, if you take the `sqrt(2)`, you'll still get `1.4142135623731` or so.

Used on numbers, the bitwise operators (`& | ^ ~ << >>`) always produce integral results. (But see also Section 48.2.36 [Bitwise String Operators], page 848.) However, `use integer` still has meaning for them. By default, their results are interpreted as unsigned integers, but if `use integer` is in effect, their results are interpreted as signed integers. For example, `~0` usually evaluates to a large integral value. However, `use integer; ~0` is `-1` on two's-complement machines.

48.2.38 Floating-point Arithmetic

While `use integer` provides integer-only arithmetic, there is no analogous mechanism to provide automatic rounding or truncation to a certain number of decimal places. For rounding to a certain number of digits, `sprintf()` or `printf()` is usually the easiest route. See `perlfaq4`.

Floating-point numbers are only approximations to what a mathematician would call real numbers. There are infinitely more reals than floats, so some corners must be cut. For example:

```
printf "%.20g\n", 123456789123456789;
#        produces 123456789123456784
```

Testing for exact floating-point equality or inequality is not a good idea. Here's a (relatively expensive) work-around to compare whether two floating-point numbers are equal to a particular number of decimal places. See Knuth, volume II, for a more robust treatment of this topic.

```
sub fp_equal {
    my ($X, $Y, $POINTS) = @_;
    my ($tX, $tY);
    $tX = sprintf("%.${POINTS}g", $X);
    $tY = sprintf("%.${POINTS}g", $Y);
    return $tX eq $tY;
}
```

The POSIX module (part of the standard perl distribution) implements `ceil()`, `floor()`, and other mathematical and trigonometric functions. The `Math-Complex` module (part of the standard perl distribution) defines mathematical functions that work on both the reals and the imaginary numbers. `Math::Complex` is not as efficient as POSIX, but POSIX can't work with complex numbers.

Rounding in financial applications can have serious implications, and the rounding method used should be specified precisely. In these cases, it probably pays not to trust whichever system rounding is being used by Perl, but to instead implement the rounding function you need yourself.

48.2.39 Bigger Numbers

The standard `Math-BigInt`, `Math-BigRat`, and `Math-BigFloat` modules, along with the `bignum`, `bigint`, and `bigrat` pragmas, provide variable-precision arithmetic and overloaded operators, although they're currently pretty slow. At the cost of some space and considerable speed, they avoid the normal pitfalls associated with limited-precision representations.

```
    use 5.010;
    use bigint;  # easy interface to Math::BigInt
    $x = 123456789123456789;
    say $x * $x;
+15241578780673678515622620750190521
```

Or with rationals:

```
use 5.010;
use bigrat;
$x = 3/22;
$y = 4/6;
say "x/y is ", $x/$y;
say "x*y is ", $x*$y;
x/y is 9/44
x*y is 1/11
```

Several modules let you calculate with unlimited or fixed precision (bound only by memory and CPU time). There are also some non-standard modules that provide faster implementations via external C libraries.

Here is a short, but incomplete summary:

`Math::String`	treat string sequences like numbers
`Math::FixedPrecision`	calculate with a fixed precision
`Math::Currency`	for currency calculations

Bit::Vector	manipulate bit vectors fast (uses C)
Math::BigIntFast	Bit::Vector wrapper for big numbers
Math::Pari	provides access to the Pari C library
Math::Cephes	uses the external Cephes C library (no big numbers)
Math::Cephes::Fraction	fractions via the Cephes library
Math::GMP	another one using an external C library
Math::GMPz	an alternative interface to libgmp's big ints
Math::GMPq	an interface to libgmp's fraction numbers
Math::GMPf	an interface to libgmp's floating point numbers

Choose wisely.

49 perlopentut

49.1 NAME

perlopentut - simple recipes for opening files and pipes in Perl

49.2 DESCRIPTION

Whenever you do I/O on a file in Perl, you do so through what in Perl is called a **filehandle**. A filehandle is an internal name for an external file. It is the job of the `open` function to make the association between the internal name and the external name, and it is the job of the `close` function to break that association.

For your convenience, Perl sets up a few special filehandles that are already open when you run. These include `STDIN`, `STDOUT`, `STDERR`, and `ARGV`. Since those are pre-opened, you can use them right away without having to go to the trouble of opening them yourself:

```
print STDERR "This is a debugging message.\n";

print STDOUT "Please enter something: ";
$response = <STDIN> // die "how come no input?";
print STDOUT "Thank you!\n";

while (<ARGV>) { ... }
```

As you see from those examples, `STDOUT` and `STDERR` are output handles, and `STDIN` and `ARGV` are input handles. They are in all capital letters because they are reserved to Perl, much like the `@ARGV` array and the `%ENV` hash are. Their external associations were set up by your shell.

You will need to open every other filehandle on your own. Although there are many variants, the most common way to call Perl's open() function is with three arguments and one return value:

`OK = open(HANDLE, MODE, PATHNAME)`

Where:

OK

will be some defined value if the open succeeds, but `undef` if it fails;

HANDLE

should be an undefined scalar variable to be filled in by the `open` function if it succeeds;

MODE

is the access mode and the encoding format to open the file with;

PATHNAME

is the external name of the file you want opened.

Most of the complexity of the `open` function lies in the many possible values that the *MODE* parameter can take on.

One last thing before we show you how to open files: opening files does not (usually) automatically lock them in Perl. See `perlfaq5` for how to lock.

49.3 Opening Text Files

49.3.1 Opening Text Files for Reading

If you want to read from a text file, first open it in read-only mode like this:

```
my $filename = "/some/path/to/a/textfile/goes/here";
my $encoding = ":encoding(UTF-8)";
my $handle   = undef;     # this will be filled in on success

open($handle, "< $encoding", $filename)
    || die "$0: can’t open $filename for reading: $!";
```

As with the shell, in Perl the "<" is used to open the file in read-only mode. If it succeeds, Perl allocates a brand new filehandle for you and fills in your previously undefined $handle argument with a reference to that handle.

Now you may use functions like readline, read, getc, and sysread on that handle. Probably the most common input function is the one that looks like an operator:

```
$line = readline($handle);
$line = <$handle>;          # same thing
```

Because the readline function returns undef at end of file or upon error, you will sometimes see it used this way:

```
$line = <$handle>;
if (defined $line) {
    # do something with $line
}
else {
    # $line is not valid, so skip it
}
```

You can also just quickly die on an undefined value this way:

```
$line = <$handle> // die "no input found";
```

However, if hitting EOF is an expected and normal event, you do not want to exit simply because you have run out of input. Instead, you probably just want to exit an input loop. You can then test to see if an actual error has caused the loop to terminate, and act accordingly:

```
while (<$handle>) {
    # do something with data in $_
}
if ($!) {
    die "unexpected error while reading from $filename: $!";
}
```

A Note on Encodings: Having to specify the text encoding every time might seem a bit of a bother. To set up a default encoding for open so that you don’t have to supply it each time, you can use the open pragma:

```
use open qw< :encoding(UTF-8) >;
```

Once you’ve done that, you can safely omit the encoding part of the open mode:

```
open($handle, "<", $filename)
    || die "$0: can’t open $filename for reading: $!";
```

But never use the bare "<" without having set up a default encoding first. Otherwise, Perl cannot know which of the many, many, many possible flavors of text file you have, and Perl will have no idea how to correctly map the data in your file into actual characters it can work with. Other common encoding formats including "ASCII", "ISO-8859-1", "ISO-8859-15", "Windows-1252", "MacRoman", and even "UTF-16LE". See Section 84.1 [perlunitut NAME], page 1367 for more about encodings.

49.3.2 Opening Text Files for Writing

When you want to write to a file, you first have to decide what to do about any existing contents of that file. You have two basic choices here: to preserve or to clobber.

If you want to preserve any existing contents, then you want to open the file in append mode. As in the shell, in Perl you use ">>" to open an existing file in append mode. ">>" creates the file if it does not already exist.

```
my $handle   = undef;
my $filename = "/some/path/to/a/textfile/goes/here";
my $encoding = ":encoding(UTF-8)";

open($handle, ">> $encoding", $filename)
    || die "$0: can’t open $filename for appending: $!";
```

Now you can write to that filehandle using any of `print`, `printf`, `say`, `write`, or `syswrite`.

As noted above, if the file does not already exist, then the append-mode open will create it for you. But if the file does already exist, its contents are safe from harm because you will be adding your new text past the end of the old text.

On the other hand, sometimes you want to clobber whatever might already be there. To empty out a file before you start writing to it, you can open it in write-only mode:

```
my $handle   = undef;
my $filename = "/some/path/to/a/textfile/goes/here";
my $encoding = ":encoding(UTF-8)";

open($handle, "> $encoding", $filename)
    || die "$0: can’t open $filename in write-open mode: $!";
```

Here again Perl works just like the shell in that the ">" clobbers an existing file.

As with the append mode, when you open a file in write-only mode, you can now write to that filehandle using any of `print`, `printf`, `say`, `write`, or `syswrite`.

What about read-write mode? You should probably pretend it doesn’t exist, because opening text files in read-write mode is unlikely to do what you would like. See `perlfaq5` for details.

49.4 Opening Binary Files

If the file to be opened contains binary data instead of text characters, then the MODE argument to open is a little different. Instead of specifying the encoding, you tell Perl that your data are in raw bytes.

```
my $filename = "/some/path/to/a/binary/file/goes/here";
my $encoding = ":raw :bytes"
my $handle   = undef;     # this will be filled in on success
```

And then open as before, choosing "<", ">>", or ">" as needed:

```
open($handle, "< $encoding", $filename)
    || die "$0: can't open $filename for reading: $!";

open($handle, ">> $encoding", $filename)
    || die "$0: can't open $filename for appending: $!";

open($handle, "> $encoding", $filename)
    || die "$0: can't open $filename in write-open mode: $!";
```

Alternately, you can change to binary mode on an existing handle this way:

```
binmode($handle)    || die "cannot binmode handle";
```

This is especially handy for the handles that Perl has already opened for you.

```
binmode(STDIN)      || die "cannot binmode STDIN";
binmode(STDOUT)     || die "cannot binmode STDOUT";
```

You can also pass binmode an explicit encoding to change it on the fly. This isn't exactly "binary" mode, but we still use binmode to do it:

```
binmode(STDIN,  ":encoding(MacRoman)") || die "cannot binmode STDIN";
binmode(STDOUT, ":encoding(UTF-8)")    || die "cannot binmode STDOUT";
```

Once you have your binary file properly opened in the right mode, you can use all the same Perl I/O functions as you used on text files. However, you may wish to use the fixed-size read instead of the variable-sized readline for your input.

Here's an example of how to copy a binary file:

```
my $BUFSIZ   = 64 * (2 ** 10);
my $name_in  = "/some/input/file";
my $name_out = "/some/output/flie";

my($in_fh, $out_fh, $buffer);

open($in_fh,  "<", $name_in)
    || die "$0: cannot open $name_in for reading: $!";
open($out_fh, ">", $name_out)
    || die "$0: cannot open $name_out for writing: $!";

for my $fh ($in_fh, $out_fh)  {
    binmode($fh)               || die "binmode failed";
}
```

```
while (read($in_fh, $buffer, $BUFSIZ)) {
    unless (print $out_fh $buffer) {
        die "couldn't write to $name_out: $!";
    }
}

close($in_fh)       || die "couldn't close $name_in: $!";
close($out_fh)      || die "couldn't close $name_out: $!";
```

49.5 Opening Pipes

To be announced.

49.6 Low-level File Opens via sysopen

To be announced. Or deleted.

49.7 SEE ALSO

To be announced.

49.8 AUTHOR and COPYRIGHT

50 perlpacktut

50.1 NAME

perlpacktut - tutorial on `pack` and `unpack`

50.2 DESCRIPTION

`pack` and `unpack` are two functions for transforming data according to a user-defined template, between the guarded way Perl stores values and some well-defined representation as might be required in the environment of a Perl program. Unfortunately, they're also two of the most misunderstood and most often overlooked functions that Perl provides. This tutorial will demystify them for you.

50.3 The Basic Principle

Most programming languages don't shelter the memory where variables are stored. In C, for instance, you can take the address of some variable, and the `sizeof` operator tells you how many bytes are allocated to the variable. Using the address and the size, you may access the storage to your heart's content.

In Perl, you just can't access memory at random, but the structural and representational conversion provided by `pack` and `unpack` is an excellent alternative. The `pack` function converts values to a byte sequence containing representations according to a given specification, the so-called `"template"` argument. `unpack` is the reverse process, deriving some values from the contents of a string of bytes. (Be cautioned, however, that not all that has been packed together can be neatly unpacked - a very common experience as seasoned travellers are likely to confirm.)

Why, you may ask, would you need a chunk of memory containing some values in binary representation? One good reason is input and output accessing some file, a device, or a network connection, whereby this binary representation is either forced on you or will give you some benefit in processing. Another cause is passing data to some system call that is not available as a Perl function: `syscall` requires you to provide parameters stored in the way it happens in a C program. Even text processing (as shown in the next section) may be simplified with judicious usage of these two functions.

To see how (un)packing works, we'll start with a simple template code where the conversion is in low gear: between the contents of a byte sequence and a string of hexadecimal digits. Let's use `unpack`, since this is likely to remind you of a dump program, or some desperate last message unfortunate programs are wont to throw at you before they expire into the wild blue yonder. Assuming that the variable `$mem` holds a sequence of bytes that we'd like to inspect without assuming anything about its meaning, we can write

```
my( $hex ) = unpack( 'H*', $mem );
print "$hex\n";
```

whereupon we might see something like this, with each pair of hex digits corresponding to a byte:

```
41204d414e204120504c414e20412043414e414c2050414e414d41
```

What was in this chunk of memory? Numbers, characters, or a mixture of both? Assuming that we're on a computer where ASCII (or some similar) encoding is used: hexadecimal values in the range `0x40` - `0x5A` indicate an uppercase letter, and `0x20` encodes a space. So we might assume it is a piece of text, which some are able to read like a tabloid; but others will have to get hold of an ASCII table and relive that firstgrader feeling. Not caring too much about which way to read this, we note that `unpack` with the template code `H` converts the contents of a sequence of bytes into the customary hexadecimal notation. Since "a sequence of" is a pretty vague indication of quantity, `H` has been defined to convert just a single hexadecimal digit unless it is followed by a repeat count. An asterisk for the repeat count means to use whatever remains.

The inverse operation - packing byte contents from a string of hexadecimal digits - is just as easily written. For instance:

```
my $s = pack( 'H2' x 10, 30..39 );
print "$s\n";
```

Since we feed a list of ten 2-digit hexadecimal strings to `pack`, the pack template should contain ten pack codes. If this is run on a computer with ASCII character coding, it will print `0123456789`.

50.4 Packing Text

Let's suppose you've got to read in a data file like this:

```
Date      |Description                | Income|Expenditure
01/24/2001 Zed's Camel Emporium                    1147.99
01/28/2001 Flea spray                                24.99
01/29/2001 Camel rides to tourists      235.00
```

How do we do it? You might think first to use `split`; however, since `split` collapses blank fields, you'll never know whether a record was income or expenditure. Oops. Well, you could always use `substr`:

```
while (<>) {
    my $date   = substr($_,  0, 11);
    my $desc   = substr($_, 12, 27);
    my $income = substr($_, 40,  7);
    my $expend = substr($_, 52,  7);
    ...
}
```

It's not really a barrel of laughs, is it? In fact, it's worse than it may seem; the eagle-eyed may notice that the first field should only be 10 characters wide, and the error has propagated right through the other numbers - which we've had to count by hand. So it's error-prone as well as horribly unfriendly.

Or maybe we could use regular expressions:

```
while (<>) {
    my($date, $desc, $income, $expend) =
        m|(\d\d/\d\d/\d{4}) (.{27}) (.{7})(.*)|;
    ...
}
```

Urgh. Well, it's a bit better, but - well, would you want to maintain that?

Hey, isn't Perl supposed to make this sort of thing easy? Well, it does, if you use the right tools. pack and unpack are designed to help you out when dealing with fixed-width data like the above. Let's have a look at a solution with unpack:

```
while (<>) {
    my($date, $desc, $income, $expend) = unpack("A10xA27xA7A*", $_);
    ...
}
```

That looks a bit nicer; but we've got to take apart that weird template. Where did I pull that out of?

OK, let's have a look at some of our data again; in fact, we'll include the headers, and a handy ruler so we can keep track of where we are.

```
         1         2         3         4         5
1234567890123456789012345678901234567890123456789012345678
Date      |Description                | Income|Expenditure
01/28/2001 Flea spray                                24.99
01/29/2001 Camel rides to tourists      235.00
```

From this, we can see that the date column stretches from column 1 to column 10 - ten characters wide. The pack-ese for "character" is A, and ten of them are A10. So if we just wanted to extract the dates, we could say this:

```
my($date) = unpack("A10", $_);
```

OK, what's next? Between the date and the description is a blank column; we want to skip over that. The x template means "skip forward", so we want one of those. Next, we have another batch of characters, from 12 to 38. That's 27 more characters, hence A27. (Don't make the fencepost error - there are 27 characters between 12 and 38, not 26. Count 'em!)

Now we skip another character and pick up the next 7 characters:

```
my($date,$description,$income) = unpack("A10xA27xA7", $_);
```

Now comes the clever bit. Lines in our ledger which are just income and not expenditure might end at column 46. Hence, we don't want to tell our unpack pattern that we **need** to find another 12 characters; we'll just say "if there's anything left, take it". As you might guess from regular expressions, that's what the * means: "use everything remaining".

- Be warned, though, that unlike regular expressions, if the unpack template doesn't match the incoming data, Perl will scream and die.

Hence, putting it all together:

```
my ($date, $description, $income, $expend) =
    unpack("A10xA27xA7xA*", $_);
```

Now, that's our data parsed. I suppose what we might want to do now is total up our income and expenditure, and add another line to the end of our ledger - in the same format - saying how much we've brought in and how much we've spent:

```
while (<>) {
    my ($date, $desc, $income, $expend) =
        unpack("A10xA27xA7xA*", $_);
    $tot_income += $income;
    $tot_expend += $expend;
}

$tot_income = sprintf("%.2f", $tot_income); # Get them into
$tot_expend = sprintf("%.2f", $tot_expend); # "financial" format

$date = POSIX::strftime("%m/%d/%Y", localtime);

# OK, let's go:

print pack("A10xA27xA7xA*", $date, "Totals",
    $tot_income, $tot_expend);
```

Oh, hmm. That didn't quite work. Let's see what happened:

```
01/24/2001 Zed's Camel Emporium                     1147.99
01/28/2001 Flea spray                                 24.99
01/29/2001 Camel rides to tourists     1235.00
03/23/2001Totals                     1235.001172.98
```

OK, it's a start, but what happened to the spaces? We put x, didn't we? Shouldn't it skip forward? Let's look at what ⟨undefined⟩ [perlfunc pack], page ⟨undefined⟩ says:

```
x   A null byte.
```

Urgh. No wonder. There's a big difference between "a null byte", character zero, and "a space", character 32. Perl's put something between the date and the description - but unfortunately, we can't see it!

What we actually need to do is expand the width of the fields. The A format pads any non-existent characters with spaces, so we can use the additional spaces to line up our fields, like this:

```
print pack("A11 A28 A8 A*", $date, "Totals",
    $tot_income, $tot_expend);
```

(Note that you can put spaces in the template to make it more readable, but they don't translate to spaces in the output.) Here's what we got this time:

```
01/24/2001 Zed's Camel Emporium                     1147.99
01/28/2001 Flea spray                                 24.99
01/29/2001 Camel rides to tourists     1235.00
03/23/2001 Totals                      1235.00 1172.98
```

That's a bit better, but we still have that last column which needs to be moved further over. There's an easy way to fix this up: unfortunately, we can't get pack to right-justify our fields, but we can get sprintf to do it:

```
$tot_income = sprintf("%.2f", $tot_income);
$tot_expend = sprintf("%12.2f", $tot_expend);
$date = POSIX::strftime("%m/%d/%Y", localtime);
print pack("A11 A28 A8 A*", $date, "Totals",
    $tot_income, $tot_expend);
```

This time we get the right answer:

```
01/28/2001 Flea spray                                   24.99
01/29/2001 Camel rides to tourists     1235.00
03/23/2001 Totals                      1235.00      1172.98
```

So that's how we consume and produce fixed-width data. Let's recap what we've seen of `pack` and `unpack` so far:

- Use `pack` to go from several pieces of data to one fixed-width version; use `unpack` to turn a fixed-width-format string into several pieces of data.
- The pack format `A` means "any character"; if you're `pack`ing and you've run out of things to pack, `pack` will fill the rest up with spaces.
- `x` means "skip a byte" when `unpack`ing; when `pack`ing, it means "introduce a null byte" - that's probably not what you mean if you're dealing with plain text.
- You can follow the formats with numbers to say how many characters should be affected by that format: `A12` means "take 12 characters"; `x6` means "skip 6 bytes" or "character 0, 6 times".
- Instead of a number, you can use `*` to mean "consume everything else left".

  Warning: when packing multiple pieces of data, `*` only means "consume all of the current piece of data". That's to say

  ```
  pack("A*A*", $one, $two)
  ```

  packs all of `$one` into the first `A*` and then all of `$two` into the second. This is a general principle: each format character corresponds to one piece of data to be `pack`ed.

50.5 Packing Numbers

So much for textual data. Let's get onto the meaty stuff that `pack` and `unpack` are best at: handling binary formats for numbers. There is, of course, not just one binary format - life would be too simple - but Perl will do all the finicky labor for you.

50.5.1 Integers

Packing and unpacking numbers implies conversion to and from some *specific* binary representation. Leaving floating point numbers aside for the moment, the salient properties of any such representation are:

- the number of bytes used for storing the integer,
- whether the contents are interpreted as a signed or unsigned number,
- the byte ordering: whether the first byte is the least or most significant byte (or: little-endian or big-endian, respectively).

So, for instance, to pack 20302 to a signed 16 bit integer in your computer's representation you write

```
my $ps = pack( 's', 20302 );
```

Again, the result is a string, now containing 2 bytes. If you print this string (which is, generally, not recommended) you might see `ON` or `NO` (depending on your system's byte ordering) - or something entirely different if your computer doesn't use ASCII character encoding. Unpacking `$ps` with the same template returns the original integer value:

```
   my( $s ) = unpack( 's', $ps );
```

This is true for all numeric template codes. But don't expect miracles: if the packed value exceeds the allotted byte capacity, high order bits are silently discarded, and unpack certainly won't be able to pull them back out of some magic hat. And, when you pack using a signed template code such as `s`, an excess value may result in the sign bit getting set, and unpacking this will smartly return a negative value.

16 bits won't get you too far with integers, but there is `l` and `L` for signed and unsigned 32-bit integers. And if this is not enough and your system supports 64 bit integers you can push the limits much closer to infinity with pack codes `q` and `Q`. A notable exception is provided by pack codes `i` and `I` for signed and unsigned integers of the "local custom" variety: Such an integer will take up as many bytes as a local C compiler returns for `sizeof(int)`, but it'll use *at least* 32 bits.

Each of the integer pack codes `sSlLqQ` results in a fixed number of bytes, no matter where you execute your program. This may be useful for some applications, but it does not provide for a portable way to pass data structures between Perl and C programs (bound to happen when you call XS extensions or the Perl function `syscall`), or when you read or write binary files. What you'll need in this case are template codes that depend on what your local C compiler compiles when you code `short` or `unsigned long`, for instance. These codes and their corresponding byte lengths are shown in the table below. Since the C standard leaves much leeway with respect to the relative sizes of these data types, actual values may vary, and that's why the values are given as expressions in C and Perl. (If you'd like to use values from `%Config` in your program you have to import it with `use Config`.)

signed	unsigned	byte length in C	byte length in Perl
s!	S!	sizeof(short)	$Config{shortsize}
i!	I!	sizeof(int)	$Config{intsize}
l!	L!	sizeof(long)	$Config{longsize}
q!	Q!	sizeof(long long)	$Config{longlongsize}

The `i!` and `I!` codes aren't different from `i` and `I`; they are tolerated for completeness' sake.

50.5.2 Unpacking a Stack Frame

Requesting a particular byte ordering may be necessary when you work with binary data coming from some specific architecture whereas your program could run on a totally different system. As an example, assume you have 24 bytes containing a stack frame as it happens on an Intel 8086:

```
      +---------+        +----+----+               +---------+
 TOS: |   IP    |  TOS+4:| FL | FH | FLAGS  TOS+14:|   SI    |
      +---------+        +----+----+               +---------+
      |   CS    |        | AL | AH | AX            |   DI    |
      +---------+        +----+----+               +---------+
                         | BL | BH | BX            |   BP    |
                         +----+----+               +---------+
                         | CL | CH | CX            |   DS    |
                         +----+----+               +---------+
                         | DL | DH | DX            |   ES    |
                      +----+----+               +---------+
```

First, we note that this time-honored 16-bit CPU uses little-endian order, and that's why the low order byte is stored at the lower address. To unpack such a (unsigned) short we'll have to use code `v`. A repeat count unpacks all 12 shorts:

```
my( $ip, $cs, $flags, $ax, $bx, $cd, $dx, $si, $di, $bp, $ds, $es ) =
  unpack( 'v12', $frame );
```

Alternatively, we could have used `C` to unpack the individually accessible byte registers FL, FH, AL, AH, etc.:

```
my( $fl, $fh, $al, $ah, $bl, $bh, $cl, $ch, $dl, $dh ) =
  unpack( 'C10', substr( $frame, 4, 10 ) );
```

It would be nice if we could do this in one fell swoop: unpack a short, back up a little, and then unpack 2 bytes. Since Perl *is* nice, it proffers the template code `X` to back up one byte. Putting this all together, we may now write:

```
my( $ip, $cs,
    $flags,$fl,$fh,
    $ax,$al,$ah, $bx,$bl,$bh, $cx,$cl,$ch, $dx,$dl,$dh,
    $si, $di, $bp, $ds, $es ) =
unpack( 'v2' . ('vXXCC' x 5) . 'v5', $frame );
```

(The clumsy construction of the template can be avoided - just read on!)

We've taken some pains to construct the template so that it matches the contents of our frame buffer. Otherwise we'd either get undefined values, or `unpack` could not unpack all. If `pack` runs out of items, it will supply null strings (which are coerced into zeroes whenever the pack code says so).

50.5.3 How to Eat an Egg on a Net

The pack code for big-endian (high order byte at the lowest address) is `n` for 16 bit and `N` for 32 bit integers. You use these codes if you know that your data comes from a compliant architecture, but, surprisingly enough, you should also use these pack codes if you exchange binary data, across the network, with some system that you know next to nothing about. The simple reason is that this order has been chosen as the *network order*, and all standard-fearing programs ought to follow this convention. (This is, of course, a stern backing for one of the Lilliputian parties and may well influence the political development there.) So, if the protocol expects you to send a message by sending the length first, followed by just so many bytes, you could write:

```
my $buf = pack( 'N', length( $msg ) ) . $msg;
```

or even:

```
my $buf = pack( 'NA*', length( $msg ), $msg );
```

and pass `$buf` to your send routine. Some protocols demand that the count should include the length of the count itself: then just add 4 to the data length. (But make sure to read Section 50.8 [Lengths and Widths], page 868 before you really code this!)

50.5.4 Byte-order modifiers

In the previous sections we've learned how to use `n`, `N`, `v` and `V` to pack and unpack integers with big- or little-endian byte-order. While this is nice, it's still rather limited because it leaves out all kinds of signed integers as well as 64-bit integers. For example, if you wanted to unpack a sequence of signed big-endian 16-bit integers in a platform-independent way, you would have to write:

```
my @data = unpack 's*', pack 'S*', unpack 'n*', $buf;
```

This is ugly. As of Perl 5.9.2, there's a much nicer way to express your desire for a certain byte-order: the `>` and `<` modifiers. `>` is the big-endian modifier, while `<` is the little-endian modifier. Using them, we could rewrite the above code as:

```
my @data = unpack 's>*', $buf;
```

As you can see, the "big end" of the arrow touches the `s`, which is a nice way to remember that `>` is the big-endian modifier. The same obviously works for `<`, where the "little end" touches the code.

You will probably find these modifiers even more useful if you have to deal with big- or little-endian C structures. Be sure to read Section 50.9 [Packing and Unpacking C Structures], page 870 for more on that.

50.5.5 Floating point Numbers

For packing floating point numbers you have the choice between the pack codes `f`, `d`, `F` and `D`. `f` and `d` pack into (or unpack from) single-precision or double-precision representation as it is provided by your system. If your systems supports it, `D` can be used to pack and unpack (`long double`) values, which can offer even more resolution than `f` or `d`. **Note that there are different long double formats.**

`F` packs an `NV`, which is the floating point type used by Perl internally.

There is no such thing as a network representation for reals, so if you want to send your real numbers across computer boundaries, you'd better stick to text representation, possibly using the hexadecimal float format (avoiding the decimal conversion loss), unless you're absolutely sure what's on the other end of the line. For the even more adventuresome, you can use the byte-order modifiers from the previous section also on floating point codes.

50.6 Exotic Templates

50.6.1 Bit Strings

Bits are the atoms in the memory world. Access to individual bits may have to be used either as a last resort or because it is the most convenient way to handle your data. Bit string (un)packing converts between strings containing a series of `0` and `1` characters and a sequence of bytes each containing a group of 8 bits. This is almost as simple as it sounds, except that there are two ways the contents of a byte may be written as a bit string. Let's have a look at an annotated byte:

```
  7 6 5 4 3 2 1 0
+-----------------+
| 1 0 0 0 1 1 0 0 |
+-----------------+
 MSB           LSB
```

It's egg-eating all over again: Some think that as a bit string this should be written "10001100" i.e. beginning with the most significant bit, others insist on "00110001". Well, Perl isn't biased, so that's why we have two bit string codes:

```
$byte = pack( 'B8', '10001100' ); # start with MSB
$byte = pack( 'b8', '00110001' ); # start with LSB
```

It is not possible to pack or unpack bit fields - just integral bytes. `pack` always starts at the next byte boundary and "rounds up" to the next multiple of 8 by adding zero bits as required. (If you do want bit fields, there is ⟨undefined⟩ [perlfunc vec], page ⟨undefined⟩. Or you could implement bit field handling at the character string level, using split, substr, and concatenation on unpacked bit strings.)

To illustrate unpacking for bit strings, we'll decompose a simple status register (a "-" stands for a "reserved" bit):

```
+-----------------+-----------------+
| S Z - A - P - C | - - - - O D I T |
+-----------------+-----------------+
 MSB           LSB MSB           LSB
```

Converting these two bytes to a string can be done with the unpack template `'b16'`. To obtain the individual bit values from the bit string we use `split` with the "empty" separator pattern which dissects into individual characters. Bit values from the "reserved" positions are simply assigned to `undef`, a convenient notation for "I don't care where this goes".

```
($carry, undef, $parity, undef, $auxcarry, undef, $zero, $sign,
 $trace, $interrupt, $direction, $overflow) =
   split( //, unpack( 'b16', $status ) );
```

We could have used an unpack template `'b12'` just as well, since the last 4 bits can be ignored anyway.

50.6.2 Uuencoding

Another odd-man-out in the template alphabet is `u`, which packs a "uuencoded string". ("uu" is short for Unix-to-Unix.) Chances are that you won't ever need this encoding technique which was invented to overcome the shortcomings of old-fashioned transmission mediums that do not support other than simple ASCII data. The essential recipe is simple: Take three bytes, or 24 bits. Split them into 4 six-packs, adding a space (0x20) to each. Repeat until all of the data is blended. Fold groups of 4 bytes into lines no longer than 60 and garnish them in front with the original byte count (incremented by 0x20) and a `"\n"` at the end. - The `pack` chef will prepare this for you, a la minute, when you select pack code `u` on the menu:

```
my $uubuf = pack( 'u', $bindat );
```

A repeat count after `u` sets the number of bytes to put into an uuencoded line, which is the maximum of 45 by default, but could be set to some (smaller) integer multiple of three. `unpack` simply ignores the repeat count.

50.6.3 Doing Sums

An even stranger template code is `%<`*number*`>`. First, because it's used as a prefix to some other template code. Second, because it cannot be used in `pack` at all, and third, in `unpack`, doesn't return the data as defined by the template code it precedes. Instead it'll give you an integer of *number* bits that is computed from the data value by doing sums. For numeric unpack codes, no big feat is achieved:

```
    my $buf = pack( 'iii', 100, 20, 3 );
    print unpack( '%32i3', $buf ), "\n";  # prints 123
```

For string values, `%` returns the sum of the byte values saving you the trouble of a sum loop with `substr` and `ord`:

```
    print unpack( '%32A*', "\x01\x10" ), "\n";  # prints 17
```

Although the `%` code is documented as returning a "checksum": don't put your trust in such values! Even when applied to a small number of bytes, they won't guarantee a noticeable Hamming distance.

In connection with `b` or `B`, `%` simply adds bits, and this can be put to good use to count set bits efficiently:

```
    my $bitcount = unpack( '%32b*', $mask );
```

And an even parity bit can be determined like this:

```
    my $evenparity = unpack( '%1b*', $mask );
```

50.6.4 Unicode

Unicode is a character set that can represent most characters in most of the world's languages, providing room for over one million different characters. Unicode 3.1 specifies 94,140 characters: The Basic Latin characters are assigned to the numbers 0 - 127. The Latin-1 Supplement with characters that are used in several European languages is in the next range, up to 255. After some more Latin extensions we find the character sets from languages using non-Roman alphabets, interspersed with a variety of symbol sets such as currency symbols, Zapf Dingbats or Braille. (You might want to visit `http://www.unicode.org/` for a look at some of them - my personal favourites are Telugu and Kannada.)

The Unicode character sets associates characters with integers. Encoding these numbers in an equal number of bytes would more than double the requirements for storing texts written in Latin alphabets. The UTF-8 encoding avoids this by storing the most common (from a western point of view) characters in a single byte while encoding the rarer ones in three or more bytes.

Perl uses UTF-8, internally, for most Unicode strings.

So what has this got to do with `pack`? Well, if you want to compose a Unicode string (that is internally encoded as UTF-8), you can do so by using template code `U`. As an example, let's produce the Euro currency symbol (code number 0x20AC):

```
   $UTF8{Euro} = pack( 'U', 0x20AC );
   # Equivalent to: $UTF8{Euro} = "\x{20ac}";
```

Inspecting `$UTF8{Euro}` shows that it contains 3 bytes: "\xe2\x82\xac". However, it contains only 1 character, number 0x20AC. The round trip can be completed with `unpack`:

```
   $Unicode{Euro} = unpack( 'U', $UTF8{Euro} );
```

Unpacking using the `U` template code also works on UTF-8 encoded byte strings.

Usually you'll want to pack or unpack UTF-8 strings:

```
   # pack and unpack the Hebrew alphabet
   my $alefbet = pack( 'U*', 0x05d0..0x05ea );
   my @hebrew = unpack( 'U*', $utf );
```

Please note: in the general case, you're better off using Encode::decode_utf8 to decode a UTF-8 encoded byte string to a Perl Unicode string, and Encode::encode_utf8 to encode a Perl Unicode string to UTF-8 bytes. These functions provide means of handling invalid byte sequences and generally have a friendlier interface.

50.6.5 Another Portable Binary Encoding

The pack code `w` has been added to support a portable binary data encoding scheme that goes way beyond simple integers. (Details can be found at `http://Casbah.org/`, the Scarab project.) A BER (Binary Encoded Representation) compressed unsigned integer stores base 128 digits, most significant digit first, with as few digits as possible. Bit eight (the high bit) is set on each byte except the last. There is no size limit to BER encoding, but Perl won't go to extremes.

```
my $berbuf = pack( 'w*', 1, 128, 128+1, 128*128+127 );
```

A hex dump of `$berbuf`, with spaces inserted at the right places, shows 01 8100 8101 81807F. Since the last byte is always less than 128, `unpack` knows where to stop.

50.7 Template Grouping

Prior to Perl 5.8, repetitions of templates had to be made by `x`-multiplication of template strings. Now there is a better way as we may use the pack codes `(` and `)` combined with a repeat count. The `unpack` template from the Stack Frame example can simply be written like this:

```
unpack( 'v2 (vXXCC)5 v5', $frame )
```

Let's explore this feature a little more. We'll begin with the equivalent of

```
join( '', map( substr( $_, 0, 1 ), @str ) )
```

which returns a string consisting of the first character from each string. Using pack, we can write

```
pack( '(A)'.@str, @str )
```

or, because a repeat count `*` means "repeat as often as required", simply

```
pack( '(A)*', @str )
```

(Note that the template `A*` would only have packed `$str[0]` in full length.)

To pack dates stored as triplets (day, month, year) in an array `@dates` into a sequence of byte, byte, short integer we can write

```
$pd = pack( '(CCS)*', map( @$_, @dates ) );
```

To swap pairs of characters in a string (with even length) one could use several techniques. First, let's use `x` and `X` to skip forward and back:

```
$s = pack( '(A)*', unpack( '(xAXXAx)*', $s ) );
```

We can also use `@` to jump to an offset, with 0 being the position where we were when the last `(` was encountered:

```
$s = pack( '(A)*', unpack( '(@1A @0A @2)*', $s ) );
```

Finally, there is also an entirely different approach by unpacking big endian shorts and packing them in the reverse byte order:

```
$s = pack( '(v)*', unpack( '(n)*', $s );
```

50.8 Lengths and Widths

50.8.1 String Lengths

In the previous section we've seen a network message that was constructed by prefixing the binary message length to the actual message. You'll find that packing a length followed by so many bytes of data is a frequently used recipe since appending a null byte won't work if a null byte may be part of the data. Here is an example where both techniques are used: after two null terminated strings with source and destination address, a Short Message (to a mobile phone) is sent after a length byte:

```
my $msg = pack( 'Z*Z*CA*', $src, $dst, length( $sm ), $sm );
```

Unpacking this message can be done with the same template:

```
( $src, $dst, $len, $sm ) = unpack( 'Z*Z*CA*', $msg );
```

There's a subtle trap lurking in the offing: Adding another field after the Short Message (in variable `$sm`) is all right when packing, but this cannot be unpacked naively:

```
# pack a message
my $msg = pack( 'Z*Z*CA*C', $src, $dst, length( $sm ), $sm, $prio );

# unpack fails - $prio remains undefined!
( $src, $dst, $len, $sm, $prio ) = unpack( 'Z*Z*CA*C', $msg );
```

The pack code `A*` gobbles up all remaining bytes, and `$prio` remains undefined! Before we let disappointment dampen the morale: Perl's got the trump card to make this trick too, just a little further up the sleeve. Watch this:

```
# pack a message: ASCIIZ, ASCIIZ, length/string, byte
my $msg = pack( 'Z* Z* C/A* C', $src, $dst, $sm, $prio );

# unpack
( $src, $dst, $sm, $prio ) = unpack( 'Z* Z* C/A* C', $msg );
```

Combining two pack codes with a slash (`/`) associates them with a single value from the argument list. In `pack`, the length of the argument is taken and packed according to the first code while the argument itself is added after being converted with the template code after the slash. This saves us the trouble of inserting the `length` call, but it is in `unpack` where we really score: The value of the length byte marks the end of the string to be taken from the buffer. Since this combination doesn't make sense except when the second pack code isn't `a*`, `A*` or `Z*`, Perl won't let you.

The pack code preceding `/` may be anything that's fit to represent a number: All the numeric binary pack codes, and even text codes such as `A4` or `Z*`:

```
# pack/unpack a string preceded by its length in ASCII
my $buf = pack( 'A4/A*', "Humpty-Dumpty" );
# unpack $buf: '13  Humpty-Dumpty'
my $txt = unpack( 'A4/A*', $buf );
```

`/` is not implemented in Perls before 5.6, so if your code is required to work on older Perls you'll need to `unpack( 'Z* Z* C')` to get the length, then use it to make a new unpack string. For example

```
# pack a message: ASCIIZ, ASCIIZ, length, string, byte
# (5.005 compatible)
my $msg = pack( 'Z* Z* C A* C', $src, $dst, length $sm, $sm, $prio );

# unpack
( undef, undef, $len) = unpack( 'Z* Z* C', $msg );
($src, $dst, $sm, $prio) = unpack ( "Z* Z* x A$len C", $msg );
```

But that second unpack is rushing ahead. It isn't using a simple literal string for the template. So maybe we should introduce...

50.8.2 Dynamic Templates

So far, we've seen literals used as templates. If the list of pack items doesn't have fixed length, an expression constructing the template is required (whenever, for some reason, ()* cannot be used). Here's an example: To store named string values in a way that can be conveniently parsed by a C program, we create a sequence of names and null terminated ASCII strings, with = between the name and the value, followed by an additional delimiting null byte. Here's how:

```
my $env = pack( '(A*A*Z*)' . keys( %Env ) . 'C',
                map( { ( $_, '=', $Env{$_} ) } keys( %Env ) ), 0 );
```

Let's examine the cogs of this byte mill, one by one. There's the map call, creating the items we intend to stuff into the $env buffer: to each key (in $_) it adds the = separator and the hash entry value. Each triplet is packed with the template code sequence A*A*Z* that is repeated according to the number of keys. (Yes, that's what the keys function returns in scalar context.) To get the very last null byte, we add a 0 at the end of the pack list, to be packed with C. (Attentive readers may have noticed that we could have omitted the 0.)

For the reverse operation, we'll have to determine the number of items in the buffer before we can let unpack rip it apart:

```
my $n = $env =~ tr/\0// - 1;
my %env = map( split( /=/, $_ ), unpack( "(Z*)$n", $env ) );
```

The tr counts the null bytes. The unpack call returns a list of name-value pairs each of which is taken apart in the map block.

50.8.3 Counting Repetitions

Rather than storing a sentinel at the end of a data item (or a list of items), we could precede the data with a count. Again, we pack keys and values of a hash, preceding each with an unsigned short length count, and up front we store the number of pairs:

```
my $env = pack( 'S(S/A* S/A*)*', scalar keys( %Env ), %Env );
```

This simplifies the reverse operation as the number of repetitions can be unpacked with the / code:

```
my %env = unpack( 'S/(S/A* S/A*)', $env );
```

Note that this is one of the rare cases where you cannot use the same template for pack and unpack because pack can't determine a repeat count for a ()-group.

50.8.4 Intel HEX

Intel HEX is a file format for representing binary data, mostly for programming various chips, as a text file. (See `http://en.wikipedia.org/wiki/.hex` for a detailed description, and `http://en.wikipedia.org/wiki/SREC_(file_format)` for the Motorola S-record format, which can be unravelled using the same technique.) Each line begins with a colon (':') and is followed by a sequence of hexadecimal characters, specifying a byte count n (8 bit), an address (16 bit, big endian), a record type (8 bit), n data bytes and a checksum (8 bit) computed as the least significant byte of the two's complement sum of the preceding bytes. Example: `:0300300002337A1E`.

The first step of processing such a line is the conversion, to binary, of the hexadecimal data, to obtain the four fields, while checking the checksum. No surprise here: we'll start with a simple `pack` call to convert everything to binary:

```
my $binrec = pack( 'H*', substr( $hexrec, 1 ) );
```

The resulting byte sequence is most convenient for checking the checksum. Don't slow your program down with a for loop adding the `ord` values of this string's bytes - the `unpack` code `%` is the thing to use for computing the 8-bit sum of all bytes, which must be equal to zero:

```
die unless unpack( "%8C*", $binrec ) == 0;
```

Finally, let's get those four fields. By now, you shouldn't have any problems with the first three fields - but how can we use the byte count of the data in the first field as a length for the data field? Here the codes `x` and `X` come to the rescue, as they permit jumping back and forth in the string to unpack.

```
my( $addr, $type, $data ) = unpack( "x n C X4 C x3 /a", $bin );
```

Code `x` skips a byte, since we don't need the count yet. Code `n` takes care of the 16-bit big-endian integer address, and `C` unpacks the record type. Being at offset 4, where the data begins, we need the count. `X4` brings us back to square one, which is the byte at offset 0. Now we pick up the count, and zoom forth to offset 4, where we are now fully furnished to extract the exact number of data bytes, leaving the trailing checksum byte alone.

50.9 Packing and Unpacking C Structures

In previous sections we have seen how to pack numbers and character strings. If it were not for a couple of snags we could conclude this section right away with the terse remark that C structures don't contain anything else, and therefore you already know all there is to it. Sorry, no: read on, please.

If you have to deal with a lot of C structures, and don't want to hack all your template strings manually, you'll probably want to have a look at the CPAN module `Convert::Binary::C`. Not only can it parse your C source directly, but it also has built-in support for all the odds and ends described further on in this section.

50.9.1 The Alignment Pit

In the consideration of speed against memory requirements the balance has been tilted in favor of faster execution. This has influenced the way C compilers allocate memory for structures: On architectures where a 16-bit or 32-bit operand can be moved faster between places in memory, or to or from a CPU register, if it is aligned at an even or multiple-of-four or even at a multiple-of eight address, a C compiler will give you this speed benefit by stuffing extra bytes into structures. If you don't cross the C shoreline this is not likely to cause you any grief (although you should care when you design large data structures, or you want your code to be portable between architectures (you do want that, don't you?)).

To see how this affects pack and unpack, we'll compare these two C structures:

```
typedef struct {
  char     c1;
  short    s;
  char     c2;
  long     l;
} gappy_t;

typedef struct {
  long     l;
  short    s;
  char     c1;
  char     c2;
} dense_t;
```

Typically, a C compiler allocates 12 bytes to a gappy_t variable, but requires only 8 bytes for a dense_t. After investigating this further, we can draw memory maps, showing where the extra 4 bytes are hidden:

```
0           +4          +8          +12
+--+--+--+--+--+--+--+--+--+--+--+--+
|c1|xx|  s  |c2|xx|xx|xx|     l     |    xx = fill byte
+--+--+--+--+--+--+--+--+--+--+--+--+
gappy_t

0           +4          +8
+--+--+--+--+--+--+--+--+
|     l     |  h  |c1|c2|
+--+--+--+--+--+--+--+--+
dense_t
```

And that's where the first quirk strikes: pack and unpack templates have to be stuffed with x codes to get those extra fill bytes.

The natural question: "Why can't Perl compensate for the gaps?" warrants an answer. One good reason is that C compilers might provide (non-ANSI) extensions permitting all sorts of fancy control over the way structures are aligned, even at the level of an individual structure field. And, if this were not enough, there is an insidious thing called union where the amount of fill bytes cannot be derived from the alignment of the next item alone.

OK, so let's bite the bullet. Here's one way to get the alignment right by inserting template codes x, which don't take a corresponding item from the list:

```
my $gappy = pack( 'cxs cxxx l!', $c1, $s, $c2, $l );
```

Note the ! after l: We want to make sure that we pack a long integer as it is compiled by our C compiler. And even now, it will only work for the platforms where the compiler aligns things as above. And somebody somewhere has a platform where it doesn't. [Probably a Cray, where `short`s, `int`s and `long`s are all 8 bytes. :-)]

Counting bytes and watching alignments in lengthy structures is bound to be a drag. Isn't there a way we can create the template with a simple program? Here's a C program that does the trick:

```
#include <stdio.h>
#include <stddef.h>

typedef struct {
  char     fc1;
  short    fs;
  char     fc2;
  long     fl;
} gappy_t;

#define Pt(struct,field,tchar) \
  printf( "@%d%s ", offsetof(struct,field), # tchar );

int main() {
  Pt( gappy_t, fc1, c  );
  Pt( gappy_t, fs,  s! );
  Pt( gappy_t, fc2, c  );
  Pt( gappy_t, fl,  l! );
  printf( "\n" );
}
```

The output line can be used as a template in a `pack` or `unpack` call:

```
my $gappy = pack( '@0c @2s! @4c @8l!', $c1, $s, $c2, $l );
```

Gee, yet another template code - as if we hadn't plenty. But `@` saves our day by enabling us to specify the offset from the beginning of the pack buffer to the next item: This is just the value the `offsetof` macro (defined in `<stddef.h>`) returns when given a `struct` type and one of its field names ("member-designator" in C standardese).

Neither using offsets nor adding `x`'s to bridge the gaps is satisfactory. (Just imagine what happens if the structure changes.) What we really need is a way of saying "skip as many bytes as required to the next multiple of N". In fluent Templatese, you say this with `x!N` where N is replaced by the appropriate value. Here's the next version of our struct packaging:

```
my $gappy = pack( 'c x!2 s c x!4 l!', $c1, $s, $c2, $l );
```

That's certainly better, but we still have to know how long all the integers are, and portability is far away. Rather than `2`, for instance, we want to say "however long a short is". But this can be done by enclosing the appropriate pack code in brackets: `[s]`. So, here's the very best we can do:

```
my $gappy = pack( 'c x![s] s c x![l!] l!', $c1, $s, $c2, $l );
```

50.9.2 Dealing with Endian-ness

Now, imagine that we want to pack the data for a machine with a different byte-order. First, we'll have to figure out how big the data types on the target machine really are. Let's assume that the longs are 32 bits wide and the shorts are 16 bits wide. You can then rewrite the template as:

```
my $gappy = pack( 'c x![s] s c x![l] l', $c1, $s, $c2, $l );
```

If the target machine is little-endian, we could write:

```
my $gappy = pack( 'c x![s] s< c x![l] l<', $c1, $s, $c2, $l );
```

This forces the short and the long members to be little-endian, and is just fine if you don't have too many struct members. But we could also use the byte-order modifier on a group and write the following:

```
my $gappy = pack( '( c x![s] s c x![l] l )<', $c1, $s, $c2, $l );
```

This is not as short as before, but it makes it more obvious that we intend to have little-endian byte-order for a whole group, not only for individual template codes. It can also be more readable and easier to maintain.

50.9.3 Alignment, Take 2

I'm afraid that we're not quite through with the alignment catch yet. The hydra raises another ugly head when you pack arrays of structures:

```
typedef struct {
  short    count;
  char     glyph;
} cell_t;

typedef cell_t buffer_t[BUFLEN];
```

Where's the catch? Padding is neither required before the first field `count`, nor between this and the next field `glyph`, so why can't we simply pack like this:

```
# something goes wrong here:
pack( 's!a' x @buffer,
      map{ ( $_->{count}, $_->{glyph} ) } @buffer );
```

This packs `3*@buffer` bytes, but it turns out that the size of `buffer_t` is four times `BUFLEN`! The moral of the story is that the required alignment of a structure or array is propagated to the next higher level where we have to consider padding *at the end* of each component as well. Thus the correct template is:

```
pack( 's!ax' x @buffer,
      map{ ( $_->{count}, $_->{glyph} ) } @buffer );
```

50.9.4 Alignment, Take 3

And even if you take all the above into account, ANSI still lets this:

```
typedef struct {
  char     foo[2];
} foo_t;
```

vary in size. The alignment constraint of the structure can be greater than any of its elements. [And if you think that this doesn't affect anything common, dismember the next cellphone that you see. Many have ARM cores, and the ARM structure rules make `sizeof (foo_t)` == 4]

50.9.5 Pointers for How to Use Them

The title of this section indicates the second problem you may run into sooner or later when you pack C structures. If the function you intend to call expects a, say, `void *` value, you *cannot* simply take a reference to a Perl variable. (Although that value certainly is a memory address, it's not the address where the variable's contents are stored.)

Template code `P` promises to pack a "pointer to a fixed length string". Isn't this what we want? Let's try:

```
# allocate some storage and pack a pointer to it
my $memory = "\x00" x $size;
my $memptr = pack( 'P', $memory );
```

But wait: doesn't `pack` just return a sequence of bytes? How can we pass this string of bytes to some C code expecting a pointer which is, after all, nothing but a number? The answer is simple: We have to obtain the numeric address from the bytes returned by `pack`.

```
my $ptr = unpack( 'L!', $memptr );
```

Obviously this assumes that it is possible to typecast a pointer to an unsigned long and vice versa, which frequently works but should not be taken as a universal law. - Now that we have this pointer the next question is: How can we put it to good use? We need a call to some C function where a pointer is expected. The read(2) system call comes to mind:

```
ssize_t read(int fd, void *buf, size_t count);
```

After reading Section 25.1 [perlfunc NAME], page 351 explaining how to use `syscall` we can write this Perl function copying a file to standard output:

```
require 'syscall.ph'; # run h2ph to generate this file
sub cat($){
    my $path = shift();
    my $size = -s $path;
    my $memory = "\x00" x $size;  # allocate some memory
    my $ptr = unpack( 'L', pack( 'P', $memory ) );
    open( F, $path ) || die( "$path: cannot open ($!)\n" );
    my $fd = fileno(F);
    my $res = syscall( &SYS_read, fileno(F), $ptr, $size );
    print $memory;
    close( F );
}
```

This is neither a specimen of simplicity nor a paragon of portability but it illustrates the point: We are able to sneak behind the scenes and access Perl's otherwise well-guarded memory! (Important note: Perl's `syscall` does *not* require you to construct pointers in this roundabout way. You simply pass a string variable, and Perl forwards the address.)

How does `unpack` with `P` work? Imagine some pointer in the buffer about to be unpacked: If it isn't the null pointer (which will smartly produce the `undef` value) we have a start address - but then what? Perl has no way of knowing how long this "fixed length string" is, so it's up to you to specify the actual size as an explicit length after `P`.

```
my $mem = "abcdefghijklmn";
print unpack( 'P5', pack( 'P', $mem ) ); # prints "abcde"
```

As a consequence, pack ignores any number or * after P.

Now that we have seen P at work, we might as well give p a whirl. Why do we need a second template code for packing pointers at all? The answer lies behind the simple fact that an unpack with p promises a null-terminated string starting at the address taken from the buffer, and that implies a length for the data item to be returned:

```
my $buf = pack( 'p', "abc\x00efhijklmn" );
print unpack( 'p', $buf );    # prints "abc"
```

Albeit this is apt to be confusing: As a consequence of the length being implied by the string's length, a number after pack code p is a repeat count, not a length as after P.

Using pack(..., $x) with P or p to get the address where $x is actually stored must be used with circumspection. Perl's internal machinery considers the relation between a variable and that address as its very own private matter and doesn't really care that we have obtained a copy. Therefore:

- Do not use pack with p or P to obtain the address of variable that's bound to go out of scope (and thereby freeing its memory) before you are done with using the memory at that address.
- Be very careful with Perl operations that change the value of the variable. Appending something to the variable, for instance, might require reallocation of its storage, leaving you with a pointer into no-man's land.
- Don't think that you can get the address of a Perl variable when it is stored as an integer or double number! pack('P', $x) will force the variable's internal representation to string, just as if you had written something like $x .= ''.

It's safe, however, to P- or p-pack a string literal, because Perl simply allocates an anonymous variable.

50.10 Pack Recipes

Here are a collection of (possibly) useful canned recipes for pack and unpack:

```
# Convert IP address for socket functions
pack( "C4", split /\./, "123.4.5.6" );

# Count the bits in a chunk of memory (e.g. a select vector)
unpack( '%32b*', $mask );

# Determine the endianness of your system
$is_little_endian = unpack( 'c', pack( 's', 1 ) );
$is_big_endian = unpack( 'xc', pack( 's', 1 ) );

# Determine the number of bits in a native integer
$bits = unpack( '%32I!', ~0 );

# Prepare argument for the nanosleep system call
my $timespec = pack( 'L!L!', $secs, $nanosecs );
```

For a simple memory dump we unpack some bytes into just as many pairs of hex digits, and use `map` to handle the traditional spacing - 16 bytes to a line:

```
my $i;
print map( ++$i % 16 ? "$_ " : "$_\n",
           unpack( 'H2' x length( $mem ), $mem ) ),
      length( $mem ) % 16 ? "\n" : '';
```

50.11 Funnies Section

```
# Pulling digits out of nowhere...
print unpack( 'C', pack( 'x' ) ),
      unpack( '%B*', pack( 'A' ) ),
      unpack( 'H', pack( 'A' ) ),
      unpack( 'A', unpack( 'C', pack( 'A' ) ) ), "\n";

# One for the road ;-)
my $advice = pack( 'all u can in a van' );
```

50.12 Authors

Simon Cozens and Wolfgang Laun.

51 perlperf

51.1 NAME

perlperf - Perl Performance and Optimization Techniques

51.2 DESCRIPTION

This is an introduction to the use of performance and optimization techniques which can be used with particular reference to perl programs. While many perl developers have come from other languages, and can use their prior knowledge where appropriate, there are many other people who might benefit from a few perl specific pointers. If you want the condensed version, perhaps the best advice comes from the renowned Japanese Samurai, Miyamoto Musashi, who said:

```
"Do Not Engage in Useless Activity"
```

in 1645.

51.3 OVERVIEW

Perhaps the most common mistake programmers make is to attempt to optimize their code before a program actually does anything useful - this is a bad idea. There's no point in having an extremely fast program that doesn't work. The first job is to get a program to *correctly* do something **useful**, (not to mention ensuring the test suite is fully functional), and only then to consider optimizing it. Having decided to optimize existing working code, there are several simple but essential steps to consider which are intrinsic to any optimization process.

51.3.1 ONE STEP SIDEWAYS

Firstly, you need to establish a baseline time for the existing code, which timing needs to be reliable and repeatable. You'll probably want to use the `Benchmark` or `Devel::NYTProf` modules, or something similar, for this step, or perhaps the Unix system `time` utility, whichever is appropriate. See the base of this document for a longer list of benchmarking and profiling modules, and recommended further reading.

51.3.2 ONE STEP FORWARD

Next, having examined the program for *hot spots*, (places where the code seems to run slowly), change the code with the intention of making it run faster. Using version control software, like `subversion`, will ensure no changes are irreversible. It's too easy to fiddle here and fiddle there - don't change too much at any one time or you might not discover which piece of code **really** was the slow bit.

51.3.3 ANOTHER STEP SIDEWAYS

It's not enough to say: `"that will make it run faster"`, you have to check it. Rerun the code under control of the benchmarking or profiling modules, from the first step above, and check that the new code executed the **same task** in *less time*. Save your work and repeat...

51.4 GENERAL GUIDELINES

The critical thing when considering performance is to remember there is no such thing as a `Golden Bullet`, which is why there are no rules, only guidelines.

It is clear that inline code is going to be faster than subroutine or method calls, because there is less overhead, but this approach has the disadvantage of being less maintainable and comes at the cost of greater memory usage - there is no such thing as a free lunch. If you are searching for an element in a list, it can be more efficient to store the data in a hash structure, and then simply look to see whether the key is defined, rather than to loop through the entire array using grep() for instance. substr() may be (a lot) faster than grep() but not as flexible, so you have another trade-off to access. Your code may contain a line which takes 0.01 of a second to execute which if you call it 1,000 times, quite likely in a program parsing even medium sized files for instance, you already have a 10 second delay, in just one single code location, and if you call that line 100,000 times, your entire program will slow down to an unbearable crawl.

Using a subroutine as part of your sort is a powerful way to get exactly what you want, but will usually be slower than the built-in *alphabetic* `cmp` and *numeric* `<=>` sort operators. It is possible to make multiple passes over your data, building indices to make the upcoming sort more efficient, and to use what is known as the `OM` (Orcish Maneuver) to cache the sort keys in advance. The cache lookup, while a good idea, can itself be a source of slowdown by enforcing a double pass over the data - once to setup the cache, and once to sort the data. Using `pack()` to extract the required sort key into a consistent string can be an efficient way to build a single string to compare, instead of using multiple sort keys, which makes it possible to use the standard, written in `c` and fast, perl `sort()` function on the output, and is the basis of the `GRT` (Guttman Rossler Transform). Some string combinations can slow the `GRT` down, by just being too plain complex for its own good.

For applications using database backends, the standard `DBIx` namespace has tries to help with keeping things nippy, not least because it tries to *not* query the database until the latest possible moment, but always read the docs which come with your choice of libraries. Among the many issues facing developers dealing with databases should remain aware of is to always use `SQL` placeholders and to consider pre-fetching data sets when this might prove advantageous. Splitting up a large file by assigning multiple processes to parsing a single file, using say `POE`, `threads` or `fork` can also be a useful way of optimizing your usage of the available `CPU` resources, though this technique is fraught with concurrency issues and demands high attention to detail.

Every case has a specific application and one or more exceptions, and there is no replacement for running a few tests and finding out which method works best for your particular environment, this is why writing optimal code is not an exact science, and why we love using Perl so much - TMTOWTDI.

51.5 BENCHMARKS

Here are a few examples to demonstrate usage of Perl's benchmarking tools.

51.5.1 Assigning and Dereferencing Variables.

I'm sure most of us have seen code which looks like, (or worse than), this:

```
if ( $obj->{_ref}->{_myscore} >= $obj->{_ref}->{_yourscore} ) {
    ...
```

This sort of code can be a real eyesore to read, as well as being very sensitive to typos, and it's much clearer to dereference the variable explicitly. We're side-stepping the issue of working with object-oriented programming techniques to encapsulate variable access via methods, only accessible through an object. Here we're just discussing the technical implementation of choice, and whether this has an effect on performance. We can see whether this dereferencing operation, has any overhead by putting comparative code in a file and running a Benchmark test.

```
# dereference

#!/usr/bin/perl

use strict;
use warnings;

use Benchmark;

my $ref = {
        'ref'   => {
            _myscore    => '100 + 1',
            _yourscore  => '102 - 1',
        },
};

timethese(1000000, {
        'direct'       => sub {
            my $x = $ref->{ref}->{_myscore} . $ref->{ref}->{_yourscore} ;
        },
        'dereference'  => sub {
            my $ref  = $ref->{ref};
            my $myscore = $ref->{_myscore};
            my $yourscore = $ref->{_yourscore};
            my $x = $myscore . $yourscore;
        },
});
```

It's essential to run any timing measurements a sufficient number of times so the numbers settle on a numerical average, otherwise each run will naturally fluctuate due to variations in the environment, to reduce the effect of contention for CPU resources and network bandwidth for instance. Running the above code for one million iterations, we can take a look at the report output by the Benchmark module, to see which approach is the most effective.

```
$> perl dereference

Benchmark: timing 1000000 iterations of dereference, direct...
dereference:  2 wallclock secs ( 1.59 usr +  0.00 sys =  1.59 CPU) @ 628930.82/s (n=100
    direct:  1 wallclock secs ( 1.20 usr +  0.00 sys =  1.20 CPU) @ 833333.33/s (n=1000
```

The difference is clear to see and the dereferencing approach is slower. While it managed to execute an average of 628,930 times a second during our test, the direct approach managed to run an additional 204,403 times, unfortunately. Unfortunately, because there are many examples of code written using the multiple layer direct variable access, and it's usually horrible. It is, however, minusculy faster. The question remains whether the minute gain is actually worth the eyestrain, or the loss of maintainability.

51.5.2 Search and replace or tr

If we have a string which needs to be modified, while a regex will almost always be much more flexible, `tr`, an oft underused tool, can still be a useful. One scenario might be replace all vowels with another character. The regex solution might look like this:

```
$str =~ s/[aeiou]/x/g
```

The `tr` alternative might look like this:

```
$str =~ tr/aeiou/xxxxx/
```

We can put that into a test file which we can run to check which approach is the fastest, using a global `$STR` variable to assign to the `my $str` variable so as to avoid perl trying to optimize any of the work away by noticing it's assigned only the once.

# regex-transliterate

```
#!/usr/bin/perl

use strict;
use warnings;

use Benchmark;

my $STR = "$$-this and that";

timethese( 1000000, {
        'sr'  => sub { my $str = $STR; $str =~ s/[aeiou]/x/g; return $str; },
        'tr'  => sub { my $str = $STR; $str =~ tr/aeiou/xxxxx/; return $str; },
});
```

Running the code gives us our results:

```
$> perl regex-transliterate

Benchmark: timing 1000000 iterations of sr, tr...
        sr:  2 wallclock secs ( 1.19 usr +  0.00 sys =  1.19 CPU) @ 840336.13/s (n=10000
        tr:  0 wallclock secs ( 0.49 usr +  0.00 sys =  0.49 CPU) @ 2040816.33/s (n=1000
```

The `tr` version is a clear winner. One solution is flexible, the other is fast - and it's appropriately the programmer's choice which to use.

Check the `Benchmark` docs for further useful techniques.

51.6 PROFILING TOOLS

A slightly larger piece of code will provide something on which a profiler can produce more extensive reporting statistics. This example uses the simplistic `wordmatch` program which parses a given input file and spews out a short report on the contents.

# wordmatch

```
#!/usr/bin/perl

use strict;
use warnings;

=head1 NAME

filewords - word analysis of input file

=head1 SYNOPSIS

    filewords -f inputfilename [-d]

=head1 DESCRIPTION

This program parses the given filename, specified with C<-f>, and displays a
simple analysis of the words found therein.  Use the C<-d> switch to enable
debugging messages.

=cut

use FileHandle;
use Getopt::Long;

my $debug   =  0;
my $file    = '';

my $result = GetOptions (
    'debug'         => \$debug,
    'file=s'        => \$file,
);
die("invalid args") unless $result;

unless ( -f $file ) {
    die("Usage: $0 -f filename [-d]");
}
my $FH = FileHandle->new("< $file") or die("unable to open file($file): $!");

my $i_LINES = 0;
my $i_WORDS = 0;
my %count   = ();
my @lines = <$FH>;
foreach my $line ( @lines ) {
    $i_LINES++;
    $line =~ s/\n//;
    my @words = split(/ +/, $line);
    my $i_words = scalar(@words);
    $i_WORDS = $i_WORDS + $i_words;
    debug("line: $i_LINES supplying $i_words words: @words");
    my $i_word = 0;
    foreach my $word ( @words ) {
        $i_word++;
        $count{$i_LINES}{spec} += matches($i_word, $word, ’[^a-zA-Z0-9]’);
        $count{$i_LINES}{only} += matches($i_word, $word, ’^[^a-zA-Z0-9]+$’);
        $count{$i_LINES}{cons} += matches($i_word, $word, ’^[(?i:bcdfghjklmnpqrstvwxyz)]
        $count{$i_LINES}{vows} += matches($i_word, $word, ’^[(?i:aeiou)]+$’);
        $count{$i_LINES}{caps} += matches($i_word, $word, ’^[(A-Z)]+$’);
    }
}

print report( %count );

sub matches {
    my $i_wd  = shift;
    my $word  = shift;
    my $regex = shift;
    my $has = 0;

    if ( $word =~ /($regex)/ ) {
        $has++ if $1;
    }

    debug("word: $i_wd ".($has ? ’matches’ : ’does not match’)." chars: /$regex/");

    return $has;
}

sub report {
    my %report = @_;
    my %rep;

    foreach my $line ( keys %report ) {
        foreach my $key ( keys %{ $report{$line} } ) {
            $rep{$key} += $report{$line}{$key};
        }
    }
    my $report = qq|
$0 report for $file:
lines in file: $i_LINES
words in file: $i_WORDS
words with special (non-word) characters: $i_spec
words with only special (non-word) characters: $i_only
words with only consonants: $i_cons
words with only capital letters: $i_caps
words with only vowels: $i_vows
|;

    return $report;
}

sub debug {
    my $message = shift;

    if ( $debug ) {
        print STDERR "DBG: $message\n";
    }
}

exit 0;
```

51.6.1 Devel::DProf

This venerable module has been the de-facto standard for Perl code profiling for more than a decade, but has been replaced by a number of other modules which have brought us back to the 21st century. Although you're recommended to evaluate your tool from the several mentioned here and from the CPAN list at the base of this document, (and currently Devel-NYTProf seems to be the weapon of choice - see below), we'll take a quick look at the output from Devel-DProf first, to set a baseline for Perl profiling tools. Run the above program under the control of Devel::DProf by using the -d switch on the command-line.

```
$> perl -d:DProf wordmatch -f perl5db.pl

<...multiple lines snipped...>

wordmatch report for perl5db.pl:
lines in file: 9428
words in file: 50243
words with special (non-word) characters: 20480
words with only special (non-word) characters: 7790
words with only consonants: 4801
words with only capital letters: 1316
words with only vowels: 1701
```

Devel::DProf produces a special file, called tmon.out by default, and this file is read by the dprofpp program, which is already installed as part of the Devel::DProf distribution.

If you call dprofpp with no options, it will read the tmon.out file in the current directory and produce a human readable statistics report of the run of your program. Note that this may take a little time.

```
$> dprofpp

Total Elapsed Time = 2.951677 Seconds
  User+System Time = 2.871677 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 102.   2.945  3.003 251215   0.0000 0.0000  main::matches
 2.40   0.069  0.069 260643   0.0000 0.0000  main::debug
 1.74   0.050  0.050      1   0.0500 0.0500  main::report
 1.04   0.030  0.049      4   0.0075 0.0123  main::BEGIN
 0.35   0.010  0.010      3   0.0033 0.0033  Exporter::as_heavy
 0.35   0.010  0.010      7   0.0014 0.0014  IO::File::BEGIN
 0.00       - -0.000      1        -      -  Getopt::Long::FindOption
 0.00       - -0.000      1        -      -  Symbol::BEGIN
 0.00       - -0.000      1        -      -  Fcntl::BEGIN
 0.00       - -0.000      1        -      -  Fcntl::bootstrap
 0.00       - -0.000      1        -      -  warnings::BEGIN
 0.00       - -0.000      1        -      -  IO::bootstrap
 0.00       - -0.000      1        -      -  Getopt::Long::ConfigDefaults
 0.00       - -0.000      1        -      -  Getopt::Long::Configure
 0.00       - -0.000      1        -      -  Symbol::gensym
```

dprofpp will produce some quite detailed reporting on the activity of the wordmatch program. The wallclock, user and system, times are at the top of the analysis, and after this are the main columns defining which define the report. Check the dprofpp docs for details of the many options it supports.

See also Apache::DProf which hooks Devel::DProf into mod_perl.

51.6.2 Devel::Profiler

Let's take a look at the same program using a different profiler: Devel::Profiler, a drop-in Perl-only replacement for Devel::DProf. The usage is very slightly different in that instead of using the special -d: flag, you pull Devel::Profiler in directly as a module using -M.

```
$> perl -MDevel::Profiler wordmatch -f perl5db.pl

<...multiple lines snipped...>

wordmatch report for perl5db.pl:
lines in file: 9428
words in file: 50243
words with special (non-word) characters: 20480
words with only special (non-word) characters: 7790
words with only consonants: 4801
words with only capital letters: 1316
words with only vowels: 1701
```

Devel::Profiler generates a tmon.out file which is compatible with the dprofpp program, thus saving the construction of a dedicated statistics reader program. dprofpp usage is therefore identical to the above example.

```
$> dprofpp

Total Elapsed Time =   20.984 Seconds
  User+System Time =   19.981 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 49.0   9.792 14.509 251215   0.0000 0.0001  main::matches
 24.4   4.887  4.887 260643   0.0000 0.0000  main::debug
 0.25   0.049  0.049      1   0.0490 0.0490  main::report
 0.00   0.000  0.000      1   0.0000 0.0000  Getopt::Long::GetOptions
 0.00   0.000  0.000      2   0.0000 0.0000  Getopt::Long::ParseOptionSpec
 0.00   0.000  0.000      1   0.0000 0.0000  Getopt::Long::FindOption
 0.00   0.000  0.000      1   0.0000 0.0000  IO::File::new
 0.00   0.000  0.000      1   0.0000 0.0000  IO::Handle::new
 0.00   0.000  0.000      1   0.0000 0.0000  Symbol::gensym
 0.00   0.000  0.000      1   0.0000 0.0000  IO::File::open
```

Interestingly we get slightly different results, which is mostly because the algorithm which generates the report is different, even though the output file format was allegedly identical. The elapsed, user and system times are clearly showing the time it took for Devel::Profiler to execute its own run, but the column listings feel more accurate somehow than the ones we had earlier from Devel::DProf. The 102% figure has disappeared, for example. This is where we have to use the tools at our disposal, and recognise their pros and cons, before using them. Interestingly, the numbers of calls for each subroutine are identical in the two reports, it's the percentages which differ. As the author of Devel::Proviler writes:

```
...running HTML::Template's test suite under Devel::DProf shows output()
taking NO time but Devel::Profiler shows around 10% of the time is in output().
I don't know which to trust but my gut tells me something is wrong with
Devel::DProf.  HTML::Template::output() is a big routine that's called for
every test. Either way, something needs fixing.
```

YMMV.

See also Devel::Apache::Profiler which hooks Devel::Profiler into mod_perl.

51.6.3 Devel::SmallProf

The Devel::SmallProf profiler examines the runtime of your Perl program and produces a line-by-line listing to show how many times each line was called, and how long each line took to execute. It is called by supplying the familiar -d flag to Perl at runtime.

```
$> perl -d:SmallProf wordmatch -f perl5db.pl

<...multiple lines snipped...>
```

```
wordmatch report for perl5db.pl:
lines in file: 9428
words in file: 50243
words with special (non-word) characters: 20480
words with only special (non-word) characters: 7790
words with only consonants: 4801
words with only capital letters: 1316
words with only vowels: 1701
```

Devel::SmallProf writes it's output into a file called smallprof.out, by default. The format of the file looks like this:

```
<num> <time> <ctime> <line>:<text>
```

When the program has terminated, the output may be examined and sorted using any standard text filtering utilities. Something like the following may be sufficient:

```
$> cat smallprof.out | grep \d*: | sort -k3 | tac | head -n20

251215   1.65674   7.68000    75: if ( $word =~ /($regex)/ ) {
251215   0.03264   4.40000    79: debug("word: $i_wd ".($has ? 'matches' :
251215   0.02693   4.10000    81: return $has;
260643   0.02841   4.07000   128: if ( $debug ) {
260643   0.02601   4.04000   126: my $message = shift;
251215   0.02641   3.91000    73: my $has = 0;
251215   0.03311   3.71000    70: my $i_wd  = shift;
251215   0.02699   3.69000    72: my $regex = shift;
251215   0.02766   3.68000    71: my $word  = shift;
 50243   0.59726   1.00000    59:  $count{$i_LINES}{cons} =
 50243   0.48175   0.92000    61:  $count{$i_LINES}{spec} =
 50243   0.00644   0.89000    56:  my $i_cons = matches($i_word, $word,
 50243   0.48837   0.88000    63:  $count{$i_LINES}{caps} =
 50243   0.00516   0.88000    58:  my $i_caps = matches($i_word, $word, '^[(A-
 50243   0.00631   0.81000    54:  my $i_spec = matches($i_word, $word, '[^a-
 50243   0.00496   0.80000    57:  my $i_vows = matches($i_word, $word,
 50243   0.00688   0.80000    53:  $i_word++;
 50243   0.48469   0.79000    62:  $count{$i_LINES}{only} =
 50243   0.48928   0.77000    60:  $count{$i_LINES}{vows} =
 50243   0.00683   0.75000    55:  my $i_only = matches($i_word, $word, '^[^a-
```

You can immediately see a slightly different focus to the subroutine profiling modules, and we start to see exactly which line of code is taking the most time. That regex line is looking a bit suspicious, for example. Remember that these tools are supposed to be used together, there is no single best way to profile your code, you need to use the best tools for the job.

See also Apache::SmallProf which hooks Devel::SmallProf into mod_perl.

51.6.4 Devel::FastProf

Devel::FastProf is another Perl line profiler. This was written with a view to getting a faster line profiler, than is possible with for example Devel::SmallProf, because it's written in C. To use Devel::FastProf, supply the -d argument to Perl:

```
$> perl -d:FastProf wordmatch -f perl5db.pl

<...multiple lines snipped...>

wordmatch report for perl5db.pl:
lines in file: 9428
words in file: 50243
words with special (non-word) characters: 20480
words with only special (non-word) characters: 7790
words with only consonants: 4801
words with only capital letters: 1316
words with only vowels: 1701
```

`Devel::FastProf` writes statistics to the file `fastprof.out` in the current directory. The output file, which can be specified, can be interpreted by using the `fprofpp` command-line program.

```
$> fprofpp | head -n20

# fprofpp output format is:
# filename:line time count: source
wordmatch:75 3.93338 251215: if ( $word =˜ /($regex)/ ) {
wordmatch:79 1.77774 251215: debug("word: $i_wd ".($has ? ’matches’ : ’does not match’)
wordmatch:81 1.47604 251215: return $has;
wordmatch:126 1.43441 260643: my $message = shift;
wordmatch:128 1.42156 260643: if ( $debug ) {
wordmatch:70 1.36824 251215: my $i_wd  = shift;
wordmatch:71 1.36739 251215: my $word  = shift;
wordmatch:72 1.35939 251215: my $regex = shift;
```

Straightaway we can see that the number of times each line has been called is identical to the `Devel::SmallProf` output, and the sequence is only very slightly different based on the ordering of the amount of time each line took to execute, `if ( $debug ) {` and `my $message = shift;`, for example. The differences in the actual times recorded might be in the algorithm used internally, or it could be due to system resource limitations or contention.

See also the `DBIx-Profile` which will profile database queries running under the `DBIx::*` namespace.

51.6.5 Devel::NYTProf

`Devel::NYTProf` is the **next generation** of Perl code profiler, fixing many shortcomings in other tools and implementing many cool features. First of all it can be used as either a *line* profiler, a *block* or a *subroutine* profiler, all at once. It can also use sub-microsecond (100ns) resolution on systems which provide `clock_gettime()`. It can be started and stopped even by the program being profiled. It's a one-line entry to profile `mod_perl` applications. It's written in `c` and is probably the fastest profiler available for Perl. The list of coolness just goes on. Enough of that, let's see how to it works - just use the familiar `-d` switch to plug it in and run the code.

```
$> perl -d:NYTProf wordmatch -f perl5db.pl
```

```
wordmatch report for perl5db.pl:
lines in file: 9427
words in file: 50243
words with special (non-word) characters: 20480
words with only special (non-word) characters: 7790
words with only consonants: 4801
words with only capital letters: 1316
words with only vowels: 1701
```

NYTProf will generate a report database into the file nytprof.out by default. Human readable reports can be generated from here by using the supplied nytprofhtml (HTML output) and nytprofcsv (CSV output) programs. We've used the Unix system html2text utility to convert the nytprof/index.html file for convenience here.

```
$> html2text nytprof/index.html

Performance Profile Index
For wordmatch
  Run on Fri Sep 26 13:46:39 2008
Reported on Fri Sep 26 13:47:23 2008

         Top 15 Subroutines -- ordered by exclusive time
|Calls |P |F |Inclusive|Exclusive|Subroutine                          |
|      |  |  |Time     |Time     |                                    |
|251215|5 |1 |13.09263 |10.47692 |main::               |matches        |
|260642|2 |1 |2.71199  |2.71199  |main::               |debug          |
|1     |1 |1 |0.21404  |0.21404  |main::               |report         |
|2     |2 |2 |0.00511  |0.00511  |XSLoader::           |load (xsub)    |
|14    |14|7 |0.00304  |0.00298  |Exporter::           |import         |
|3     |1 |1 |0.00265  |0.00254  |Exporter::           |as_heavy       |
|10    |10|4 |0.00140  |0.00140  |vars::               |import         |
|13    |13|1 |0.00129  |0.00109  |constant::           |import         |
|1     |1 |1 |0.00360  |0.00096  |FileHandle::         |import         |
|3     |3 |3 |0.00086  |0.00074  |warnings::register::|import         |
|9     |3 |1 |0.00036  |0.00036  |strict::             |bits           |
|13    |13|13|0.00032  |0.00029  |strict::             |import         |
|2     |2 |2 |0.00020  |0.00020  |warnings::           |import         |
|2     |1 |1 |0.00020  |0.00020  |Getopt::Long::       |ParseOptionSpec|
|7     |7 |6 |0.00043  |0.00020  |strict::             |unimport       |

For more information see the full list of 189 subroutines.
```

The first part of the report already shows the critical information regarding which subroutines are using the most time. The next gives some statistics about the source files profiled.

```
        Source Code Files -- ordered by exclusive time then name
|Stmts  |Exclusive|Avg.   |Reports                     |Source File         |
|       |Time     |       |                            |                    |
|2699761|15.66654 |6e-06  |line   .    block   .    sub|wordmatch           |
|35     |0.02187  |0.00062|line   .    block   .    sub|IO/Handle.pm        |
|274    |0.01525  |0.00006|line   .    block   .    sub|Getopt/Long.pm      |
|20     |0.00585  |0.00029|line   .    block   .    sub|Fcntl.pm            |
|128    |0.00340  |0.00003|line   .    block   .    sub|Exporter/Heavy.pm   |
|42     |0.00332  |0.00008|line   .    block   .    sub|IO/File.pm          |
|261    |0.00308  |0.00001|line   .    block   .    sub|Exporter.pm         |
|323    |0.00248  |8e-06  |line   .    block   .    sub|constant.pm         |
|12     |0.00246  |0.00021|line   .    block   .    sub|File/Spec/Unix.pm   |
|191    |0.00240  |0.00001|line   .    block   .    sub|vars.pm             |
|77     |0.00201  |0.00003|line   .    block   .    sub|FileHandle.pm       |
|12     |0.00198  |0.00016|line   .    block   .    sub|Carp.pm             |
|14     |0.00175  |0.00013|line   .    block   .    sub|Symbol.pm           |
|15     |0.00130  |0.00009|line   .    block   .    sub|IO.pm               |
|22     |0.00120  |0.00005|line   .    block   .    sub|IO/Seekable.pm      |
|198    |0.00085  |4e-06  |line   .    block   .    sub|warnings/register.pm|
|114    |0.00080  |7e-06  |line   .    block   .    sub|strict.pm           |
|47     |0.00068  |0.00001|line   .    block   .    sub|warnings.pm         |
|27     |0.00054  |0.00002|line   .    block   .    sub|overload.pm         |
|9      |0.00047  |0.00005|line   .    block   .    sub|SelectSaver.pm      |
|13     |0.00045  |0.00003|line   .    block   .    sub|File/Spec.pm        |
|2701595|15.73869 |       |Total                     |
|128647 |0.74946  |       |Average                   |
|       |0.00201  |0.00003|Median                    |
|       |0.00121  |0.00003|Deviation                 |

Report produced by the NYTProf 2.03 Perl profiler, developed by Tim Bunce and
Adam Kaplan.
```

At this point, if you're using the *html* report, you can click through the various links to bore down into each subroutine and each line of code. Because we're using the text reporting here, and there's a whole directory full of reports built for each source file, we'll just display a part of the corresponding wordmatch-line.html file, sufficient to give an idea of the sort of output you can expect from this cool tool.

```
$> html2text nytprof/wordmatch-line.html

Performance Profile -- -block view-.-line view-.-sub view-
For wordmatch
Run on Fri Sep 26 13:46:39 2008
Reported on Fri Sep 26 13:47:22 2008

File wordmatch

 Subroutines -- ordered by exclusive time
|Calls |P|F|Inclusive|Exclusive|Subroutine    |
|      | | |Time     |Time     |              |
|251215|5|1|13.09263 |10.47692 |main::|matches|
|260642|2|1|2.71199  |2.71199  |main::|debug  |
|1     |1|1|0.21404  |0.21404  |main::|report |
|0     |0|0|0        |0        |main::|BEGIN  |


|Line|Stmts.|Exclusive|Avg.   |Code                                         |
|    |      |Time     |       |                                             |
|1   |      |         |       |#!/usr/bin/perl                              |
|2   |      |         |       |                                             |
|    |      |         |       |use strict;                                  |
|3   |3     |0.00086  |0.00029|# spent 0.00003s making 1 calls to strict::  |
|    |      |         |       |import                                       |
|    |      |         |       |use warnings;                                |
|4   |3     |0.01563  |0.00521|# spent 0.00012s making 1 calls to warnings::|
|    |      |         |       |import                                       |
|5   |      |         |       |                                             |
|6   |      |         |       |=head1 NAME                                  |
|7   |      |         |       |                                             |
|8   |      |         |       |filewords - word analysis of input file      |
<...snip...>
|62  |1     |0.00445  |0.00445|print report( %count );                      |
|    |      |         |       |# spent 0.21404s making 1 calls to main::report|
|63  |      |         |       |                                             |
|    |      |         |       |# spent 23.56955s (10.47692+2.61571) within  |
|    |      |         |       |main::matches which was called 251215 times, |
|    |      |         |       |avg 0.00005s/call: # 50243 times             |
|    |      |         |       |(2.12134+0.51939s) at line 57 of wordmatch, avg|
|    |      |         |       |0.00005s/call # 50243 times (2.17735+0.54550s) |
|64  |      |         |       |at line 56 of wordmatch, avg 0.00005s/call #  |
|    |      |         |       |50243 times (2.10992+0.51797s) at line 58 of  |
|    |      |         |       |wordmatch, avg 0.00005s/call # 50243 times    |
|    |      |         |       |(2.12696+0.51598s) at line 55 of wordmatch, avg|
|    |      |         |       |0.00005s/call # 50243 times (1.94134+0.51687s) |
|    |      |         |       |at line 54 of wordmatch, avg 0.00005s/call    |
|    |      |         |       |sub matches {                                 |
<...snip...>
|102 |      |         |       |                                              |
|    |      |         |       |# spent 2.71199s within main::debug which was |
|    |      |         |       |called 260642 times, avg 0.00001s/call: #     |
|    |      |         |       |251215 times (2.61571+0s) by main::matches at |
|103 |      |         |       |line 74 of wordmatch, avg 0.00001s/call # 9427 |
|    |      |         |       |times (0.09628+0s) at line 50 of wordmatch, avg|
|    |      |         |       |0.00001s/call                                 |
|    |      |         |       |sub debug {                                   |
|104 |260642|0.58496  |2e-06  |my $message = shift;                          |
|105 |      |         |       |                                              |
|106 |260642|1.09917  |4e-06  |if ( $debug ) {                               |
|107 |      |         |       |print STDERR "DBG: $message\n";                 |
|108 |      |         |       |}                                               |
|109 |      |         |       |}                                               |
|110 |      |         |       |                                                |
|111 |1     |0.01501  |0.01501|exit 0;                                         |
|112 |      |         |       |                                                |
```

Oodles of very useful information in there - this seems to be the way forward.

See also `Devel::NYTProf::Apache` which hooks `Devel::NYTProf` into `mod_perl`.

51.7 SORTING

Perl modules are not the only tools a performance analyst has at their disposal, system tools like `time` should not be overlooked as the next example shows, where we take a quick look at sorting. Many books, theses and articles, have been written about efficient sorting algorithms, and this is not the place to repeat such work, there's several good sorting modules which deserve taking a look at too: `Sort::Maker`, `Sort::Key` spring to mind. However, it's still possible to make some observations on certain Perl specific interpretations on issues relating to sorting data sets and give an example or two with regard to how sorting large data volumes can effect performance. Firstly, an often overlooked point when sorting large amounts of data, one can attempt to reduce the data set to be dealt with and in many cases `grep()` can be quite useful as a simple filter:

```
@data = sort grep { /$filter/ } @incoming
```

A command such as this can vastly reduce the volume of material to actually sort through in the first place, and should not be too lightly disregarded purely on the basis of its simplicity. The `KISS` principle is too often overlooked - the next example uses the simple system `time` utility to demonstrate. Let's take a look at an actual example of sorting the contents of a large file, an apache logfile would do. This one has over a quarter of a million lines, is 50M in size, and a snippet of it looks like this:

# logfile

```
188.209-65-87.adsl-dyn.isp.belgacom.be - - [08/Feb/2007:12:57:16 +0000] "GET /favicon.i
188.209-65-87.adsl-dyn.isp.belgacom.be - - [08/Feb/2007:12:57:16 +0000] "GET /favicon.i
151.56.71.198 - - [08/Feb/2007:12:57:41 +0000] "GET /suse-on-vaio.html HTTP/1.1" 200 28
151.56.71.198 - - [08/Feb/2007:12:57:42 +0000] "GET /data/css HTTP/1.1" 404 206 "http:/
151.56.71.198 - - [08/Feb/2007:12:57:43 +0000] "GET /favicon.ico HTTP/1.1" 404 209 "-"
217.113.68.60 - - [08/Feb/2007:13:02:15 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/4.0
217.113.68.60 - - [08/Feb/2007:13:02:16 +0000] "GET /data/css HTTP/1.1" 404 206 "http:/
debora.to.isac.cnr.it - - [08/Feb/2007:13:03:58 +0000] "GET /suse-on-vaio.html HTTP/1.1
debora.to.isac.cnr.it - - [08/Feb/2007:13:03:58 +0000] "GET /data/css HTTP/1.1" 404 206
debora.to.isac.cnr.it - - [08/Feb/2007:13:03:58 +0000] "GET /favicon.ico HTTP/1.1" 404
195.24.196.99 - - [08/Feb/2007:13:26:48 +0000] "GET / HTTP/1.0" 200 3309 "-" "Mozilla/5
195.24.196.99 - - [08/Feb/2007:13:26:58 +0000] "GET /data/css HTTP/1.0" 404 206 "http:/
195.24.196.99 - - [08/Feb/2007:13:26:59 +0000] "GET /favicon.ico HTTP/1.0" 404 209 "-"
crawl1.cosmixcorp.com - - [08/Feb/2007:13:27:57 +0000] "GET /robots.txt HTTP/1.0" 200 1
crawl1.cosmixcorp.com - - [08/Feb/2007:13:28:25 +0000] "GET /links.html HTTP/1.0" 200 3
fhm226.internetdsl.tpnet.pl - - [08/Feb/2007:13:37:32 +0000] "GET /suse-on-vaio.html HT
fhm226.internetdsl.tpnet.pl - - [08/Feb/2007:13:37:34 +0000] "GET /data/css HTTP/1.1" 4
```

```
80.247.140.134 - - [08/Feb/2007:13:57:35 +0000] "GET / HTTP/1.1" 200 3309 "-" "Mozilla/
80.247.140.134 - - [08/Feb/2007:13:57:37 +0000] "GET /data/css HTTP/1.1" 404 206 "http:
pop.compuscan.co.za - - [08/Feb/2007:14:10:43 +0000] "GET / HTTP/1.1" 200 3309 "-" "www
livebot-207-46-98-57.search.live.com - - [08/Feb/2007:14:12:04 +0000] "GET /robots.txt
livebot-207-46-98-57.search.live.com - - [08/Feb/2007:14:12:04 +0000] "GET /html/oracle
dslb-088-064-005-154.pools.arcor-ip.net - - [08/Feb/2007:14:12:15 +0000] "GET / HTTP/1.
196.201.92.41 - - [08/Feb/2007:14:15:01 +0000] "GET / HTTP/1.1" 200 3309 "-" "MOT-L7/08
```

The specific task here is to sort the 286,525 lines of this file by Response Code, Query, Browser, Referring Url, and lastly Date. One solution might be to use the following code, which iterates over the files given on the command-line.

# sort-apache-log

```
#!/usr/bin/perl -n

use strict;
use warnings;

my @data;

LINE:
while ( <> ) {
    my $line = $_;
    if (
        $line =~ m/^(
            ([\w\.\-]+)              # client
            \s*-\s*-\s*\[
            ([^]]+)                  # date
            \]\s*"\w+\s*
            (\S+)                    # query
            [^"]+"\s*
            (\d+)                    # status
            \s+\S+\s+"[^"]*"\s+"
            ([^"]*)                  # browser
            "
            .*
        )$/x
    ) {
        my @chunks = split(/ +/, $line);
        my $ip      = $1;
        my $date    = $2;
        my $query   = $3;
        my $status  = $4;
        my $browser = $5;

        push(@data, [$ip, $date, $query, $status, $browser, $line]);
    }
}
```

```
my @sorted = sort {
    $a->[3] cmp $b->[3]
            ||
    $a->[2] cmp $b->[2]
            ||
    $a->[0] cmp $b->[0]
            ||
    $a->[1] cmp $b->[1]
            ||
    $a->[4] cmp $b->[4]
} @data;

foreach my $data ( @sorted ) {
    print $data->[5];
}

exit 0;
```

When running this program, redirect `STDOUT` so it is possible to check the output is correct from following test runs and use the system `time` utility to check the overall runtime.

```
$> time ./sort-apache-log logfile > out-sort

real    0m17.371s
user    0m15.757s
sys     0m0.592s
```

The program took just over 17 wallclock seconds to run. Note the different values `time` outputs, it's important to always use the same one, and to not confuse what each one means.

Elapsed Real Time
: The overall, or wallclock, time between when `time` was called, and when it terminates. The elapsed time includes both user and system times, and time spent waiting for other users and processes on the system. Inevitably, this is the most approximate of the measurements given.

User CPU Time
: The user time is the amount of time the entire process spent on behalf of the user on this system executing this program.

System CPU Time
: The system time is the amount of time the kernel itself spent executing routines, or system calls, on behalf of this process user.

Running this same process as a `Schwarzian Transform` it is possible to eliminate the input and output arrays for storing all the data, and work on the input directly as it arrives too. Otherwise, the code looks fairly similar:

# sort-apache-log-schwarzian

```
#!/usr/bin/perl -n
```

```
use strict;
use warnings;

print

    map $_->[0] =>

    sort {
        $a->[4] cmp $b->[4]
                ||
        $a->[3] cmp $b->[3]
                ||
        $a->[1] cmp $b->[1]
                ||
        $a->[2] cmp $b->[2]
                ||
        $a->[5] cmp $b->[5]
    }
    map  [ $_, m/^(
        ([\w\.\-]+)             # client
        \s*-\s*-\s*\[
        ([^]]+)                 # date
        \]\s*"\w+\s*
        (\S+)                   # query
        [^"]+"\s*
        (\d+)                   # status
        \s+\S+\s+"[^"]*"\s+"
        ([^"]*)                 # browser
        "
        .*
    )$/xo ]

    => <>;

exit 0;
```

Run the new code against the same logfile, as above, to check the new time.

```
$> time ./sort-apache-log-schwarzian logfile > out-schwarz

real    0m9.664s
user    0m8.873s
sys     0m0.704s
```

The time has been cut in half, which is a respectable speed improvement by any standard. Naturally, it is important to check the output is consistent with the first program run, this is where the Unix system `cksum` utility comes in.

```
$> cksum out-sort out-schwarz
3044173777 52029194 out-sort
```

```
3044173777 52029194 out-schwarz
```

BTW. Beware too of pressure from managers who see you speed a program up by 50% of the runtime once, only to get a request one month later to do the same again (true story) - you'll just have to point out you're only human, even if you are a Perl programmer, and you'll see what you can do...

51.8 LOGGING

An essential part of any good development process is appropriate error handling with appropriately informative messages, however there exists a school of thought which suggests that log files should be *chatty*, as if the chain of unbroken output somehow ensures the survival of the program. If speed is in any way an issue, this approach is wrong.

A common sight is code which looks something like this:

```
logger->debug( "A logging message via process-id: $$ INC: " . Dumper(\%INC) )
```

The problem is that this code will always be parsed and executed, even when the debug level set in the logging configuration file is zero. Once the debug() subroutine has been entered, and the internal `$debug` variable confirmed to be zero, for example, the message which has been sent in will be discarded and the program will continue. In the example given though, the `\%INC` hash will already have been dumped, and the message string constructed, all of which work could be bypassed by a debug variable at the statement level, like this:

```
logger->debug( "A logging message via process-id: $$ INC: " . Dumper(\%INC) ) if $DEBUG
```

This effect can be demonstrated by setting up a test script with both forms, including a `debug()` subroutine to emulate typical `logger()` functionality.

# ifdebug

```
#!/usr/bin/perl

use strict;
use warnings;

use Benchmark;
use Data::Dumper;
my $DEBUG = 0;

sub debug {
    my $msg = shift;

    if ( $DEBUG ) {
        print "DEBUG: $msg\n";
    }
};

timethese(100000, {
        'debug'       => sub {
            debug( "A $0 logging message via process-id: $$" . Dumper(\%INC) )
        },
        'ifdebug'  => sub {
            debug( "A $0 logging message via process-id: $$" . Dumper(\%INC) ) if $DEBU
        },
});
```

Let's see what `Benchmark` makes of this:

```
$> perl ifdebug
Benchmark: timing 100000 iterations of constant, sub...
   ifdebug:  0 wallclock secs ( 0.01 usr +  0.00 sys =  0.01 CPU) @ 10000000.00/s (n=10
            (warning: too few iterations for a reliable count)
     debug: 14 wallclock secs (13.18 usr +  0.04 sys = 13.22 CPU) @ 7564.30/s (n=100000
```

In the one case the code, which does exactly the same thing as far as outputting any debugging information is concerned, in other words nothing, takes 14 seconds, and in the other case the code takes one hundredth of a second. Looks fairly definitive. Use a `$DEBUG` variable BEFORE you call the subroutine, rather than relying on the smart functionality inside it.

51.8.1 Logging if DEBUG (constant)

It's possible to take the previous idea a little further, by using a compile time `DEBUG` constant.

# ifdebug-constant

```
#!/usr/bin/perl

use strict;
use warnings;

use Benchmark;
use Data::Dumper;
use constant
    DEBUG => 0
;

sub debug {
    if ( DEBUG ) {
        my $msg = shift;
        print "DEBUG: $msg\n";
    }
};

timethese(100000, {
        'debug'       => sub {
            debug( "A $0 logging message via process-id: $$" . Dumper(\%INC) )
        },
        'constant'  => sub {
            debug( "A $0 logging message via process-id: $$" . Dumper(\%INC) ) if DEBUG
        },
});
```

Running this program produces the following output:

```
$> perl ifdebug-constant
Benchmark: timing 100000 iterations of constant, sub...
  constant:  0 wallclock secs (-0.00 usr +  0.00 sys = -0.00 CPU) @ -720575940379279360
            (warning: too few iterations for a reliable count)
       sub: 14 wallclock secs (13.09 usr +  0.00 sys = 13.09 CPU) @ 7639.42/s (n=100000
```

The DEBUG constant wipes the floor with even the $debug variable, clocking in at minus zero seconds, and generates a "warning: too few iterations for a reliable count" message into the bargain. To see what is really going on, and why we had too few iterations when we thought we asked for 100000, we can use the very useful B::Deparse to inspect the new code:

```
$> perl -MO=Deparse ifdebug-constant

use Benchmark;
use Data::Dumper;
use constant ('DEBUG', 0);
sub debug {
    use warnings;
    use strict 'refs';
    0;
}
use warnings;
use strict 'refs';
timethese(100000, {'sub', sub {
    debug "A $0 logging message via process-id: $$" . Dumper(\%INC);
}
, 'constant', sub {
    0;
}
});
ifdebug-constant syntax OK
```

The output shows the constant() subroutine we're testing being replaced with the value of the DEBUG constant: zero. The line to be tested has been completely optimized away, and you can't get much more efficient than that.

51.9 POSTSCRIPT

This document has provided several way to go about identifying hot-spots, and checking whether any modifications have improved the runtime of the code.

As a final thought, remember that it's not (at the time of writing) possible to produce a useful program which will run in zero or negative time and this basic principle can be written as: *useful programs are slow* by their very definition. It is of course possible to write a nearly instantaneous program, but it's not going to do very much, here's a very efficient one:

```
$> perl -e 0
```

Optimizing that any further is a job for p5p.

51.10 SEE ALSO

Further reading can be found using the modules and links below.

51.10.1 PERLDOCS

For example: `perldoc -f sort`.

`perlfaq4`.

Section 23.1 [perlfork NAME], page 337, Section 25.1 [perlfunc NAME], page 351, Section 68.1 [perlretut NAME], page 1131, Section 75.1 [perlthrtut NAME], page 1269.

`threads`.

51.10.2 MAN PAGES

`time`.

51.10.3 MODULES

It's not possible to individually showcase all the performance related code for Perl here, naturally, but here's a short list of modules from the CPAN which deserve further attention.

```
Apache::DProf
Apache::SmallProf
Benchmark
DBIx::Profile
Devel::AutoProfiler
Devel::DProf
Devel::DProfLB
Devel::FastProf
Devel::GraphVizProf
Devel::NYTProf
Devel::NYTProf::Apache
Devel::Profiler
Devel::Profile
Devel::Profit
Devel::SmallProf
Devel::WxProf
POE::Devel::Profiler
Sort::Key
Sort::Maker
```

51.10.4 URLS

Very useful online reference material:

```
http://www.ccl4.org/~nick/P/Fast_Enough/

http://www-128.ibm.com/developerworks/library/l-optperl.html

http://perlbuzz.com/2007/11/bind-output-variables-in-dbi-for-speed-and-safety.html

http://en.wikipedia.org/wiki/Performance_analysis
```

http://apache.perl.org/docs/1.0/guide/performance.html

http://perlgolf.sourceforge.net/

http://www.sysarch.com/Perl/sort_paper.html

51.11 AUTHOR

Richard Foley <richard.foley@rfi.net> Copyright (c) 2008

52 perlpod

52.1 NAME

perlpod - the Plain Old Documentation format

52.2 DESCRIPTION

Pod is a simple-to-use markup language used for writing documentation for Perl, Perl programs, and Perl modules.

Translators are available for converting Pod to various formats like plain text, HTML, man pages, and more.

Pod markup consists of three basic kinds of paragraphs: Section 52.2.1 [ordinary], page 900, Section 52.2.2 [verbatim], page 900, and Section 52.2.3 [command], page 900.

52.2.1 Ordinary Paragraph

Most paragraphs in your documentation will be ordinary blocks of text, like this one. You can simply type in your text without any markup whatsoever, and with just a blank line before and after. When it gets formatted, it will undergo minimal formatting, like being rewrapped, probably put into a proportionally spaced font, and maybe even justified.

You can use formatting codes in ordinary paragraphs, for **bold**, *italic*, `code-style`, `perlfaq`, and more. Such codes are explained in the "Section 52.2.4 [Formatting Codes], page 904" section, below.

52.2.2 Verbatim Paragraph

Verbatim paragraphs are usually used for presenting a codeblock or other text which does not require any special parsing or formatting, and which shouldn't be wrapped.

A verbatim paragraph is distinguished by having its first character be a space or a tab. (And commonly, all its lines begin with spaces and/or tabs.) It should be reproduced exactly, with tabs assumed to be on 8-column boundaries. There are no special formatting codes, so you can't italicize or anything like that. A \ means \, and nothing else.

52.2.3 Command Paragraph

A command paragraph is used for special treatment of whole chunks of text, usually as headings or parts of lists.

All command paragraphs (which are typically only one line long) start with "=", followed by an identifier, followed by arbitrary text that the command can use however it pleases. Currently recognized commands are

```
=pod
=head1 Heading Text
=head2 Heading Text
=head3 Heading Text
=head4 Heading Text
=over indentlevel
=item stuff
```

```
=back
=begin format
=end format
=for format text...
=encoding type
=cut
```

To explain them each in detail:

`=head1` *`Heading Text`*
`=head2` *`Heading Text`*
`=head3` *`Heading Text`*
`=head4` *`Heading Text`*

Head1 through head4 produce headings, head1 being the highest level. The text in the rest of this paragraph is the content of the heading. For example:

```
=head2 Object Attributes
```

The text "Object Attributes" comprises the heading there. The text in these heading commands can use formatting codes, as seen here:

```
=head2 Possible Values for C<$/>
```

Such commands are explained in the "Section 52.2.4 [Formatting Codes], page 904" section, below.

`=over` *`indentlevel`*
`=item` *`stuff...`*
`=back`

Item, over, and back require a little more explanation: "=over" starts a region specifically for the generation of a list using "=item" commands, or for indenting (groups of) normal paragraphs. At the end of your list, use "=back" to end it. The *indentlevel* option to "=over" indicates how far over to indent, generally in ems (where one em is the width of an "M" in the document's base font) or roughly comparable units; if there is no *indentlevel* option, it defaults to four. (And some formatters may just ignore whatever *indentlevel* you provide.) In the *stuff* in `=item` *`stuff...`*, you may use formatting codes, as seen here:

```
=item Using C<$|> to Control Buffering
```

Such commands are explained in the "Section 52.2.4 [Formatting Codes], page 904" section, below.

Note also that there are some basic rules to using "=over" ... "=back" regions:

- Don't use "=item"s outside of an "=over" ... "=back" region.
- The first thing after the "=over" command should be an "=item", unless there aren't going to be any items at all in this "=over" ... "=back" region.
- Don't put "=head*n*" commands inside an "=over" ... "=back" region.
- And perhaps most importantly, keep the items consistent: either use "=item *" for all of them, to produce bullets; or use "=item 1.", "=item 2.", etc., to produce numbered lists; or use "=item foo", "=item bar", etc.–namely, things that look nothing like bullets or numbers.

  If you start with bullets or numbers, stick with them, as formatters use the first "=item" type to decide how to format the list.

`=cut`

To end a Pod block, use a blank line, then a line beginning with "=cut", and a blank line after it. This lets Perl (and the Pod formatter) know that this is where Perl code is resuming. (The blank line before the "=cut" is not technically necessary, but many older Pod processors require it.)

`=pod`

The "=pod" command by itself doesn't do much of anything, but it signals to Perl (and Pod formatters) that a Pod block starts here. A Pod block starts with *any* command paragraph, so a "=pod" command is usually used just when you want to start a Pod block with an ordinary paragraph or a verbatim paragraph. For example:

```
=item stuff()

This function does stuff.

=cut

sub stuff {
  ...
}

=pod

Remember to check its return value, as in:

  stuff() || die "Couldn't do stuff!";

=cut
```

`=begin` *`formatname`*
`=end` *`formatname`*
`=for` *`formatname text...`*

For, begin, and end will let you have regions of text/code/data that are not generally interpreted as normal Pod text, but are passed directly to particular formatters, or are otherwise special. A formatter that can use that format will use the region, otherwise it will be completely ignored.

A command "=begin *formatname*", some paragraphs, and a command "=end *formatname*", mean that the text/data in between is meant for formatters that understand the special format called *formatname*. For example,

```
=begin html

<hr> <img src="thang.png">
<p> This is a raw HTML paragraph </p>

=end html
```

The command "=for *formatname text...*" specifies that the remainder of just this paragraph (starting right after *formatname*) is in that special format.

```
=for html <hr> <img src="thang.png">
<p> This is a raw HTML paragraph </p>
```

This means the same thing as the above "=begin html" ... "=end html" region.

That is, with "=for", you can have only one paragraph's worth of text (i.e., the text in "=foo targetname text..."), but with "=begin targetname" ... "=end targetname", you can have any amount of stuff in between. (Note that there still must be a blank line after the "=begin" command and a blank line before the "=end" command.)

Here are some examples of how to use these:

```
=begin html

<br>Figure 1.<br><IMG SRC="figure1.png"><br>

=end html

=begin text

  ---------------
  |  foo        |
  |        bar  |
  ---------------

^^^^ Figure 1. ^^^^

=end text
```

Some format names that formatters currently are known to accept include "roff", "man", "latex", "tex", "text", and "html". (Some formatters will treat some of these as synonyms.)

A format name of "comment" is common for just making notes (presumably to yourself) that won't appear in any formatted version of the Pod document:

```
=for comment
Make sure that all the available options are documented!
```

Some *formatnames* will require a leading colon (as in "=for :formatname", or "=begin :formatname" ... "=end :formatname"), to signal that the text is not raw data, but instead *is* Pod text (i.e., possibly containing formatting codes) that's just not for normal formatting (e.g., may not be a normal-use paragraph, but might be for formatting as a footnote).

=encoding *encodingname*

This command is used for declaring the encoding of a document. Most users won't need this; but if your encoding isn't US-ASCII, then put a =encoding *encodingname* command very early in the document so that pod formatters will know how to decode the document. For *encodingname*, use a name recognized by the `Encode-Supported` module. Some pod formatters may try to guess between a Latin-1 or CP-1252 versus UTF-8 encoding, but they may guess wrong. It's best to be explicit if you use anything besides strict ASCII. Examples:

```
  =encoding latin1

  =encoding utf8

  =encoding koi8-r

  =encoding ShiftJIS

  =encoding big5
```

`=encoding` affects the whole document, and must occur only once.

And don't forget, all commands but `=encoding` last up until the end of its *paragraph*, not its line. So in the examples below, you can see that every command needs the blank line after it, to end its paragraph. (And some older Pod translators may require the `=encoding` line to have a following blank line as well, even though it should be legal to omit.)

Some examples of lists include:

```
  =over

  =item *

  First item

  =item *

  Second item

  =back

  =over

  =item Foo()

  Description of Foo function

  =item Bar()

  Description of Bar function

  =back
```

52.2.4 Formatting Codes

In ordinary paragraphs and in some command paragraphs, various formatting codes (a.k.a. "interior sequences") can be used:

`I<text>` – italic text `<> >>`

Used for emphasis (`"be I<careful!>"`) and parameters (`"redo I<LABEL>"`)

`B<text>` – bold text `<> >>`

Used for switches (`"perl's B<-n> switch"`), programs (`"some systems provide a B<chfn> for that"`), emphasis (`"be B<careful!>"`), and so on (`"and that feature is known as B<autovivification>"`).

`C<code>` – code text `<> >>`

Renders code in a typewriter font, or gives some other indication that this represents program text (`"C<gmtime($^T)>"`) or some other form of computerese (`"C<drwxr-xr-x>"`).

`L<name>` – a hyperlink `<> >>`

There are various syntaxes, listed below. In the syntaxes given, `text`, `name`, and `section` cannot contain the characters '/' and '|'; and any '<' or '>' should be matched.

- `L<name>`

  Link to a Perl manual page (e.g., `L<Net::Ping>`). Note that `name` should not contain spaces. This syntax is also occasionally used for references to Unix man pages, as in `L<crontab(5)>`.

- `L<name/"sec">` or `L<name/sec>`

  Link to a section in other manual page. E.g., `L<perlsyn/"For Loops">`

- `L</"sec">` or `L</sec>`

  Link to a section in this manual page. E.g., `L</"Object Methods">`

A section is started by the named heading or item. For example, `L<perlvar/$.>` or `L<perlvar/"$.">` both link to the section started by `"=item $."` in perlvar. And `L<perlsyn/For Loops>` or `L<perlsyn/"For Loops">` both link to the section started by `"=head2 For Loops"` in perlsyn.

To control what text is used for display, you use `"L<text|...>"`, as in:

- `L<text|name>`

  Link this text to that manual page. E.g., `L<Perl Error Messages|perldiag>`

- `L<text|name/"sec">` or `L<text|name/sec>`

  Link this text to that section in that manual page. E.g., `L<postfix "if"|perlsyn/"Statement Modifiers">`

- `L<text|/"sec">` or `L<text|/sec>` or `L<text|"sec">`

  Link this text to that section in this manual page. E.g., `L<the various attributes|/"Member Data">`

Or you can link to a web page:

- `L<scheme:...>`

  `L<text|scheme:...>`

  Links to an absolute URL. For example, `L<http://www.perl.org/>` or `L<The Perl Home Page|http://www.perl.org/>`.

`E<escape>` – a character escape `<> >>`

Very similar to HTML/XML `&`*`foo`*`;` "entity references":

- `E<lt>` – a literal `<` (less than)
- `E<gt>` – a literal `>` (greater than)
- `E<verbar>` – a literal `|` (*ver*tical *bar*)
- `E<sol>` – a literal / (*sol*idus)

  The above four are optional except in other formatting codes, notably `L<...>`, and when preceded by a capital letter.

- `E<htmlname>`

  Some non-numeric HTML entity name, such as `E<eacute>`, meaning the same thing as `&eacute;` in HTML – i.e., a lowercase e with an acute (/-shaped) accent.

- `E<number>`

  The ASCII/Latin-1/Unicode character with that number. A leading "0x" means that *number* is hex, as in `E<0x201E>`. A leading "0" means that *number* is octal, as in `E<075>`. Otherwise *number* is interpreted as being in decimal, as in `E<181>`.

  Note that older Pod formatters might not recognize octal or hex numeric escapes, and that many formatters cannot reliably render characters above 255. (Some formatters may even have to use compromised renderings of Latin-1/CP-1252 characters, like rendering `E<eacute>` as just a plain "e".)

`F<filename>` – used for filenames `<> >>`

Typically displayed in italics. Example: "`F<.cshrc>`"

`S<text>` – text contains non-breaking spaces `<> >>`

This means that the words in *text* should not be broken across lines. Example: `S<$x ? $y : $z>`.

`X<topic name>` – an index entry `<> >>`

This is ignored by most formatters, but some may use it for building indexes. It always renders as empty-string. Example: `X<absolutizing relative URLs>`

`Z<>` – a null (zero-effect) formatting code `<> >>`

This is rarely used. It's one way to get around using an E<...> code sometimes. For example, instead of "`NE<lt>3`" (for "N<3") you could write "`NZ<><3`" (the "Z<>" breaks up the "N" and the "<" so they can't be considered the part of a (fictitious) "N<...>" code).

Most of the time, you will need only a single set of angle brackets to delimit the beginning and end of formatting codes. However, sometimes you will want to put a real right angle bracket (a greater-than sign, '>') inside of a formatting code. This is particularly common when using a formatting code to provide a different font-type for a snippet of code. As with all things in Perl, there is more than one way to do it. One way is to simply escape the closing bracket using an `E` code:

```
C<$a E<lt>=E<gt> $b>
```

This will produce: "`$a <=> $b`"

A more readable, and perhaps more "plain" way is to use an alternate set of delimiters that doesn't require a single ">" to be escaped. Doubled angle brackets ("<<" and ">>") may be used *if and only if there is whitespace right after the opening delimiter and whitespace right before the closing delimiter!* For example, the following will do the trick:

```
C<< $a <=> $b >>
```

In fact, you can use as many repeated angle-brackets as you like so long as you have the same number of them in the opening and closing delimiters, and make sure that whitespace immediately follows the last '<' of the opening delimiter, and immediately precedes the first '>' of the closing delimiter. (The whitespace is ignored.) So the following will also work:

```
C<<< $a <=> $b >>>
C<<<<  $a <=> $b     >>>>
```

And they all mean exactly the same as this:

```
C<$a E<lt>=E<gt> $b>
```

The multiple-bracket form does not affect the interpretation of the contents of the formatting code, only how it must end. That means that the examples above are also exactly the same as this:

```
C<< $a E<lt>=E<gt> $b >>
```

As a further example, this means that if you wanted to put these bits of code in `C` (code) style:

```
open(X, ">>thing.dat") || die $!
$foo->bar();
```

you could do it like so:

```
C<<< open(X, ">>thing.dat") || die $! >>>
C<< $foo->bar(); >>
```

which is presumably easier to read than the old way:

```
C<open(X, "E<gt>E<gt>thing.dat") || die $!>
C<$foo-E<gt>bar();>
```

This is currently supported by pod2text (Pod::Text), pod2man (Pod::Man), and any other pod2xxx or Pod::Xxxx translators that use Pod::Parser 1.093 or later, or Pod::Tree 1.02 or later.

52.2.5 The Intent

The intent is simplicity of use, not power of expression. Paragraphs look like paragraphs (block format), so that they stand out visually, and so that I could run them through `fmt` easily to reformat them (that's F7 in my version of **vi**, or Esc Q in my version of **emacs**). I wanted the translator to always leave the `'` and `` ` `` and `"` quotes alone, in verbatim mode, so I could slurp in a working program, shift it over four spaces, and have it print out, er, verbatim. And presumably in a monospace font.

The Pod format is not necessarily sufficient for writing a book. Pod is just meant to be an idiot-proof common source for nroff, HTML, TeX, and other markup languages, as used for online documentation. Translators exist for **pod2text**, **pod2html**, **pod2man** (that's for nroff(1) and troff(1)), **pod2latex**, and **pod2fm**. Various others are available in CPAN.

52.2.6 Embedding Pods in Perl Modules

You can embed Pod documentation in your Perl modules and scripts. Start your documentation with an empty line, a "=head1" command at the beginning, and end it with a "=cut" command and an empty line. The **perl** executable will ignore the Pod text. You can place a Pod statement where **perl** expects the beginning of a new statement, but not within a statement, as that would result in an error. See any of the supplied library modules for examples.

If you're going to put your Pod at the end of the file, and you're using an `__END__` or `__DATA__` cut mark, make sure to put an empty line there before the first Pod command.

```
__END__

=head1 NAME

Time::Local - efficiently compute time from local and GMT time
```

Without that empty line before the "=head1", many translators wouldn't have recognized the "=head1" as starting a Pod block.

52.2.7 Hints for Writing Pod

- The **podchecker** command is provided for checking Pod syntax for errors and warnings. For example, it checks for completely blank lines in Pod blocks and for unknown commands and formatting codes. You should still also pass your document through one or more translators and proofread the result, or print out the result and proofread that. Some of the problems found may be bugs in the translators, which you may or may not wish to work around.
- If you're more familiar with writing in HTML than with writing in Pod, you can try your hand at writing documentation in simple HTML, and converting it to Pod with the experimental `Pod-HTML2Pod` module, (available in CPAN), and looking at the resulting code. The experimental `Pod-PXML` module in CPAN might also be useful.
- Many older Pod translators require the lines before every Pod command and after every Pod command (including "=cut"!) to be a blank line. Having something like this:

  ```
  # - - - - - - - - - - - -
  =item $firecracker->boom()

  This noisily detonates the firecracker object.
  =cut
  sub boom {
  ...
  ```

  ...will make such Pod translators completely fail to see the Pod block at all.

  Instead, have it like this:

  ```
  # - - - - - - - - - - - -

  =item $firecracker->boom()

  This noisily detonates the firecracker object.
  ```

```
=cut

sub boom {
...
```

- Some older Pod translators require paragraphs (including command paragraphs like "=head2 Functions") to be separated by *completely* empty lines. If you have an apparently empty line with some spaces on it, this might not count as a separator for those translators, and that could cause odd formatting.
- Older translators might add wording around an L<> link, so that `L<Foo::Bar>` may become "the Foo::Bar manpage", for example. So you shouldn't write things like `the L<foo> documentation`, if you want the translated document to read sensibly. Instead, write `the L<Foo::Bar|Foo::Bar> documentation` or `L<the Foo::Bar documentation|Foo::Bar>`, to control how the link comes out.
- Going past the 70th column in a verbatim block might be ungracefully wrapped by some formatters.

52.3 SEE ALSO

Section 53.1 [perlpodspec NAME], page 910, Section 74.2.14 [perlsyn PODs: Embedded Documentation], page 1261, Section 44.1 [perlnewmod NAME], page 760, `perldoc`, `pod2html`, `pod2man`, `podchecker`.

52.4 AUTHOR

Larry Wall, Sean M. Burke

53 perlpodspec

53.1 NAME

perlpodspec - Plain Old Documentation: format specification and notes

53.2 DESCRIPTION

This document is detailed notes on the Pod markup language. Most people will only have to read Section 52.1 [perlpod], page 900 to know how to write in Pod, but this document may answer some incidental questions to do with parsing and rendering Pod.

In this document, "must" / "must not", "should" / "should not", and "may" have their conventional (cf. RFC 2119) meanings: "X must do Y" means that if X doesn't do Y, it's against this specification, and should really be fixed. "X should do Y" means that it's recommended, but X may fail to do Y, if there's a good reason. "X may do Y" is merely a note that X can do Y at will (although it is up to the reader to detect any connotation of "and I think it would be *nice* if X did Y" versus "it wouldn't really *bother* me if X did Y").

Notably, when I say "the parser should do Y", the parser may fail to do Y, if the calling application explicitly requests that the parser *not* do Y. I often phrase this as "the parser should, by default, do Y." This doesn't *require* the parser to provide an option for turning off whatever feature Y is (like expanding tabs in verbatim paragraphs), although it implicates that such an option *may* be provided.

53.3 Pod Definitions

Pod is embedded in files, typically Perl source files, although you can write a file that's nothing but Pod.

A **line** in a file consists of zero or more non-newline characters, terminated by either a newline or the end of the file.

A **newline sequence** is usually a platform-dependent concept, but Pod parsers should understand it to mean any of CR (ASCII 13), LF (ASCII 10), or a CRLF (ASCII 13 followed immediately by ASCII 10), in addition to any other system-specific meaning. The first CR/CRLF/LF sequence in the file may be used as the basis for identifying the newline sequence for parsing the rest of the file.

A **blank line** is a line consisting entirely of zero or more spaces (ASCII 32) or tabs (ASCII 9), and terminated by a newline or end-of-file. A **non-blank line** is a line containing one or more characters other than space or tab (and terminated by a newline or end-of-file).

(*Note:* Many older Pod parsers did not accept a line consisting of spaces/tabs and then a newline as a blank line. The only lines they considered blank were lines consisting of *no characters at all*, terminated by a newline.)

Whitespace is used in this document as a blanket term for spaces, tabs, and newline sequences. (By itself, this term usually refers to literal whitespace. That is, sequences of whitespace characters in Pod source, as opposed to "E<32>", which is a formatting code that *denotes* a whitespace character.)

A **Pod parser** is a module meant for parsing Pod (regardless of whether this involves calling callbacks or building a parse tree or directly formatting it). A **Pod formatter** (or

Pod translator) is a module or program that converts Pod to some other format (HTML, plaintext, TeX, PostScript, RTF). A **Pod processor** might be a formatter or translator, or might be a program that does something else with the Pod (like counting words, scanning for index points, etc.).

Pod content is contained in **Pod blocks**. A Pod block starts with a line that matches `m/\A=[a-zA-Z]/`, and continues up to the next line that matches `m/\A=cut/` or up to the end of the file if there is no `m/\A=cut/` line.

Within a Pod block, there are **Pod paragraphs**. A Pod paragraph consists of non-blank lines of text, separated by one or more blank lines.

For purposes of Pod processing, there are four types of paragraphs in a Pod block:

- A command paragraph (also called a "directive"). The first line of this paragraph must match `m/\A=[a-zA-Z]/`. Command paragraphs are typically one line, as in:

  ```
  =head1 NOTES

  =item *
  ```

  But they may span several (non-blank) lines:

  ```
  =for comment
  Hm, I wonder what it would look like if
  you tried to write a BNF for Pod from this.

  =head3 Dr. Strangelove, or: How I Learned to
  Stop Worrying and Love the Bomb
  ```

  Some command paragraphs allow formatting codes in their content (i.e., after the part that matches `m/\A=[a-zA-Z]\S*\s*/`), as in:

  ```
  =head1 Did You Remember to C<use strict;>?
  ```

  In other words, the Pod processing handler for "head1" will apply the same processing to "Did You Remember to C<use strict;>?" that it would to an ordinary paragraph (i.e., formatting codes like "C<...>") are parsed and presumably formatted appropriately, and whitespace in the form of literal spaces and/or tabs is not significant.

- A **verbatim paragraph**. The first line of this paragraph must be a literal space or tab, and this paragraph must not be inside a "=begin *identifier*", ... "=end *identifier*" sequence unless "*identifier*" begins with a colon (":"). That is, if a paragraph starts with a literal space or tab, but *is* inside a "=begin *identifier*", ... "=end *identifier*" region, then it's a data paragraph, unless "*identifier*" begins with a colon.

  Whitespace *is* significant in verbatim paragraphs (although, in processing, tabs are probably expanded).

- An **ordinary paragraph**. A paragraph is an ordinary paragraph if its first line matches neither `m/\A=[a-zA-Z]/` nor `m/\A[ \t]/`, *and* if it's not inside a "=begin *identifier*", ... "=end *identifier*" sequence unless "*identifier*" begins with a colon (":").

- A **data paragraph**. This is a paragraph that *is* inside a "=begin *identifier*" ... "=end *identifier*" sequence where "*identifier*" does *not* begin with a literal colon (":"). In some sense, a data paragraph is not part of Pod at all (i.e., effectively it's "out-of-band"), since it's not subject to most kinds of Pod parsing; but it is specified here, since Pod parsers need to be able to call an event for it, or store it in some form in a parse tree, or at least just parse *around* it.

For example: consider the following paragraphs:

```
# <- that's the 0th column

=head1 Foo

Stuff

  $foo->bar

=cut
```

Here, "=head1 Foo" and "=cut" are command paragraphs because the first line of each matches `m/\A=[a-zA-Z]/`. "*[space][space]*$foo->bar" is a verbatim paragraph, because its first line starts with a literal whitespace character (and there's no "=begin"..."=end" region around).

The "=begin *identifier*" ... "=end *identifier*" commands stop paragraphs that they surround from being parsed as ordinary or verbatim paragraphs, if *identifier* doesn't begin with a colon. This is discussed in detail in the section Section 53.9 [About Data Paragraphs and "=begin/=end" Regions], page 932.

53.4 Pod Commands

This section is intended to supplement and clarify the discussion in Section 52.2.3 [perlpod Command Paragraph], page 900. These are the currently recognized Pod commands:

"=head1", "=head2", "=head3", "=head4"

This command indicates that the text in the remainder of the paragraph is a heading. That text may contain formatting codes. Examples:

```
=head1 Object Attributes

=head3 What B<Not> to Do!
```

"=pod"

This command indicates that this paragraph begins a Pod block. (If we are already in the middle of a Pod block, this command has no effect at all.) If there is any text in this command paragraph after "=pod", it must be ignored. Examples:

```
=pod

This is a plain Pod paragraph.

=pod This text is ignored.
```

"=cut"

This command indicates that this line is the end of this previously started Pod block. If there is any text after "=cut" on the line, it must be ignored. Examples:

```
=cut

=cut The documentation ends here.

=cut
# This is the first line of program text.
sub foo { # This is the second.
```

It is an error to try to *start* a Pod block with a "=cut" command. In that case, the Pod processor must halt parsing of the input file, and must by default emit a warning.

"=over"

This command indicates that this is the start of a list/indent region. If there is any text following the "=over", it must consist of only a nonzero positive numeral. The semantics of this numeral is explained in the Section 53.8 [About =over...=back Regions], page 928 section, further below. Formatting codes are not expanded. Examples:

```
=over 3

=over 3.5

=over
```

"=item"

This command indicates that an item in a list begins here. Formatting codes are processed. The semantics of the (optional) text in the remainder of this paragraph are explained in the Section 53.8 [About =over...=back Regions], page 928 section, further below. Examples:

```
=item

=item *

=item      *

=item 14

=item   3.

=item C<< $thing->stuff(I<dodad>) >>

=item For transporting us beyond seas to be tried for pretended
offenses

=item He is at this time transporting large armies of foreign
mercenaries to complete the works of death, desolation and
tyranny, already begun with circumstances of cruelty and perfidy
scarcely paralleled in the most barbarous ages, and totally
```

```
        unworthy the head of a civilized nation.
```

"=back"

This command indicates that this is the end of the region begun by the most recent "=over" command. It permits no text after the "=back" command.

"=begin formatname"
"=begin formatname parameter"

This marks the following paragraphs (until the matching "=end formatname") as being for some special kind of processing. Unless "formatname" begins with a colon, the contained non-command paragraphs are data paragraphs. But if "formatname" *does* begin with a colon, then non-command paragraphs are ordinary paragraphs or data paragraphs. This is discussed in detail in the section Section 53.9 [About Data Paragraphs and "=begin/=end" Regions], page 932.

It is advised that formatnames match the regexp `m/\A:?[-a-zA-Z0-9_]+\z/`. Everything following whitespace after the formatname is a parameter that may be used by the formatter when dealing with this region. This parameter must not be repeated in the "=end" paragraph. Implementors should anticipate future expansion in the semantics and syntax of the first parameter to "=begin"/"=end"/"=for".

"=end formatname"

This marks the end of the region opened by the matching "=begin formatname" region. If "formatname" is not the formatname of the most recent open "=begin formatname" region, then this is an error, and must generate an error message. This is discussed in detail in the section Section 53.9 [About Data Paragraphs and "=begin/=end" Regions], page 932.

"=for formatname text..."

This is synonymous with:

```
=begin formatname

text...

=end formatname
```

That is, it creates a region consisting of a single paragraph; that paragraph is to be treated as a normal paragraph if "formatname" begins with a ":"; if "formatname" *doesn't* begin with a colon, then "text..." will constitute a data paragraph. There is no way to use "=for formatname text..." to express "text..." as a verbatim paragraph.

"=encoding encodingname"

This command, which should occur early in the document (at least before any non-US-ASCII data!), declares that this document is encoded in the encoding *encodingname*, which must be an encoding name that `Encode` recognizes. (Encode's list of supported encodings, in `Encode-Supported`, is useful here.) If the Pod parser cannot decode the declared encoding, it should emit a warning and may abort parsing the document altogether.

A document having more than one "=encoding" line should be considered an error. Pod processors may silently tolerate this if the not-first "=encoding" lines are just duplicates of the first one (e.g., if there's a "=encoding utf8" line, and later on another "=encoding utf8" line). But Pod processors should complain if there are contradictory "=encoding" lines in the same document (e.g., if there is a "=encoding utf8" early in the document and "=encoding big5" later). Pod processors that recognize BOMs may also complain if they see an "=encoding" line that contradicts the BOM (e.g., if a document with a UTF-16LE BOM has an "=encoding shiftjis" line).

If a Pod processor sees any command other than the ones listed above (like "=head", or "=haed1", or "=stuff", or "=cuttlefish", or "=w123"), that processor must by default treat this as an error. It must not process the paragraph beginning with that command, must by default warn of this as an error, and may abort the parse. A Pod parser may allow a way for particular applications to add to the above list of known commands, and to stipulate, for each additional command, whether formatting codes should be processed.

Future versions of this specification may add additional commands.

53.5 Pod Formatting Codes

(Note that in previous drafts of this document and of perlpod, formatting codes were referred to as "interior sequences", and this term may still be found in the documentation for Pod parsers, and in error messages from Pod processors.)

There are two syntaxes for formatting codes:

- A formatting code starts with a capital letter (just US-ASCII [A-Z]) followed by a "<", any number of characters, and ending with the first matching ">". Examples:

  ```
  That's what I<you> think!

  What's C<dump()> for?

   and C<unlink()> Under Different Operating Systems>
  ```

- A formatting code starts with a capital letter (just US-ASCII [A-Z]) followed by two or more "<"'s, one or more whitespace characters, any number of characters, one or more whitespace characters, and ending with the first matching sequence of two or more ">"'s, where the number of ">"'s equals the number of "<"'s in the opening of this formatting code. Examples:

  ```
  That's what I<< you >> think!

  C<<< open(X, ">>thing.dat") || die $! >>>

  B<< $foo->bar(); >>
  ```

  With this syntax, the whitespace character(s) after the "C<<<" and before the ">>>" (or whatever letter) are *not* renderable. They do not signify whitespace, are merely part of the formatting codes themselves. That is, these are all synonymous:

  ```
  C<thing>
  C<< thing >>
  ```

```
C<<           thing     >>
C<<<   thing >>>
C<<<<
thing
           >>>>
```

and so on.

Finally, the multiple-angle-bracket form does *not* alter the interpretation of nested formatting codes, meaning that the following four example lines are identical in meaning:

```
B<example: C<$a E<lt>=E<gt> $b>>

B<example: C<< $a <=> $b >>>

B<example: C<< $a E<lt>=E<gt> $b >>>

B<<< example: C<< $a E<lt>=E<gt> $b >> >>>
```

In parsing Pod, a notably tricky part is the correct parsing of (potentially nested!) formatting codes. Implementors should consult the code in the `parse_text` routine in Pod::Parser as an example of a correct implementation.

`I<text>` – italic text

See the brief discussion in Section 52.2.4 [perlpod Formatting Codes], page 904.

`B<text>` – bold text

See the brief discussion in Section 52.2.4 [perlpod Formatting Codes], page 904.

`C<code>` – code text

See the brief discussion in Section 52.2.4 [perlpod Formatting Codes], page 904.

`F<filename>` – style for filenames

See the brief discussion in Section 52.2.4 [perlpod Formatting Codes], page 904.

`X<topic name>` – an index entry

See the brief discussion in Section 52.2.4 [perlpod Formatting Codes], page 904.

This code is unusual in that most formatters completely discard this code and its content. Other formatters will render it with invisible codes that can be used in building an index of the current document.

`Z<>` – a null (zero-effect) formatting code

Discussed briefly in Section 52.2.4 [perlpod Formatting Codes], page 904.

This code is unusual is that it should have no content. That is, a processor may complain if it sees `Z<potatoes>`. Whether or not it complains, the *potatoes* text should ignored.

`L<name>` – a hyperlink

The complicated syntaxes of this code are discussed at length in Section 52.2.4 [perlpod Formatting Codes], page 904, and implementation details are discussed below, in Section 53.7 [About L<...> Codes], page 925. Parsing the contents of L<content> is tricky. Notably, the content has to be checked for whether it looks like a URL, or whether it has to be split on literal `"|"` and/or `"/"` (in the right order!), and so on, *before* E<...> codes are resolved.

`E<escape>` – a character escape

See Section 52.2.4 [perlpod Formatting Codes], page 904, and several points in Section 53.6 [Notes on Implementing Pod Processors], page 918.

`S<text>` – text contains non-breaking spaces

This formatting code is syntactically simple, but semantically complex. What it means is that each space in the printable content of this code signifies a non-breaking space.

Consider:

```
C<$x ? $y    :  $z>

S<C<$x ? $y     :  $z>>
```

Both signify the monospace (c[ode] style) text consisting of "$x", one space, "?", one space, ":", one space, "$z". The difference is that in the latter, with the S code, those spaces are not "normal" spaces, but instead are non-breaking spaces.

If a Pod processor sees any formatting code other than the ones listed above (as in "N<...>", or "Q<...>", etc.), that processor must by default treat this as an error. A Pod parser may allow a way for particular applications to add to the above list of known formatting codes; a Pod parser might even allow a way to stipulate, for each additional command, whether it requires some form of special processing, as L<...> does.

Future versions of this specification may add additional formatting codes.

Historical note: A few older Pod processors would not see a ">" as closing a "C<" code, if the ">" was immediately preceded by a "-". This was so that this:

```
C<$foo->bar>
```

would parse as equivalent to this:

```
C<$foo-E<gt>bar>
```

instead of as equivalent to a "C" formatting code containing only "$foo-", and then a "bar>" outside the "C" formatting code. This problem has since been solved by the addition of syntaxes like this:

```
C<< $foo->bar >>
```

Compliant parsers must not treat "->" as special.

Formatting codes absolutely cannot span paragraphs. If a code is opened in one paragraph, and no closing code is found by the end of that paragraph, the Pod parser must close that formatting code, and should complain (as in "Unterminated I code in the paragraph starting at line 123: 'Time objects are not...'"). So these two paragraphs:

```
I<I told you not to do this!

Don't make me say it again!>
```

...must *not* be parsed as two paragraphs in italics (with the I code starting in one paragraph and starting in another.) Instead, the first paragraph should generate a warning, but that aside, the above code must parse as if it were:

```
I<I told you not to do this!>
```

```
Don’t make me say it again!E<gt>
```

(In SGMLish jargon, all Pod commands are like block-level elements, whereas all Pod formatting codes are like inline-level elements.)

53.6 Notes on Implementing Pod Processors

The following is a long section of miscellaneous requirements and suggestions to do with Pod processing.

- Pod formatters should tolerate lines in verbatim blocks that are of any length, even if that means having to break them (possibly several times, for very long lines) to avoid text running off the side of the page. Pod formatters may warn of such line-breaking. Such warnings are particularly appropriate for lines are over 100 characters long, which are usually not intentional.
- Pod parsers must recognize *all* of the three well-known newline formats: CR, LF, and CRLF. See Section 56.1 [perlport], page 951.
- Pod parsers should accept input lines that are of any length.
- Since Perl recognizes a Unicode Byte Order Mark at the start of files as signaling that the file is Unicode encoded as in UTF-16 (whether big-endian or little-endian) or UTF-8, Pod parsers should do the same. Otherwise, the character encoding should be understood as being UTF-8 if the first highbit byte sequence in the file seems valid as a UTF-8 sequence, or otherwise as CP-1252 (earlier versions of this specification used Latin-1 instead of CP-1252).

  Future versions of this specification may specify how Pod can accept other encodings. Presumably treatment of other encodings in Pod parsing would be as in XML parsing: whatever the encoding declared by a particular Pod file, content is to be stored in memory as Unicode characters.
- The well known Unicode Byte Order Marks are as follows: if the file begins with the two literal byte values 0xFE 0xFF, this is the BOM for big-endian UTF-16. If the file begins with the two literal byte value 0xFF 0xFE, this is the BOM for little-endian UTF-16. On an ASCII platform, if the file begins with the three literal byte values 0xEF 0xBB 0xBF, this is the BOM for UTF-8. A mechanism portable to EBCDIC platforms is to:

  ```
  my $utf8_bom = "\x{FEFF}";
  utf8::encode($utf8_bom);
  ```
- A naive, but often sufficient heuristic on ASCII platforms, for testing the first highbit byte-sequence in a BOM-less file (whether in code or in Pod!), to see whether that sequence is valid as UTF-8 (RFC 2279) is to check whether that the first byte in the sequence is in the range 0xC2 - 0xFD *and* whether the next byte is in the range 0x80 - 0xBF. If so, the parser may conclude that this file is in UTF-8, and all highbit sequences in the file should be assumed to be UTF-8. Otherwise the parser should treat the file as being in CP-1252. (A better check, and which works on EBCDIC platforms as well, is to pass a copy of the sequence to `utf8` which performs a full validity check on the sequence and returns TRUE if it is valid UTF-8, FALSE otherwise. This function is always pre-loaded, is fast because it is written in C, and will only get called at most once, so you don’t need to avoid it out of performance concerns.) In the unlikely circumstance that

the first highbit sequence in a truly non-UTF-8 file happens to appear to be UTF-8, one can cater to our heuristic (as well as any more intelligent heuristic) by prefacing that line with a comment line containing a highbit sequence that is clearly *not* valid as UTF-8. A line consisting of simply "#", an e-acute, and any non-highbit byte, is sufficient to establish this file's encoding.

- Pod processors must treat a "=for [label] [content...]" paragraph as meaning the same thing as a "=begin [label]" paragraph, content, and an "=end [label]" paragraph. (The parser may conflate these two constructs, or may leave them distinct, in the expectation that the formatter will nevertheless treat them the same.)
- When rendering Pod to a format that allows comments (i.e., to nearly any format other than plaintext), a Pod formatter must insert comment text identifying its name and version number, and the name and version numbers of any modules it might be using to process the Pod. Minimal examples:

  ```
  %% POD::Pod2PS v3.14159, using POD::Parser v1.92

  <!-- Pod::HTML v3.14159, using POD::Parser v1.92 -->

  {\doccomm generated by Pod::Tree::RTF 3.14159 using Pod::Tree 1.08}

  .\" Pod::Man version 3.14159, using POD::Parser version 1.92
  ```

  Formatters may also insert additional comments, including: the release date of the Pod formatter program, the contact address for the author(s) of the formatter, the current time, the name of input file, the formatting options in effect, version of Perl used, etc.

  Formatters may also choose to note errors/warnings as comments, besides or instead of emitting them otherwise (as in messages to STDERR, or `die`ing).
- Pod parsers *may* emit warnings or error messages ("Unknown E code E<zslig>!") to STDERR (whether through printing to STDERR, or `warn`ing/`carp`ing, or `die`ing/`croak`ing), but *must* allow suppressing all such STDERR output, and instead allow an option for reporting errors/warnings in some other way, whether by triggering a callback, or noting errors in some attribute of the document object, or some similarly unobtrusive mechanism – or even by appending a "Pod Errors" section to the end of the parsed form of the document.
- In cases of exceptionally aberrant documents, Pod parsers may abort the parse. Even then, using `die`ing/`croak`ing is to be avoided; where possible, the parser library may simply close the input file and add text like "*** Formatting Aborted ***" to the end of the (partial) in-memory document.
- In paragraphs where formatting codes (like E<...>, B<...>) are understood (i.e., *not* verbatim paragraphs, but *including* ordinary paragraphs, and command paragraphs that produce renderable text, like "=head1"), literal whitespace should generally be considered "insignificant", in that one literal space has the same meaning as any (nonzero) number of literal spaces, literal newlines, and literal tabs (as long as this produces no blank lines, since those would terminate the paragraph). Pod parsers should compact literal whitespace in each processed paragraph, but may provide an option for overriding this (since some processing tasks do not require it), or may follow additional special rules (for example, specially treating period-space-space or period-newline sequences).

- Pod parsers should not, by default, try to coerce apostrophe (’) and quote (") into smart quotes (little 9’s, 66’s, 99’s, etc), nor try to turn backtick (‘) into anything else but a single backtick character (distinct from an open quote character!), nor "–" into anything but two minus signs. They *must never* do any of those things to text in C<...> formatting codes, and never *ever* to text in verbatim paragraphs.

- When rendering Pod to a format that has two kinds of hyphens (-), one that’s a non-breaking hyphen, and another that’s a breakable hyphen (as in "object-oriented", which can be split across lines as "object-", newline, "oriented"), formatters are encouraged to generally translate "-" to non-breaking hyphen, but may apply heuristics to convert some of these to breaking hyphens.

- Pod formatters should make reasonable efforts to keep words of Perl code from being broken across lines. For example, "Foo::Bar" in some formatting systems is seen as eligible for being broken across lines as "Foo::" newline "Bar" or even "Foo::-" newline "Bar". This should be avoided where possible, either by disabling all line-breaking in mid-word, or by wrapping particular words with internal punctuation in "don’t break this across lines" codes (which in some formats may not be a single code, but might be a matter of inserting non-breaking zero-width spaces between every pair of characters in a word.)

- Pod parsers should, by default, expand tabs in verbatim paragraphs as they are processed, before passing them to the formatter or other processor. Parsers may also allow an option for overriding this.

- Pod parsers should, by default, remove newlines from the end of ordinary and verbatim paragraphs before passing them to the formatter. For example, while the paragraph you’re reading now could be considered, in Pod source, to end with (and contain) the newline(s) that end it, it should be processed as ending with (and containing) the period character that ends this sentence.

- Pod parsers, when reporting errors, should make some effort to report an approximate line number ("Nested E<>’s in Paragraph #52, near line 633 of Thing/Foo.pm!"), instead of merely noting the paragraph number ("Nested E<>’s in Paragraph #52 of Thing/Foo.pm!"). Where this is problematic, the paragraph number should at least be accompanied by an excerpt from the paragraph ("Nested E<>’s in Paragraph #52 of Thing/Foo.pm, which begins ’Read/write accessor for the C<interest rate> attribute...’").

- Pod parsers, when processing a series of verbatim paragraphs one after another, should consider them to be one large verbatim paragraph that happens to contain blank lines. I.e., these two lines, which have a blank line between them:

  ```
          use Foo;

          print Foo->VERSION
  ```

  should be unified into one paragraph ("\tuse Foo;\n\n\tprint Foo->VERSION") before being passed to the formatter or other processor. Parsers may also allow an option for overriding this.

  While this might be too cumbersome to implement in event-based Pod parsers, it is straightforward for parsers that return parse trees.

- Pod formatters, where feasible, are advised to avoid splitting short verbatim paragraphs (under twelve lines, say) across pages.
- Pod parsers must treat a line with only spaces and/or tabs on it as a "blank line" such as separates paragraphs. (Some older parsers recognized only two adjacent newlines as a "blank line" but would not recognize a newline, a space, and a newline, as a blank line. This is noncompliant behavior.)
- Authors of Pod formatters/processors should make every effort to avoid writing their own Pod parser. There are already several in CPAN, with a wide range of interface styles – and one of them, Pod::Parser, comes with modern versions of Perl.
- Characters in Pod documents may be conveyed either as literals, or by number in E<n> codes, or by an equivalent mnemonic, as in E<eacute> which is exactly equivalent to E<233>. The numbers are the Latin1/Unicode values, even on EBCDIC platforms.

  When referring to characters by using a E<n> numeric code, numbers in the range 32-126 refer to those well known US-ASCII characters (also defined there by Unicode, with the same meaning), which all Pod formatters must render faithfully. Characters whose E<> numbers are in the ranges 0-31 and 127-159 should not be used (neither as literals, nor as E<number> codes), except for the literal byte-sequences for newline (ASCII 13, ASCII 13 10, or ASCII 10), and tab (ASCII 9).

  Numbers in the range 160-255 refer to Latin-1 characters (also defined there by Unicode, with the same meaning). Numbers above 255 should be understood to refer to Unicode characters.
- Be warned that some formatters cannot reliably render characters outside 32-126; and many are able to handle 32-126 and 160-255, but nothing above 255.
- Besides the well-known "E<lt>" and "E<gt>" codes for less-than and greater-than, Pod parsers must understand "E<sol>" for "/" (solidus, slash), and "E<verbar>" for "|" (vertical bar, pipe). Pod parsers should also understand "E<lchevron>" and "E<rchevron>" as legacy codes for characters 171 and 187, i.e., "left-pointing double angle quotation mark" = "left pointing guillemet" and "right-pointing double angle quotation mark" = "right pointing guillemet". (These look like little "<<" and ">>", and they are now preferably expressed with the HTML/XHTML codes "E<laquo>" and "E<raquo>".)
- Pod parsers should understand all "E<html>" codes as defined in the entity declarations in the most recent XHTML specification at `www.W3.org`. Pod parsers must understand at least the entities that define characters in the range 160-255 (Latin-1). Pod parsers, when faced with some unknown "E<*identifier*>" code, shouldn't simply replace it with nullstring (by default, at least), but may pass it through as a string consisting of the literal characters E, less-than, *identifier*, greater-than. Or Pod parsers may offer the alternative option of processing such unknown "E<*identifier*>" codes by firing an event especially for such codes, or by adding a special node-type to the in-memory document tree. Such "E<*identifier*>" may have special meaning to some processors, or some processors may choose to add them to a special error report.
- Pod parsers must also support the XHTML codes "E<quot>" for character 34 (double-quote, "), "E<amp>" for character 38 (ampersand, &), and "E<apos>" for character 39 (apostrophe, ').

- Note that in all cases of "E<whatever>", *whatever* (whether an htmlname, or a number in any base) must consist only of alphanumeric characters – that is, *whatever* must watch `m/\A\w+\z/`. So "E< 0 1 2 3 >" is invalid, because it contains spaces, which aren't alphanumeric characters. This presumably does not *need* special treatment by a Pod processor; " 0 1 2 3 " doesn't look like a number in any base, so it would presumably be looked up in the table of HTML-like names. Since there isn't (and cannot be) an HTML-like entity called " 0 1 2 3 ", this will be treated as an error. However, Pod processors may treat "E< 0 1 2 3 >" or "E<e-acute>" as *syntactically* invalid, potentially earning a different error message than the error message (or warning, or event) generated by a merely unknown (but theoretically valid) htmlname, as in "E<qacute>" [sic]. However, Pod parsers are not required to make this distinction.

- Note that E<number> *must not* be interpreted as simply "codepoint *number* in the current/native character set". It always means only "the character represented by codepoint *number* in Unicode." (This is identical to the semantics of &#*number*; in XML.)

  This will likely require many formatters to have tables mapping from treatable Unicode codepoints (such as the "\xE9" for the e-acute character) to the escape sequences or codes necessary for conveying such sequences in the target output format. A converter to *roff would, for example know that "\xE9" (whether conveyed literally, or via a E<...> sequence) is to be conveyed as "e\\*'". Similarly, a program rendering Pod in a Mac OS application window, would presumably need to know that "\xE9" maps to codepoint 142 in MacRoman encoding that (at time of writing) is native for Mac OS. Such Unicode2whatever mappings are presumably already widely available for common output formats. (Such mappings may be incomplete! Implementers are not expected to bend over backwards in an attempt to render Cherokee syllabics, Etruscan runes, Byzantine musical symbols, or any of the other weird things that Unicode can encode.) And if a Pod document uses a character not found in such a mapping, the formatter should consider it an unrenderable character.

- If, surprisingly, the implementor of a Pod formatter can't find a satisfactory pre-existing table mapping from Unicode characters to escapes in the target format (e.g., a decent table of Unicode characters to *roff escapes), it will be necessary to build such a table. If you are in this circumstance, you should begin with the characters in the range 0x00A0 - 0x00FF, which is mostly the heavily used accented characters. Then proceed (as patience permits and fastidiousness compels) through the characters that the (X)HTML standards groups judged important enough to merit mnemonics for. These are declared in the (X)HTML specifications at the www.W3.org site. At time of writing (September 2001), the most recent entity declaration files are:

  ```
  http://www.w3.org/TR/xhtml1/DTD/xhtml-lat1.ent
  http://www.w3.org/TR/xhtml1/DTD/xhtml-special.ent
  http://www.w3.org/TR/xhtml1/DTD/xhtml-symbol.ent
  ```

  Then you can progress through any remaining notable Unicode characters in the range 0x2000-0x204D (consult the character tables at www.unicode.org), and whatever else strikes your fancy. For example, in `xhtml-symbol.ent`, there is the entry:

  ```
  <!ENTITY infin    "&#8734;"> <!-- infinity, U+221E ISOtech -->
  ```

While the mapping "infin" to the character "\x{221E}" will (hopefully) have been already handled by the Pod parser, the presence of the character in this file means that it's reasonably important enough to include in a formatter's table that maps from notable Unicode characters to the codes necessary for rendering them. So for a Unicode-to-*roff mapping, for example, this would merit the entry:

```
"\x{221E}" => '\(in',
```

It is eagerly hoped that in the future, increasing numbers of formats (and formatters) will support Unicode characters directly (as (X)HTML does with `&infin;`, `&#8734;`, or `&#x221E;`), reducing the need for idiosyncratic mappings of Unicode-to-*my_escapes*.

- It is up to individual Pod formatter to display good judgement when confronted with an unrenderable character (which is distinct from an unknown E<thing> sequence that the parser couldn't resolve to anything, renderable or not). It is good practice to map Latin letters with diacritics (like "E<eacute>"/"E<233>") to the corresponding unaccented US-ASCII letters (like a simple character 101, "e"), but clearly this is often not feasible, and an unrenderable character may be represented as "?", or the like. In attempting a sane fallback (as from E<233> to "e"), Pod formatters may use the %Latin1Code_to_fallback table in `Pod-Escapes`, or `Text-Unidecode`, if available.

  For example, this Pod text:

  ```
  magic is enabled if you set C<$Currency> to 'E<euro>'.
  ```

  may be rendered as: "magic is enabled if you set `$Currency` to '*?*'" or as "magic is enabled if you set `$Currency` to '**[euro]**'", or as "magic is enabled if you set `$Currency` to '[x20AC]', etc.

  A Pod formatter may also note, in a comment or warning, a list of what unrenderable characters were encountered.

- E<...> may freely appear in any formatting code (other than in another E<...> or in an Z<>). That is, "X<The E<euro>1,000,000 Solution>" is valid, as is "L<The E<euro>1,000,000 Solution|Million::Euros>".

- Some Pod formatters output to formats that implement non-breaking spaces as an individual character (which I'll call "NBSP"), and others output to formats that implement non-breaking spaces just as spaces wrapped in a "don't break this across lines" code. Note that at the level of Pod, both sorts of codes can occur: Pod can contain a NBSP character (whether as a literal, or as a "E<160>" or "E<nbsp>" code); and Pod can contain "S<foo I<bar> baz>" codes, where "mere spaces" (character 32) in such codes are taken to represent non-breaking spaces. Pod parsers should consider supporting the optional parsing of "S<foo I<bar> baz>" as if it were "foo*NBSP*I<bar>*NBSP*baz", and, going the other way, the optional parsing of groups of words joined by NBSP's as if each group were in a S<...> code, so that formatters may use the representation that maps best to what the output format demands.

- Some processors may find that the `S<...>` code is easiest to implement by replacing each space in the parse tree under the content of the S, with an NBSP. But note: the replacement should apply *not* to spaces in *all* text, but *only* to spaces in *printable* text. (This distinction may or may not be evident in the particular tree/event model implemented by the Pod parser.) For example, consider this unusual case:

  ```
   S<L</Autoloaded Functions>>
  ```

This means that the space in the middle of the visible link text must not be broken across lines. In other words, it's the same as this:

```
L<"AutoloadedE<160>Functions"/Autoloaded Functions>
```

However, a misapplied space-to-NBSP replacement could (wrongly) produce something equivalent to this:

```
L<"AutoloadedE<160>Functions"/AutoloadedE<160>Functions>
```

...which is almost definitely not going to work as a hyperlink (assuming this formatter outputs a format supporting hypertext).

Formatters may choose to just not support the S format code, especially in cases where the output format simply has no NBSP character/code and no code for "don't break this stuff across lines".

- Besides the NBSP character discussed above, implementors are reminded of the existence of the other "special" character in Latin-1, the "soft hyphen" character, also known as "discretionary hyphen", i.e. `E<173>` = `E<0xAD>` = `E<shy>`). This character expresses an optional hyphenation point. That is, it normally renders as nothing, but may render as a "-" if a formatter breaks the word at that point. Pod formatters should, as appropriate, do one of the following: 1) render this with a code with the same meaning (e.g., "\-" in RTF), 2) pass it through in the expectation that the formatter understands this character as such, or 3) delete it.

  For example:

  ```
  sigE<shy>action
  manuE<shy>script
  JarkE<shy>ko HieE<shy>taE<shy>nieE<shy>mi
  ```

  These signal to a formatter that if it is to hyphenate "sigaction" or "manuscript", then it should be done as "sig-*[linebreak]*action" or "manu-*[linebreak]*script" (and if it doesn't hyphenate it, then the `E<shy>` doesn't show up at all). And if it is to hyphenate "Jarkko" and/or "Hietaniemi", it can do so only at the points where there is a `E<shy>` code.

  In practice, it is anticipated that this character will not be used often, but formatters should either support it, or delete it.

- If you think that you want to add a new command to Pod (like, say, a "=biblio" command), consider whether you could get the same effect with a for or begin/end sequence: "=for biblio ..." or "=begin biblio" ... "=end biblio". Pod processors that don't understand "=for biblio", etc, will simply ignore it, whereas they may complain loudly if they see "=biblio".

- Throughout this document, "Pod" has been the preferred spelling for the name of the documentation format. One may also use "POD" or "pod". For the documentation that is (typically) in the Pod format, you may use "pod", or "Pod", or "POD". Understanding these distinctions is useful; but obsessing over how to spell them, usually is not.

53.7 About L<...> Codes

As you can tell from a glance at Section 52.1 [perlpod], page 900, the L<...> code is the most complex of the Pod formatting codes. The points below will hopefully clarify what it means and how processors should deal with it.

- In parsing an L<...> code, Pod parsers must distinguish at least four attributes:

  First:

  The link-text. If there is none, this must be `undef`. (E.g., in "L<Perl Functions|perlfunc>", the link-text is "Perl Functions". In "L<Time::HiRes>" and even "L<|Time::HiRes>", there is no link text. Note that link text may contain formatting.)

  Second:

  The possibly inferred link-text; i.e., if there was no real link text, then this is the text that we'll infer in its place. (E.g., for "L<Getopt::Std>", the inferred link text is "Getopt::Std".)

  Third:

  The name or URL, or `undef` if none. (E.g., in "L<Perl Functions|perlfunc>", the name (also sometimes called the page) is "perlfunc". In "L</CAVEATS>", the name is `undef`.)

  Fourth:

  The section (AKA "item" in older perlpods), or `undef` if none. E.g., in "L<Getopt::Std/DESCRIPTION>", "DESCRIPTION" is the section. (Note that this is not the same as a manpage section like the "5" in "man 5 crontab". "Section Foo" in the Pod sense means the part of the text that's introduced by the heading or item whose text is "Foo".)

  Pod parsers may also note additional attributes including:

  Fifth:

  A flag for whether item 3 (if present) is a URL (like "http://lists.perl.org" is), in which case there should be no section attribute; a Pod name (like "perldoc" and "Getopt::Std" are); or possibly a man page name (like "crontab(5)" is).

  Sixth:

  The raw original L<...> content, before text is split on "|", "/", etc, and before E<...> codes are expanded.

  (The above were numbered only for concise reference below. It is not a requirement that these be passed as an actual list or array.)

  For example:

```
L<Foo::Bar>
  =>  undef,                          # link text
      "Foo::Bar",                     # possibly inferred link text
      "Foo::Bar",                     # name
      undef,                          # section
      ’pod’,                         # what sort of link
      "Foo::Bar"                     # original content

L<Perlport’s section on NL’s|perlport/Newlines>
  =>  "Perlport’s section on NL’s",  # link text
      "Perlport’s section on NL’s",  # possibly inferred link text
      "perlport",                    # name
      "Newlines",                    # section
      ’pod’,                         # what sort of link
      "Perlport’s section on NL’s|perlport/Newlines"
                                     # original content

L<perlport/Newlines>
  =>  undef,                         # link text
      ’"Newlines" in perlport’,      # possibly inferred link text
      "perlport",                    # name
      "Newlines",                    # section
      ’pod’,                         # what sort of link
      "perlport/Newlines"            # original content

L<crontab(5)/"DESCRIPTION">
  =>  undef,                         # link text
      ’"DESCRIPTION" in crontab(5)’, # possibly inferred link text
      "crontab(5)",                  # name
      "DESCRIPTION",                 # section
      ’man’,                         # what sort of link
      ’crontab(5)/"DESCRIPTION"’     # original content

L</Object Attributes>
  =>  undef,                         # link text
      ’"Object Attributes"’,         # possibly inferred link text
      undef,                         # name
      "Object Attributes",           # section
      ’pod’,                         # what sort of link
      "/Object Attributes"           # original content

L<http://www.perl.org/>
  =>  undef,                         # link text
      "http://www.perl.org/",        # possibly inferred link text
      "http://www.perl.org/",        # name
      undef,                         # section
      ’url’,                         # what sort of link
      "http://www.perl.org/"         # original content

L<Perl.org|http://www.perl.org/>
  =>  "Perl.org",                    # link text
      "http://www.perl.org/",        # possibly inferred link text
      "http://www.perl.org/",        # name
      undef,                         # section
      'url',                         # what sort of link
      "Perl.org|http://www.perl.org/" # original content
```

Note that you can distinguish URL-links from anything else by the fact that they match `m/\A\w+:[^:\s]\S*\z/`. So `L<http://www.perl.com>` is a URL, but `L<HTTP::Response>` isn't.

- In case of L<...> codes with no "text|" part in them, older formatters have exhibited great variation in actually displaying the link or cross reference. For example, L<crontab(5)> would render as "the `crontab(5)` manpage", or "in the `crontab(5)` manpage" or just "`crontab(5)`".

  Pod processors must now treat "text|"-less links as follows:

  ```
  L<name>         =>  L<name|name>
  L</section>     =>  L<"section"|/section>
  L<name/section> =>  L<"section" in name|name/section>
  ```

- Note that section names might contain markup. I.e., if a section starts with:

  ```
  =head2 About the C<-M> Operator
  ```

  or with:

  ```
  =item About the C<-M> Operator
  ```

  then a link to it would look like this:

  ```
  L<somedoc/About the C<-M> Operator>
  ```

  Formatters may choose to ignore the markup for purposes of resolving the link and use only the renderable characters in the section name, as in:

  ```
  <h1><a name="About_the_-M_Operator">About the <code>-M</code>
  Operator</h1>

  ...

  <a href="somedoc#About_the_-M_Operator">About the <code>-M</code>
  Operator" in somedoc</a>
  ```

- Previous versions of perlpod distinguished `L<name/"section">` links from `L<name/item>` links (and their targets). These have been merged syntactically and semantically in the current specification, and *section* can refer either to a "=head*n* Heading Content" command or to a "=item Item Content" command. This specification does not specify what behavior should be in the case of a given document having several things all seeming to produce the same *section* identifier (e.g., in HTML, several things all producing the same *anchorname* in <a name="*anchorname*">...</a> elements). Where Pod processors can control this behavior, they should use the first such anchor. That is, `L<Foo/Bar>` refers to the *first* "Bar" section in Foo.

  But for some processors/formats this cannot be easily controlled; as with the HTML example, the behavior of multiple ambiguous <a name="*anchorname*">...</a> is most easily just left up to browsers to decide.

- In a `L<text|...>` code, text may contain formatting codes for formatting or for E<...> escapes, as in:

```
L<B<ummE<234>stuff>|...>
```

For `L<...>` codes without a "name|" part, only `E<...>` and `Z<>` codes may occur. That is, authors should not use "`L<B<Foo::Bar>>`".

Note, however, that formatting codes and Z<>'s can occur in any and all parts of an L<...> (i.e., in *name*, *section*, *text*, and *url*).

Authors must not nest L<...> codes. For example, "L<The L<Foo::Bar> man page>" should be treated as an error.

- Note that Pod authors may use formatting codes inside the "text" part of "L<text|name>" (and so on for L<text|/"sec">).

  In other words, this is valid:

  ```
  Go read L<the docs on C<$.>|perlvar/"$.">
  ```

  Some output formats that do allow rendering "L<...>" codes as hypertext, might not allow the link-text to be formatted; in that case, formatters will have to just ignore that formatting.

- At time of writing, `L<name>` values are of two types: either the name of a Pod page like `L<Foo::Bar>` (which might be a real Perl module or program in an @INC / PATH directory, or a .pod file in those places); or the name of a Unix man page, like `L<crontab(5)>`. In theory, `L<chmod>` in ambiguous between a Pod page called "chmod", or the Unix man page "chmod" (in whatever man-section). However, the presence of a string in parens, as in "crontab(5)", is sufficient to signal that what is being discussed is not a Pod page, and so is presumably a Unix man page. The distinction is of no importance to many Pod processors, but some processors that render to hypertext formats may need to distinguish them in order to know how to render a given `L<foo>` code.

- Previous versions of perlpod allowed for a `L<section>` syntax (as in `L<Object Attributes>`), which was not easily distinguishable from `L<name>` syntax and for `L<"section">` which was only slightly less ambiguous. This syntax is no longer in the specification, and has been replaced by the `L</section>` syntax (where the slash was formerly optional). Pod parsers should tolerate the `L<"section">` syntax, for a while at least. The suggested heuristic for distinguishing `L<section>` from `L<name>` is that if it contains any whitespace, it's a *section*. Pod processors should warn about this being deprecated syntax.

53.8 About =over...=back Regions

"=over"..."=back" regions are used for various kinds of list-like structures. (I use the term "region" here simply as a collective term for everything from the "=over" to the matching "=back".)

- The non-zero numeric *indentlevel* in "=over *indentlevel*" ... "=back" is used for giving the formatter a clue as to how many "spaces" (ems, or roughly equivalent units) it should tab over, although many formatters will have to convert this to an absolute measurement that may not exactly match with the size of spaces (or M's) in the document's base font. Other formatters may have to completely ignore the number. The lack of any explicit *indentlevel* parameter is equivalent to an *indentlevel* value of 4.

Pod processors may complain if *indentlevel* is present but is not a positive number matching `m/\A(\d*\.)?\d+\z/`.

- Authors of Pod formatters are reminded that "=over" ... "=back" may map to several different constructs in your output format. For example, in converting Pod to (X)HTML, it can map to any of <ul>...</ul>, <ol>...</ol>, <dl>...</dl>, or <blockquote>...</blockquote>. Similarly, "=item" can map to <li> or <dt>.
- Each "=over" ... "=back" region should be one of the following:
  - An "=over" ... "=back" region containing only "=item *" commands, each followed by some number of ordinary/verbatim paragraphs, other nested "=over" ... "=back" regions, "=for..." paragraphs, and "=begin"..."=end" regions.

    (Pod processors must tolerate a bare "=item" as if it were "=item *".) Whether "*" is rendered as a literal asterisk, an "o", or as some kind of real bullet character, is left up to the Pod formatter, and may depend on the level of nesting.
  - An "=over" ... "=back" region containing only `m/\A=item\s+\d+\.?\s*\z/` paragraphs, each one (or each group of them) followed by some number of ordinary/verbatim paragraphs, other nested "=over" ... "=back" regions, "=for..." paragraphs, and/or "=begin"..."=end" codes. Note that the numbers must start at 1 in each section, and must proceed in order and without skipping numbers.

    (Pod processors must tolerate lines like "=item 1" as if they were "=item 1.", with the period.)
  - An "=over" ... "=back" region containing only "=item [text]" commands, each one (or each group of them) followed by some number of ordinary/verbatim paragraphs, other nested "=over" ... "=back" regions, or "=for..." paragraphs, and "=begin"..."=end" regions.

    The "=item [text]" paragraph should not match `m/\A=item\s+\d+\.?\s*\z/` or `m/\A=item\s+\*\s*\z/`, nor should it match just `m/\A=item\s*\z/`.
  - An "=over" ... "=back" region containing no "=item" paragraphs at all, and containing only some number of ordinary/verbatim paragraphs, and possibly also some nested "=over" ... "=back" regions, "=for..." paragraphs, and "=begin"..."=end" regions. Such an itemless "=over" ... "=back" region in Pod is equivalent in meaning to a "<blockquote>...</blockquote>" element in HTML.

  Note that with all the above cases, you can determine which type of "=over" ... "=back" you have, by examining the first (non-"=cut", non-"=pod") Pod paragraph after the "=over" command.
- Pod formatters *must* tolerate arbitrarily large amounts of text in the "=item *text...*" paragraph. In practice, most such paragraphs are short, as in:

  ```
  =item For cutting off our trade with all parts of the world
  ```

  But they may be arbitrarily long:

  ```
  =item For transporting us beyond seas to be tried for pretended
  offenses

  =item He is at this time transporting large armies of foreign
  mercenaries to complete the works of death, desolation and
  tyranny, already begun with circumstances of cruelty and perfidy
  ```

```
scarcely paralleled in the most barbarous ages, and totally
unworthy the head of a civilized nation.
```

- Pod processors should tolerate "=item *" / "=item *number*" commands with no accompanying paragraph. The middle item is an example:

  ```
  =over

  =item 1

  Pick up dry cleaning.

  =item 2

  =item 3

  Stop by the store.  Get Abba Zabas, Stoli, and cheap lawn chairs.

  =back
  ```

- No "=over" ... "=back" region can contain headings. Processors may treat such a heading as an error.
- Note that an "=over" ... "=back" region should have some content. That is, authors should not have an empty region like this:

  ```
  =over

  =back
  ```

  Pod processors seeing such a contentless "=over" ... "=back" region, may ignore it, or may report it as an error.
- Processors must tolerate an "=over" list that goes off the end of the document (i.e., which has no matching "=back"), but they may warn about such a list.
- Authors of Pod formatters should note that this construct:

  ```
  =item Neque

  =item Porro

  =item Quisquam Est

  Qui dolorem ipsum quia dolor sit amet, consectetur, adipisci
  velit, sed quia non numquam eius modi tempora incidunt ut
  labore et dolore magnam aliquam quaerat voluptatem.

  =item Ut Enim
  ```

  is semantically ambiguous, in a way that makes formatting decisions a bit difficult. On the one hand, it could be mention of an item "Neque", mention of another item "Porro", and mention of another item "Quisquam Est", with just the last one requiring the explanatory paragraph "Qui dolorem ipsum quia dolor..."; and then an item "Ut Enim". In that case, you'd want to format it like so:

```
  Neque

  Porro

  Quisquam Est
    Qui dolorem ipsum quia dolor sit amet, consectetur, adipisci
    velit, sed quia non numquam eius modi tempora incidunt ut
    labore et dolore magnam aliquam quaerat voluptatem.

  Ut Enim
```

But it could equally well be a discussion of three (related or equivalent) items, "Neque", "Porro", and "Quisquam Est", followed by a paragraph explaining them all, and then a new item "Ut Enim". In that case, you'd probably want to format it like so:

```
  Neque
  Porro
  Quisquam Est
    Qui dolorem ipsum quia dolor sit amet, consectetur, adipisci
    velit, sed quia non numquam eius modi tempora incidunt ut
    labore et dolore magnam aliquam quaerat voluptatem.

  Ut Enim
```

But (for the foreseeable future), Pod does not provide any way for Pod authors to distinguish which grouping is meant by the above "=item"-cluster structure. So formatters should format it like so:

```
  Neque

  Porro

  Quisquam Est

    Qui dolorem ipsum quia dolor sit amet, consectetur, adipisci
    velit, sed quia non numquam eius modi tempora incidunt ut
    labore et dolore magnam aliquam quaerat voluptatem.

  Ut Enim
```

That is, there should be (at least roughly) equal spacing between items as between paragraphs (although that spacing may well be less than the full height of a line of text). This leaves it to the reader to use (con)textual cues to figure out whether the "Qui dolorem ipsum..." paragraph applies to the "Quisquam Est" item or to all three items "Neque", "Porro", and "Quisquam Est". While not an ideal situation, this is preferable to providing formatting cues that may be actually contrary to the author's intent.

53.9 About Data Paragraphs and "=begin/=end" Regions

Data paragraphs are typically used for inlining non-Pod data that is to be used (typically passed through) when rendering the document to a specific format:

```
=begin rtf

\par{\pard\qr\sa4500{\i Printed\~\chdate\~\chtime}\par}

=end rtf
```

The exact same effect could, incidentally, be achieved with a single "=for" paragraph:

```
=for rtf \par{\pard\qr\sa4500{\i Printed\~\chdate\~\chtime}\par}
```

(Although that is not formally a data paragraph, it has the same meaning as one, and Pod parsers may parse it as one.)

Another example of a data paragraph:

```
=begin html

I like <em>PIE</em>!

<hr>Especially pecan pie!

=end html
```

If these were ordinary paragraphs, the Pod parser would try to expand the "E</em>" (in the first paragraph) as a formatting code, just like "E<lt>" or "E<eacute>". But since this is in a "=begin *identifier*"..."=end *identifier*" region *and* the identifier "html" doesn't begin have a ":" prefix, the contents of this region are stored as data paragraphs, instead of being processed as ordinary paragraphs (or if they began with a spaces and/or tabs, as verbatim paragraphs).

As a further example: At time of writing, no "biblio" identifier is supported, but suppose some processor were written to recognize it as a way of (say) denoting a bibliographic reference (necessarily containing formatting codes in ordinary paragraphs). The fact that "biblio" paragraphs were meant for ordinary processing would be indicated by prefacing each "biblio" identifier with a colon:

```
=begin :biblio

Wirth, Niklaus.  1976.  I<Algorithms + Data Structures =
Programs.>  Prentice-Hall, Englewood Cliffs, NJ.

=end :biblio
```

This would signal to the parser that paragraphs in this begin...end region are subject to normal handling as ordinary/verbatim paragraphs (while still tagged as meant only for processors that understand the "biblio" identifier). The same effect could be had with:

```
=for :biblio
Wirth, Niklaus.  1976.  I<Algorithms + Data Structures =
Programs.>  Prentice-Hall, Englewood Cliffs, NJ.
```

The ":" on these identifiers means simply "process this stuff normally, even though the result will be for some special target". I suggest that parser APIs report "biblio" as the target identifier, but also report that it had a ":" prefix. (And similarly, with the above "html", report "html" as the target identifier, and note the *lack* of a ":" prefix.)

Note that a "=begin *identifier*"..."=end *identifier*" region where *identifier* begins with a colon, *can* contain commands. For example:

```
=begin :biblio

Wirth's classic is available in several editions, including:

=for comment
 hm, check abebooks.com for how much used copies cost.

=over

=item

Wirth, Niklaus.  1975.  I<Algorithmen und Datenstrukturen.>
Teubner, Stuttgart.  [Yes, it's in German.]

=item

Wirth, Niklaus.  1976.  I<Algorithms + Data Structures =
Programs.>  Prentice-Hall, Englewood Cliffs, NJ.

=back

=end :biblio
```

Note, however, a "=begin *identifier*"..."=end *identifier*" region where *identifier* does *not* begin with a colon, should not directly contain "=head1" ... "=head4" commands, nor "=over", nor "=back", nor "=item". For example, this may be considered invalid:

```
=begin somedata

This is a data paragraph.

=head1 Don't do this!

This is a data paragraph too.

=end somedata
```

A Pod processor may signal that the above (specifically the "=head1" paragraph) is an error. Note, however, that the following should *not* be treated as an error:

```
=begin somedata

This is a data paragraph.
```

```
=cut

# Yup, this isn't Pod anymore.
sub excl { (rand() > .5) ? "hoo!" : "hah!" }

=pod

This is a data paragraph too.

=end somedata
```

And this too is valid:

```
=begin someformat

This is a data paragraph.

  And this is a data paragraph.

=begin someotherformat

This is a data paragraph too.

  And this is a data paragraph too.

=begin :yetanotherformat

=head2 This is a command paragraph!

This is an ordinary paragraph!

  And this is a verbatim paragraph!

=end :yetanotherformat

=end someotherformat

Another data paragraph!

=end someformat
```

The contents of the above "=begin :yetanotherformat" ... "=end :yetanotherformat" region *aren't* data paragraphs, because the immediately containing region's identifier (":yetanotherformat") begins with a colon. In practice, most regions that contain data paragraphs will contain *only* data paragraphs; however, the above nesting is syntactically valid as Pod, even if it is rare. However, the handlers for some formats, like "html", will accept only data paragraphs, not nested regions; and they may complain if they see (targeted for them) nested regions, or commands, other than "=end", "=pod", and "=cut".

Also consider this valid structure:

```
=begin :biblio

Wirth's classic is available in several editions, including:

=over

=item

Wirth, Niklaus.  1975.  I<Algorithmen und Datenstrukturen.>
Teubner, Stuttgart.  [Yes, it's in German.]

=item

Wirth, Niklaus.  1976.  I<Algorithms + Data Structures =
Programs.>  Prentice-Hall, Englewood Cliffs, NJ.

=back

Buy buy buy!

=begin html

<img src='wirth_spokesmodeling_book.png'>

<hr>

=end html

Now now now!

=end :biblio
```

There, the "=begin html"..."=end html" region is nested inside the larger "=begin :biblio"..."=end :biblio" region. Note that the content of the "=begin html"..."=end html" region is data paragraph(s), because the immediately containing region's identifier ("html") *doesn't* begin with a colon.

Pod parsers, when processing a series of data paragraphs one after another (within a single region), should consider them to be one large data paragraph that happens to contain blank lines. So the content of the above "=begin html"..."=end html" *may* be stored as two data paragraphs (one consisting of "<img src='wirth_spokesmodeling_book.png'>\n" and another consisting of "<hr>\n"), but *should* be stored as a single data paragraph (consisting of "<img src='wirth_spokesmodeling_book.png'>\n\n<hr>\n").

Pod processors should tolerate empty "=begin *something*"..."=end *something*" regions, empty "=begin :*something*"..."=end :*something*" regions, and contentless "=for *something*" and "=for :*something*" paragraphs. I.e., these should be tolerated:

```
=for html
```

```
=begin html

=end html

=begin :biblio

=end :biblio
```

Incidentally, note that there's no easy way to express a data paragraph starting with something that looks like a command. Consider:

```
=begin stuff

=shazbot

=end stuff
```

There, "=shazbot" will be parsed as a Pod command "shazbot", not as a data paragraph "=shazbot\n". However, you can express a data paragraph consisting of "=shazbot\n" using this code:

```
=for stuff =shazbot
```

The situation where this is necessary, is presumably quite rare.

Note that =end commands must match the currently open =begin command. That is, they must properly nest. For example, this is valid:

```
=begin outer

X

=begin inner

Y

=end inner

Z

=end outer
```

while this is invalid:

```
=begin outer

X

=begin inner

Y

=end outer
```

```
Z

=end inner
```

This latter is improper because when the "=end outer" command is seen, the currently open region has the formatname "inner", not "outer". (It just happens that "outer" is the format name of a higher-up region.) This is an error. Processors must by default report this as an error, and may halt processing the document containing that error. A corollary of this is that regions cannot "overlap". That is, the latter block above does not represent a region called "outer" which contains X and Y, overlapping a region called "inner" which contains Y and Z. But because it is invalid (as all apparently overlapping regions would be), it doesn't represent that, or anything at all.

Similarly, this is invalid:

```
=begin thing

=end hting
```

This is an error because the region is opened by "thing", and the "=end" tries to close "hting" [sic].

This is also invalid:

```
=begin thing

=end
```

This is invalid because every "=end" command must have a formatname parameter.

53.10 SEE ALSO

Section 52.1 [perlpod NAME], page 900, Section 74.2.14 [perlsyn PODs: Embedded Documentation], page 1261, `podchecker`

53.11 AUTHOR

Sean M. Burke

54 perlpodstyle

54.1 NAME

perlpodstyle - Perl POD style guide

54.2 DESCRIPTION

These are general guidelines for how to write POD documentation for Perl scripts and modules, based on general guidelines for writing good UNIX man pages. All of these guidelines are, of course, optional, but following them will make your documentation more consistent with other documentation on the system.

The name of the program being documented is conventionally written in bold (using B<>) wherever it occurs, as are all program options. Arguments should be written in italics (I<>). Function names are traditionally written in italics; if you write a function as function(), Pod::Man will take care of this for you. Literal code or commands should be in C<>. References to other man pages should be in the form `manpage(section)` or `L<manpage(section)>`, and Pod::Man will automatically format those appropriately. The second form, with L<>, is used to request that a POD formatter make a link to the man page if possible. As an exception, one normally omits the section when referring to module documentation since it's not clear what section module documentation will be in; use `L<Module::Name>` for module references instead.

References to other programs or functions are normally in the form of man page references so that cross-referencing tools can provide the user with links and the like. It's possible to overdo this, though, so be careful not to clutter your documentation with too much markup. References to other programs that are not given as man page references should be enclosed in B<>.

The major headers should be set out using a `=head1` directive, and are historically written in the rather startling ALL UPPER CASE format; this is not mandatory, but it's strongly recommended so that sections have consistent naming across different software packages. Minor headers may be included using `=head2`, and are typically in mixed case.

The standard sections of a manual page are:

NAME

Mandatory section; should be a comma-separated list of programs or functions documented by this POD page, such as:

```
foo, bar - programs to do something
```

Manual page indexers are often extremely picky about the format of this section, so don't put anything in it except this line. Every program or function documented by this POD page should be listed, separated by a comma and a space. For a Perl module, just give the module name. A single dash, and only a single dash, should separate the list of programs or functions from the description. Do not use any markup such as C<> or B<> anywhere in this line. Functions should not be qualified with `()` or the like. The description should ideally fit on a single line, even if a man program replaces the dash with a few tabs.

SYNOPSIS

A short usage summary for programs and functions. This section is mandatory for section 3 pages. For Perl module documentation, it's usually convenient to have the contents of this section be a verbatim block showing some (brief) examples of typical ways the module is used.

DESCRIPTION

Extended description and discussion of the program or functions, or the body of the documentation for man pages that document something else. If particularly long, it's a good idea to break this up into subsections `=head2` directives like:

```
    =head2 Normal Usage

    =head2 Advanced Features

    =head2 Writing Configuration Files
```

or whatever is appropriate for your documentation.

For a module, this is generally where the documentation of the interfaces provided by the module goes, usually in the form of a list with an `=item` for each interface. Depending on how many interfaces there are, you may want to put that documentation in separate METHODS, FUNCTIONS, CLASS METHODS, or INSTANCE METHODS sections instead and save the DESCRIPTION section for an overview.

OPTIONS

Detailed description of each of the command-line options taken by the program. This should be separate from the description for the use of parsers like `Pod-Usage`. This is normally presented as a list, with each option as a separate `=item`. The specific option string should be enclosed in B<>. Any values that the option takes should be enclosed in I<>. For example, the section for the option –**section**=*manext* would be introduced with:

```
    =item B<--section>=I<manext>
```

Synonymous options (like both the short and long forms) are separated by a comma and a space on the same `=item` line, or optionally listed as their own item with a reference to the canonical name. For example, since –**section** can also be written as **-s**, the above would be:

```
    =item B<-s> I<manext>, B<--section>=I<manext>
```

Writing the short option first is recommended because it's easier to read. The long option is long enough to draw the eye to it anyway and the short option can otherwise get lost in visual noise.

RETURN VALUE

What the program or function returns, if successful. This section can be omitted for programs whose precise exit codes aren't important, provided they return 0 on success and non-zero on failure as is standard. It should always be present for functions. For modules, it may be useful to summarize return values from the module interface here, or it may be more useful to discuss return values separately in the documentation of each function or method the module provides.

ERRORS

Exceptions, error return codes, exit statuses, and errno settings. Typically used for function or module documentation; program documentation uses DIAGNOSTICS instead. The general rule of thumb is that errors printed to `STDOUT` or `STDERR` and intended for the end user are documented in DIAGNOSTICS while errors passed internal to the calling program and intended for other programmers are documented in ERRORS. When documenting a function that sets errno, a full list of the possible errno values should be given here.

DIAGNOSTICS

All possible messages the program can print out and what they mean. You may wish to follow the same documentation style as the Perl documentation; see perldiag(1) for more details (and look at the POD source as well).

If applicable, please include details on what the user should do to correct the error; documenting an error as indicating "the input buffer is too small" without telling the user how to increase the size of the input buffer (or at least telling them that it isn't possible) aren't very useful.

EXAMPLES

Give some example uses of the program or function. Don't skimp; users often find this the most useful part of the documentation. The examples are generally given as verbatim paragraphs.

Don't just present an example without explaining what it does. Adding a short paragraph saying what the example will do can increase the value of the example immensely.

ENVIRONMENT

Environment variables that the program cares about, normally presented as a list using `=over`, `=item`, and `=back`. For example:

```
=over 6

=item HOME

Used to determine the user's home directory.  F<.foorc> in this
directory is read for configuration details, if it exists.

=back
```

Since environment variables are normally in all uppercase, no additional special formatting is generally needed; they're glaring enough as it is.

FILES

All files used by the program or function, normally presented as a list, and what it uses them for. File names should be enclosed in F<>. It's particularly important to document files that will be potentially modified.

CAVEATS

Things to take special care with, sometimes called WARNINGS.

BUGS

Things that are broken or just don't work quite right.

RESTRICTIONS

Bugs you don't plan to fix. :-)

NOTES

Miscellaneous commentary.

AUTHOR

Who wrote it (use AUTHORS for multiple people). It's a good idea to include your current e-mail address (or some e-mail address to which bug reports should be sent) or some other contact information so that users have a way of contacting you. Remember that program documentation tends to roam the wild for far longer than you expect and pick a contact method that's likely to last.

HISTORY

Programs derived from other sources sometimes have this. Some people keep a modification log here, but that usually gets long and is normally better maintained in a separate file.

COPYRIGHT AND LICENSE

For copyright

```
Copyright YEAR(s) YOUR NAME(s)
```

(No, (C) is not needed. No, "all rights reserved" is not needed.)

For licensing the easiest way is to use the same licensing as Perl itself:

```
This library is free software; you may redistribute it and/or
modify it under the same terms as Perl itself.
```

This makes it easy for people to use your module with Perl. Note that this licensing example is neither an endorsement or a requirement, you are of course free to choose any licensing.

SEE ALSO

Other man pages to check out, like man(1), man(7), makewhatis(8), or catman(8). Normally a simple list of man pages separated by commas, or a paragraph giving the name of a reference work. Man page references, if they use the standard `name(section)` form, don't have to be enclosed in L<> (although it's recommended), but other things in this section probably should be when appropriate.

If the package has a mailing list, include a URL or subscription instructions here.

If the package has a web site, include a URL here.

Documentation of object-oriented libraries or modules may want to use CONSTRUCTORS and METHODS sections, or CLASS METHODS and INSTANCE METHODS sections, for detailed documentation of the parts of the library and save the DESCRIPTION section for an overview. Large modules with a function interface may want to use FUNCTIONS for similar reasons. Some people use OVERVIEW to summarize the description if it's quite long.

Section ordering varies, although NAME must always be the first section (you'll break some man page systems otherwise), and NAME, SYNOPSIS, DESCRIPTION, and OPTIONS generally always occur first and in that order if present. In general, SEE ALSO, AUTHOR, and similar material should be left for last. Some systems also move WARNINGS and NOTES to last. The order given above should be reasonable for most purposes.

Some systems use CONFORMING TO to note conformance to relevant standards and MT-LEVEL to note safeness for use in threaded programs or signal handlers. These headings are primarily useful when documenting parts of a C library.

Finally, as a general note, try not to use an excessive amount of markup. As documented here and in `Pod-Man`, you can safely leave Perl variables, function names, man page references, and the like unadorned by markup and the POD translators will figure it out for you. This makes it much easier to later edit the documentation. Note that many existing translators will do the wrong thing with e-mail addresses when wrapped in L<>, so don't do that.

54.3 SEE ALSO

For additional information that may be more accurate for your specific system, see either man(5) or man(7) depending on your system manual section numbering conventions.

This documentation is maintained as part of the podlators distribution. The current version is always available from its web site at <http://www.eyrie.org/~eagle/software/podlators/>.

54.4 AUTHOR

Russ Allbery <rra@stanford.edu>, with large portions of this documentation taken from the documentation of the original **pod2man** implementation by Larry Wall and Tom Christiansen.

54.5 COPYRIGHT AND LICENSE

Copyright 1999, 2000, 2001, 2004, 2006, 2008, 2010 Russ Allbery <rra@stanford.edu>.

This documentation is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

55 perlpolicy

55.1 NAME

perlpolicy - Various and sundry policies and commitments related to the Perl core

55.2 DESCRIPTION

This document is the master document which records all written policies about how the Perl 5 Porters collectively develop and maintain the Perl core.

55.3 GOVERNANCE

55.3.1 Perl 5 Porters

Subscribers to perl5-porters (the porters themselves) come in several flavours. Some are quiet curious lurkers, who rarely pitch in and instead watch the ongoing development to ensure they're forewarned of new changes or features in Perl. Some are representatives of vendors, who are there to make sure that Perl continues to compile and work on their platforms. Some patch any reported bug that they know how to fix, some are actively patching their pet area (threads, Win32, the regexp -engine), while others seem to do nothing but complain. In other words, it's your usual mix of technical people.

Over this group of porters presides Larry Wall. He has the final word in what does and does not change in any of the Perl programming languages. These days, Larry spends most of his time on Perl 6, while Perl 5 is shepherded by a "pumpking", a porter responsible for deciding what goes into each release and ensuring that releases happen on a regular basis.

Larry sees Perl development along the lines of the US government: there's the Legislature (the porters), the Executive branch (the -pumpking), and the Supreme Court (Larry). The legislature can discuss and submit patches to the executive branch all they like, but the executive branch is free to veto them. Rarely, the Supreme Court will side with the executive branch over the legislature, or the legislature over the executive branch. Mostly, however, the legislature and the executive branch are supposed to get along and work out their differences without impeachment or court cases.

You might sometimes see reference to Rule 1 and Rule 2. Larry's power as Supreme Court is expressed in The Rules:

1. Larry is always by definition right about how Perl should behave. This means he has final veto power on the core functionality.
2. Larry is allowed to change his mind about any matter at a later date, regardless of whether he previously invoked Rule 1.

Got that? Larry is always right, even when he was wrong. It's rare to see either Rule exercised, but they are often alluded to.

55.4 MAINTENANCE AND SUPPORT

Perl 5 is developed by a community, not a corporate entity. Every change contributed to the Perl core is the result of a donation. Typically, these donations are contributions of code or time by individual members of our community. On occasion, these donations come in the form of corporate or organizational sponsorship of a particular individual or project.

As a volunteer organization, the commitments we make are heavily dependent on the goodwill and hard work of individuals who have no obligation to contribute to Perl.

That being said, we value Perl's stability and security and have long had an unwritten covenant with the broader Perl community to support and maintain releases of Perl.

This document codifies the support and maintenance commitments that the Perl community should expect from Perl's developers:

- We "officially" support the two most recent stable release series. 5.16.x and earlier are now out of support. As of the release of 5.22.0, we will "officially" end support for Perl 5.18.x, other than providing security updates as described below.
- To the best of our ability, we will attempt to fix critical issues in the two most recent stable 5.x release series. Fixes for the current release series take precedence over fixes for the previous release series.
- To the best of our ability, we will provide "critical" security patches / releases for any major version of Perl whose 5.x.0 release was within the past three years. We can only commit to providing these for the most recent .y release in any 5.x.y series.
- We will not provide security updates or bug fixes for development releases of Perl.
- We encourage vendors to ship the most recent supported release of Perl at the time of their code freeze.
- As a vendor, you may have a requirement to backport security fixes beyond our 3 year support commitment. We can provide limited support and advice to you as you do so and, where possible will try to apply those patches to the relevant -maint branches in git, though we may or may not choose to make numbered releases or "official" patches available. Contact us at <perl5-security-report@perl.org> to begin that process.

55.5 BACKWARD COMPATIBILITY AND DEPRECATION

Our community has a long-held belief that backward-compatibility is a virtue, even when the functionality in question is a design flaw.

We would all love to unmake some mistakes we've made over the past decades. Living with every design error we've ever made can lead to painful stagnation. Unwinding our mistakes is very, very difficult. Doing so without actively harming our users is nearly impossible.

Lately, ignoring or actively opposing compatibility with earlier versions of Perl has come into vogue. Sometimes, a change is proposed which wants to usurp syntax which previously had another meaning. Sometimes, a change wants to improve previously-crazy semantics.

Down this road lies madness.

Requiring end-user programmers to change just a few language constructs, even language constructs which no well-educated developer would ever intentionally use is tantamount to saying "you should not upgrade to a new release of Perl unless you have 100% test coverage and can do a full manual audit of your codebase." If we were to have tools capable of reliably upgrading Perl source code from one version of Perl to another, this concern could be significantly mitigated.

We want to ensure that Perl continues to grow and flourish in the coming years and decades, but not at the expense of our user community.

Existing syntax and semantics should only be marked for destruction in very limited circumstances. If they are believed to be very rarely used, stand in the way of actual improvement to the Perl language or perl interpreter, and if affected code can be easily updated to continue working, they may be considered for removal. When in doubt, caution dictates that we will favor backward compatibility. When a feature is deprecated, a statement of reasoning describing the decision process will be posted, and a link to it will be provided in the relevant perldelta documents.

Using a lexical pragma to enable or disable legacy behavior should be considered when appropriate, and in the absence of any pragma legacy behavior should be enabled. Which backward-incompatible changes are controlled implicitly by a 'use v5.x.y' is a decision which should be made by the pumpking in consultation with the community.

Historically, we've held ourselves to a far higher standard than backward-compatibility – bugward-compatibility. Any accident of implementation or unintentional side-effect of running some bit of code has been considered to be a feature of the language to be defended with the same zeal as any other feature or functionality. No matter how frustrating these unintentional features may be to us as we continue to improve Perl, these unintentional features often deserve our protection. It is very important that existing software written in Perl continue to work correctly. If end-user developers have adopted a bug as a feature, we need to treat it as such.

New syntax and semantics which don't break existing language constructs and syntax have a much lower bar. They merely need to prove themselves to be useful, elegant, well designed, and well tested. In most cases, these additions will be marked as *experimental* for some time. See below for more on that.

55.5.1 Terminology

To make sure we're talking about the same thing when we discuss the removal of features or functionality from the Perl core, we have specific definitions for a few words and phrases.

experimental

> If something in the Perl core is marked as **experimental**, we may change its behaviour, deprecate or remove it without notice. While we'll always do our best to smooth the transition path for users of experimental features, you should contact the perl5-porters mailinglist if you find an experimental feature useful and want to help shape its future.
>
> Experimental features must be experimental in two stable releases before being marked non-experimental. Experimental features will only have their experimental status revoked when they no longer have any design-changing bugs open against them and when they have remained unchanged in behavior for the entire length of a development cycle. In other words, a feature present in v5.20.0 may be marked no longer experimental in v5.22.0 if and only if its behavior is unchanged throughout all of v5.21.

deprecated

> If something in the Perl core is marked as **deprecated**, we may remove it from the core in the future, though we might not. Generally, backward incompati-

ble changes will have deprecation warnings for two release cycles before being removed, but may be removed after just one cycle if the risk seems quite low or the benefits quite high.

As of Perl 5.12, deprecated features and modules warn the user as they're used. When a module is deprecated, it will also be made available on CPAN. Installing it from CPAN will silence deprecation warnings for that module.

If you use a deprecated feature or module and believe that its removal from the Perl core would be a mistake, please contact the perl5-porters mailinglist and plead your case. We don't deprecate things without a good reason, but sometimes there's a counterargument we haven't considered. Historically, we did not distinguish between "deprecated" and "discouraged" features.

discouraged

From time to time, we may mark language constructs and features which we consider to have been mistakes as **discouraged**. Discouraged features aren't currently candidates for removal, but we may later deprecate them if they're found to stand in the way of a significant improvement to the Perl core.

removed

Once a feature, construct or module has been marked as deprecated, we may remove it from the Perl core. Unsurprisingly, we say we've **removed** these things. When a module is removed, it will no longer ship with Perl, but will continue to be available on CPAN.

55.6 MAINTENANCE BRANCHES

New releases of maintenance branches should only contain changes that fall into one of the "acceptable" categories set out below, but must not contain any changes that fall into one of the "unacceptable" categories. (For example, a fix for a crashing bug must not be included if it breaks binary compatibility.)

It is not necessary to include every change meeting these criteria, and in general the focus should be on addressing security issues, crashing bugs, regressions and serious installation issues. The temptation to include a plethora of minor changes that don't affect the installation or execution of perl (e.g. spelling corrections in documentation) should be resisted in order to reduce the overall risk of overlooking something. The intention is to create maintenance releases which are both worthwhile and which users can have full confidence in the stability of. (A secondary concern is to avoid burning out the maint-pumpking or overwhelming other committers voting on changes to be included (see Section 55.6.1 [Getting changes into a maint branch], page 947 below).)

The following types of change may be considered acceptable, as long as they do not also fall into any of the "unacceptable" categories set out below:

- Patches that fix CVEs or security issues. These changes should be run through the perl5-security-report@perl.org mailing list rather than applied directly.
- Patches that fix crashing bugs, assertion failures and memory corruption but which do not otherwise change perl's functionality or negatively impact performance.

- Patches that fix regressions in perl's behavior relative to previous releases, no matter how old the regression, since some people may upgrade from very old versions of perl to the latest version.
- Patches that fix anything which prevents or seriously impacts the build or installation of perl.
- Portability fixes, such as changes to Configure and the files in the hints/ folder.
- Minimal patches that fix platform-specific test failures.
- Documentation updates that correct factual errors, explain significant bugs or deficiencies in the current implementation, or fix broken markup.
- Updates to dual-life modules should consist of minimal patches to fix crashing bugs or security issues (as above). Any changes made to dual-life modules for which CPAN is canonical should be coordinated with the upstream author.

The following types of change are NOT acceptable:

- Patches that break binary compatibility. (Please talk to a pumpking.)
- Patches that add or remove features.
- Patches that add new warnings or errors or deprecate features.
- Ports of Perl to a new platform, architecture or OS release that involve changes to the implementation.
- New versions of dual-life modules should NOT be imported into maint. Those belong in the next stable series.

If there is any question about whether a given patch might merit inclusion in a maint release, then it almost certainly should not be included.

55.6.1 Getting changes into a maint branch

Historically, only the pumpking cherry-picked changes from bleadperl into maintperl. This has scaling problems. At the same time, maintenance branches of stable versions of Perl need to be treated with great care. To that end, as of Perl 5.12, we have a new process for maint branches.

Any committer may cherry-pick any commit from blead to a maint branch if they send mail to perl5-porters announcing their intent to cherry-pick a specific commit along with a rationale for doing so and at least two other committers respond to the list giving their assent. (This policy applies to current and former pumpkings, as well as other committers.)

Other voting mechanisms may be used instead, as long as the same number of votes is gathered in a transparent manner. Specifically, proposals of which changes to cherry-pick must be visible to everyone on perl5-porters so that the views of everyone interested may be heard.

It is not necessary for voting to be held on cherry-picking perldelta entries associated with changes that have already been cherry-picked, nor for the maint-pumpking to obtain votes on changes required by the `Porting/release_managers_guide.pod` where such changes can be applied by the means of cherry-picking from blead.

55.7 CONTRIBUTED MODULES

55.7.1 A Social Contract about Artistic Control

What follows is a statement about artistic control, defined as the ability of authors of packages to guide the future of their code and maintain control over their work. It is a recognition that authors should have control over their work, and that it is a responsibility of the rest of the Perl community to ensure that they retain this control. It is an attempt to document the standards to which we, as Perl developers, intend to hold ourselves. It is an attempt to write down rough guidelines about the respect we owe each other as Perl developers.

This statement is not a legal contract. This statement is not a legal document in any way, shape, or form. Perl is distributed under the GNU Public License and under the Artistic License; those are the precise legal terms. This statement isn't about the law or licenses. It's about community, mutual respect, trust, and good-faith cooperation.

We recognize that the Perl core, defined as the software distributed with the heart of Perl itself, is a joint project on the part of all of us. From time to time, a script, module, or set of modules (hereafter referred to simply as a "module") will prove so widely useful and/or so integral to the correct functioning of Perl itself that it should be distributed with the Perl core. This should never be done without the author's explicit consent, and a clear recognition on all parts that this means the module is being distributed under the same terms as Perl itself. A module author should realize that inclusion of a module into the Perl core will necessarily mean some loss of control over it, since changes may occasionally have to be made on short notice or for consistency with the rest of Perl.

Once a module has been included in the Perl core, however, everyone involved in maintaining Perl should be aware that the module is still the property of the original author unless the original author explicitly gives up their ownership of it. In particular:

- The version of the module in the Perl core should still be considered the work of the original author. All patches, bug reports, and so forth should be fed back to them. Their development directions should be respected whenever possible.
- Patches may be applied by the pumpkin holder without the explicit cooperation of the module author if and only if they are very minor, time-critical in some fashion (such as urgent security fixes), or if the module author cannot be reached. Those patches must still be given back to the author when possible, and if the author decides on an alternate fix in their version, that fix should be strongly preferred unless there is a serious problem with it. Any changes not endorsed by the author should be marked as such, and the contributor of the change acknowledged.
- The version of the module distributed with Perl should, whenever possible, be the latest version of the module as distributed by the author (the latest non-beta version in the case of public Perl releases), although the pumpkin holder may hold off on upgrading the version of the module distributed with Perl to the latest version until the latest version has had sufficient testing.

In other words, the author of a module should be considered to have final say on modifications to their module whenever possible (bearing in mind that it's expected that everyone involved will work together and arrive at reasonable compromises when there are disagreements).

As a last resort, however:

If the author's vision of the future of their module is sufficiently different from the vision of the pumpkin holder and perl5-porters as a whole so as to cause serious problems for Perl, the pumpkin holder may choose to formally fork the version of the module in the Perl core from the one maintained by the author. This should not be done lightly and should **always** if at all possible be done only after direct input from Larry. If this is done, it must then be made explicit in the module as distributed with the Perl core that it is a forked version and that while it is based on the original author's work, it is no longer maintained by them. This must be noted in both the documentation and in the comments in the source of the module.

Again, this should be a last resort only. Ideally, this should never happen, and every possible effort at cooperation and compromise should be made before doing this. If it does prove necessary to fork a module for the overall health of Perl, proper credit must be given to the original author in perpetuity and the decision should be constantly re-evaluated to see if a remerging of the two branches is possible down the road.

In all dealings with contributed modules, everyone maintaining Perl should keep in mind that the code belongs to the original author, that they may not be on perl5-porters at any given time, and that a patch is not official unless it has been integrated into the author's copy of the module. To aid with this, and with points #1, #2, and #3 above, contact information for the authors of all contributed modules should be kept with the Perl distribution.

Finally, the Perl community as a whole recognizes that respect for ownership of code, respect for artistic control, proper credit, and active effort to prevent unintentional code skew or communication gaps is vital to the health of the community and Perl itself. Members of a community should not normally have to resort to rules and laws to deal with each other, and this document, although it contains rules so as to be clear, is about an attitude and general approach. The first step in any dispute should be open communication, respect for opposing views, and an attempt at a compromise. In nearly every circumstance nothing more will be necessary, and certainly no more drastic measure should be used until every avenue of communication and discussion has failed.

55.8 DOCUMENTATION

Perl's documentation is an important resource for our users. It's incredibly important for Perl's documentation to be reasonably coherent and to accurately reflect the current implementation.

Just as P5P collectively maintains the codebase, we collectively maintain the documentation. Writing a particular bit of documentation doesn't give an author control of the future of that documentation. At the same time, just as source code changes should match the style of their surrounding blocks, so should documentation changes.

Examples in documentation should be illustrative of the concept they're explaining. Sometimes, the best way to show how a language feature works is with a small program the reader can run without modification. More often, examples will consist of a snippet of code containing only the "important" bits. The definition of "important" varies from snippet to snippet. Sometimes it's important to declare `use strict` and `use warnings`, initialize all variables and fully catch every error condition. More often than not, though, those things obscure the lesson the example was intended to teach.

As Perl is developed by a global team of volunteers, our documentation often contains spellings which look funny to *somebody*. Choice of American/British/Other spellings is left as an exercise for the author of each bit of documentation. When patching documentation, try to emulate the documentation around you, rather than changing the existing prose.

In general, documentation should describe what Perl does "now" rather than what it used to do. It's perfectly reasonable to include notes in documentation about how behaviour has changed from previous releases, but, with very few exceptions, documentation isn't "dual-life" – it doesn't need to fully describe how all old versions used to work.

55.9 STANDARDS OF CONDUCT

The official forum for the development of perl is the perl5-porters mailing list, mentioned above, and its bugtracker at rt.perl.org. All participants in discussion there are expected to adhere to a standard of conduct.

- Always be civil.
- Heed the moderators.

Civility is simple: stick to the facts while avoiding demeaning remarks and sarcasm. It is not enough to be factual. You must also be civil. Responding in kind to incivility is not acceptable.

If the list moderators tell you that you are not being civil, carefully consider how your words have appeared before responding in any way. You may protest, but repeated protest in the face of a repeatedly reaffirmed decision is not acceptable.

Unacceptable behavior will result in a public and clearly identified warning. Repeated unacceptable behavior will result in removal from the mailing list and revocation of rights to update rt.perl.org. The first removal is for one month. Subsequent removals will double in length. After six months with no warning, a user's ban length is reset. Removals, like warnings, are public.

The list of moderators will be public knowledge. At present, it is: Aaron Crane, Andy Dougherty, Ricardo Signes, Steffen Müller.

55.10 CREDITS

"Social Contract about Contributed Modules" originally by Russ Allbery <rra@stanford.edu> and the perl5-porters.

56 perlport

56.1 NAME

perlport - Writing portable Perl

56.2 DESCRIPTION

Perl runs on numerous operating systems. While most of them share much in common, they also have their own unique features.

This document is meant to help you to find out what constitutes portable Perl code. That way once you make a decision to write portably, you know where the lines are drawn, and you can stay within them.

There is a tradeoff between taking full advantage of one particular type of computer and taking advantage of a full range of them. Naturally, as you broaden your range and become more diverse, the common factors drop, and you are left with an increasingly smaller area of common ground in which you can operate to accomplish a particular task. Thus, when you begin attacking a problem, it is important to consider under which part of the tradeoff curve you want to operate. Specifically, you must decide whether it is important that the task that you are coding has the full generality of being portable, or whether to just get the job done right now. This is the hardest choice to be made. The rest is easy, because Perl provides many choices, whichever way you want to approach your problem.

Looking at it another way, writing portable code is usually about willfully limiting your available choices. Naturally, it takes discipline and sacrifice to do that. The product of portability and convenience may be a constant. You have been warned.

Be aware of two important points:

Not all Perl programs have to be portable

> There is no reason you should not use Perl as a language to glue Unix tools together, or to prototype a Macintosh application, or to manage the Windows registry. If it makes no sense to aim for portability for one reason or another in a given program, then don't bother.

Nearly all of Perl already *is* portable

> Don't be fooled into thinking that it is hard to create portable Perl code. It isn't. Perl tries its level-best to bridge the gaps between what's available on different platforms, and all the means available to use those features. Thus almost all Perl code runs on any machine without modification. But there are some significant issues in writing portable code, and this document is entirely about those issues.

Here's the general rule: When you approach a task commonly done using a whole range of platforms, think about writing portable code. That way, you don't sacrifice much by way of the implementation choices you can avail yourself of, and at the same time you can give your users lots of platform choices. On the other hand, when you have to take advantage of some unique feature of a particular platform, as is often the case with systems programming (whether for Unix, Windows, VMS, etc.), consider writing platform-specific code.

When the code will run on only two or three operating systems, you may need to consider only the differences of those particular systems. The important thing is to decide where the code will run and to be deliberate in your decision.

The material below is separated into three main sections: main issues of portability (Section 56.3 [ISSUES], page 952), platform-specific issues (Section 56.5 [PLATFORMS], page 963), and built-in Perl functions that behave differently on various ports (Section 56.6 [FUNCTION IMPLEMENTATIONS], page 971).

This information should not be considered complete; it includes possibly transient information about idiosyncrasies of some of the ports, almost all of which are in a state of constant evolution. Thus, this material should be considered a perpetual work in progress (`<IMG SRC="yellow_sign.gif" ALT="Under Construction">`).

56.3 ISSUES

56.3.1 Newlines

In most operating systems, lines in files are terminated by newlines. Just what is used as a newline may vary from OS to OS. Unix traditionally uses `\012`, one type of DOSish I/O uses `\015\012`, Mac OS uses `\015`, and z/OS uses `\025`.

Perl uses `\n` to represent the "logical" newline, where what is logical may depend on the platform in use. In MacPerl, `\n` always means `\015`. On EBCDIC platforms, `\n` could be `\025` or `\045`. In DOSish perls, `\n` usually means `\012`, but when accessing a file in "text" mode, perl uses the `:crlf` layer that translates it to (or from) `\015\012`, depending on whether you're reading or writing. Unix does the same thing on ttys in canonical mode. `\015\012` is commonly referred to as CRLF.

To trim trailing newlines from text lines use `chomp()`. With default settings that function looks for a trailing `\n` character and thus trims in a portable way.

When dealing with binary files (or text files in binary mode) be sure to explicitly set $/ to the appropriate value for your file format before using `chomp()`.

Because of the "text" mode translation, DOSish perls have limitations in using `seek` and `tell` on a file accessed in "text" mode. Stick to `seek`-ing to locations you got from `tell` (and no others), and you are usually free to use `seek` and `tell` even in "text" mode. Using `seek` or `tell` or other file operations may be non-portable. If you use `binmode` on a file, however, you can usually `seek` and `tell` with arbitrary values safely.

A common misconception in socket programming is that `\n eq \012` everywhere. When using protocols such as common Internet protocols, `\012` and `\015` are called for specifically, and the values of the logical `\n` and `\r` (carriage return) are not reliable.

```
print SOCKET "Hi there, client!\r\n";      # WRONG
print SOCKET "Hi there, client!\015\012";  # RIGHT
```

However, using `\015\012` (or `\cM\cJ`, or `\x0D\x0A`) can be tedious and unsightly, as well as confusing to those maintaining the code. As such, the `Socket` module supplies the Right Thing for those who want it.

```
use Socket qw(:DEFAULT :crlf);
print SOCKET "Hi there, client!$CRLF"      # RIGHT
```

When reading from a socket, remember that the default input record separator $/ is \n, but robust socket code will recognize as either \012 or \015\012 as end of line:

```
while (<SOCKET>) {  # NOT ADVISABLE!
    # ...
}
```

Because both CRLF and LF end in LF, the input record separator can be set to LF and any CR stripped later. Better to write:

```
use Socket qw(:DEFAULT :crlf);
local($/) = LF;      # not needed if $/ is already \012

while (<SOCKET>) {
    s/$CR?$LF/\n/;   # not sure if socket uses LF or CRLF, OK
#   s/\015?\012/\n/; # same thing
}
```

This example is preferred over the previous one–even for Unix platforms–because now any \015's (\cM's) are stripped out (and there was much rejoicing).

Similarly, functions that return text data–such as a function that fetches a web page–should sometimes translate newlines before returning the data, if they've not yet been translated to the local newline representation. A single line of code will often suffice:

```
$data =~ s/\015?\012/\n/g;
return $data;
```

Some of this may be confusing. Here's a handy reference to the ASCII CR and LF characters. You can print it out and stick it in your wallet.

```
LF  eq  \012  eq  \x0A  eq  \cJ  eq  chr(10)  eq  ASCII 10
CR  eq  \015  eq  \x0D  eq  \cM  eq  chr(13)  eq  ASCII 13

         | Unix | DOS  | Mac  |
    ---------------------------
    \n   |  LF  |  LF  |  CR  |
    \r   |  CR  |  CR  |  LF  |
    \n * |  LF  | CRLF |  CR  |
    \r * |  CR  |  CR  |  LF  |
    ---------------------------
    * text-mode STDIO
```

The Unix column assumes that you are not accessing a serial line (like a tty) in canonical mode. If you are, then CR on input becomes "\n", and "\n" on output becomes CRLF.

These are just the most common definitions of \n and \r in Perl. There may well be others. For example, on an EBCDIC implementation such as z/OS (OS/390) or OS/400 (using the ILE, the PASE is ASCII-based) the above material is similar to "Unix" but the code numbers change:

```
LF  eq  \025  eq  \x15  eq  \cU  eq  chr(21)  eq  CP-1047 21
LF  eq  \045  eq  \x25  eq           chr(37)  eq  CP-0037 37
CR  eq  \015  eq  \x0D  eq  \cM  eq  chr(13)  eq  CP-1047 13
CR  eq  \015  eq  \x0D  eq  \cM  eq  chr(13)  eq  CP-0037 13
```

```
     | z/OS | OS/400 |
----------------------
\n   |  LF  |  LF    |
\r   |  CR  |  CR    |
\n * |  LF  |  LF    |
\r * |  CR  |  CR    |
----------------------
* text-mode STDIO
```

56.3.2 Numbers endianness and Width

Different CPUs store integers and floating point numbers in different orders (called *endianness*) and widths (32-bit and 64-bit being the most common today). This affects your programs when they attempt to transfer numbers in binary format from one CPU architecture to another, usually either "live" via network connection, or by storing the numbers to secondary storage such as a disk file or tape.

Conflicting storage orders make an utter mess out of the numbers. If a little-endian host (Intel, VAX) stores 0x12345678 (305419896 in decimal), a big-endian host (Motorola, Sparc, PA) reads it as 0x78563412 (2018915346 in decimal). Alpha and MIPS can be either: Digital/Compaq used/uses them in little-endian mode; SGI/Cray uses them in big-endian mode. To avoid this problem in network (socket) connections use the `pack` and `unpack` formats `n` and `N`, the "network" orders. These are guaranteed to be portable.

As of Perl 5.10.0, you can also use the `>` and `<` modifiers to force big- or little-endian byte-order. This is useful if you want to store signed integers or 64-bit integers, for example.

You can explore the endianness of your platform by unpacking a data structure packed in native format such as:

```
print unpack("h*", pack("s2", 1, 2)), "\n";
# '10002000' on e.g. Intel x86 or Alpha 21064 in little-endian mode
# '00100020' on e.g. Motorola 68040
```

If you need to distinguish between endian architectures you could use either of the variables set like so:

```
$is_big_endian   = unpack("h*", pack("s", 1)) =~ /01/;
$is_little_endian = unpack("h*", pack("s", 1)) =~ /^1/;
```

Differing widths can cause truncation even between platforms of equal endianness. The platform of shorter width loses the upper parts of the number. There is no good solution for this problem except to avoid transferring or storing raw binary numbers.

One can circumnavigate both these problems in two ways. Either transfer and store numbers always in text format, instead of raw binary, or else consider using modules like `Data::Dumper` and `Storable` (included as of Perl 5.8). Keeping all data as text significantly simplifies matters.

The v-strings are portable only up to v2147483647 (0x7FFF_FFFF), that's how far EBCDIC, or more precisely UTF-EBCDIC will go.

56.3.3 Files and Filesystems

Most platforms these days structure files in a hierarchical fashion. So, it is reasonably safe to assume that all platforms support the notion of a "path" to uniquely identify a file on the system. How that path is really written, though, differs considerably.

Although similar, file path specifications differ between Unix, Windows, Mac OS, OS/2, VMS, VOS, RISC OS, and probably others. Unix, for example, is one of the few OSes that has the elegant idea of a single root directory.

DOS, OS/2, VMS, VOS, and Windows can work similarly to Unix with `/` as path separator, or in their own idiosyncratic ways (such as having several root directories and various "unrooted" device files such NIL: and LPT:).

Mac OS 9 and earlier used `:` as a path separator instead of `/`.

The filesystem may support neither hard links (`link`) nor symbolic links (`symlink`, `readlink`, `lstat`).

The filesystem may support neither access timestamp nor change timestamp (meaning that about the only portable timestamp is the modification timestamp), or one second granularity of any timestamps (e.g. the FAT filesystem limits the time granularity to two seconds).

The "inode change timestamp" (the `-C` filetest) may really be the "creation timestamp" (which it is not in Unix).

VOS perl can emulate Unix filenames with `/` as path separator. The native pathname characters greater-than, less-than, number-sign, and percent-sign are always accepted.

RISC OS perl can emulate Unix filenames with `/` as path separator, or go native and use `.` for path separator and `:` to signal filesystems and disk names.

Don't assume Unix filesystem access semantics: that read, write, and execute are all the permissions there are, and even if they exist, that their semantics (for example what do "`r`", "`w`", and "`x`" mean on a directory) are the Unix ones. The various Unix/POSIX compatibility layers usually try to make interfaces like `chmod()` work, but sometimes there simply is no good mapping.

If all this is intimidating, have no (well, maybe only a little) fear. There are modules that can help. The `File::Spec` modules provide methods to do the Right Thing on whatever platform happens to be running the program.

```
use File::Spec::Functions;
chdir(updir());        # go up one directory
my $file = catfile(curdir(), 'temp', 'file.txt');
# on Unix and Win32, './temp/file.txt'
# on Mac OS Classic, ':temp:file.txt'
# on VMS, '[.temp]file.txt'
```

`File::Spec` is available in the standard distribution as of version 5.004_05. `File::Spec::Functions` is only in `File::Spec` 0.7 and later, and some versions of Perl come with version 0.6. If `File::Spec` is not updated to 0.7 or later, you must use the object-oriented interface from `File::Spec` (or upgrade `File::Spec`).

In general, production code should not have file paths hardcoded. Making them user-supplied or read from a configuration file is better, keeping in mind that file path syntax varies on different machines.

This is especially noticeable in scripts like Makefiles and test suites, which often assume / as a path separator for subdirectories.

Also of use is File::Basename from the standard distribution, which splits a pathname into pieces (base filename, full path to directory, and file suffix).

Even when on a single platform (if you can call Unix a single platform), remember not to count on the existence or the contents of particular system-specific files or directories, like /etc/passwd, /etc/sendmail.conf, /etc/resolv.conf, or even /tmp/. For example, /etc/passwd may exist but not contain the encrypted passwords, because the system is using some form of enhanced security. Or it may not contain all the accounts, because the system is using NIS. If code does need to rely on such a file, include a description of the file and its format in the code's documentation, then make it easy for the user to override the default location of the file.

Don't assume a text file will end with a newline. They should, but people forget.

Do not have two files or directories of the same name with different case, like test.pl and Test.pl, as many platforms have case-insensitive (or at least case-forgiving) filenames. Also, try not to have non-word characters (except for .) in the names, and keep them to the 8.3 convention, for maximum portability, onerous a burden though this may appear.

Likewise, when using the AutoSplit module, try to keep your functions to 8.3 naming and case-insensitive conventions; or, at the least, make it so the resulting files have a unique (case-insensitively) first 8 characters.

Whitespace in filenames is tolerated on most systems, but not all, and even on systems where it might be tolerated, some utilities might become confused by such whitespace.

Many systems (DOS, VMS ODS-2) cannot have more than one . in their filenames.

Don't assume > won't be the first character of a filename. Always use < explicitly to open a file for reading, or even better, use the three-arg version of open, unless you want the user to be able to specify a pipe open.

```
open my $fh, '<', $existing_file) or die $!;
```

If filenames might use strange characters, it is safest to open it with sysopen instead of open. open is magic and can translate characters like >, <, and |, which may be the wrong thing to do. (Sometimes, though, it's the right thing.) Three-arg open can also help protect against this translation in cases where it is undesirable.

Don't use : as a part of a filename since many systems use that for their own semantics (Mac OS Classic for separating pathname components, many networking schemes and utilities for separating the nodename and the pathname, and so on). For the same reasons, avoid @, ; and |.

Don't assume that in pathnames you can collapse two leading slashes // into one: some networking and clustering filesystems have special semantics for that. Let the operating system sort it out.

The *portable filename characters* as defined by ANSI C are

```
a b c d e f g h i j k l m n o p q r t u v w x y z
A B C D E F G H I J K L M N O P Q R T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
. _ -
```

and the "-" shouldn't be the first character. If you want to be hypercorrect, stay case-insensitive and within the 8.3 naming convention (all the files and directories have to be unique within one directory if their names are lowercased and truncated to eight characters before the ., if any, and to three characters after the ., if any). (And do not use .s in directory names.)

56.3.4 System Interaction

Not all platforms provide a command line. These are usually platforms that rely primarily on a Graphical User Interface (GUI) for user interaction. A program requiring a command line interface might not work everywhere. This is probably for the user of the program to deal with, so don't stay up late worrying about it.

Some platforms can't delete or rename files held open by the system, this limitation may also apply to changing filesystem metainformation like file permissions or owners. Remember to `close` files when you are done with them. Don't `unlink` or `rename` an open file. Don't `tie` or `open` a file already tied or opened; `untie` or `close` it first.

Don't open the same file more than once at a time for writing, as some operating systems put mandatory locks on such files.

Don't assume that write/modify permission on a directory gives the right to add or delete files/directories in that directory. That is filesystem specific: in some filesystems you need write/modify permission also (or even just) in the file/directory itself. In some filesystems (AFS, DFS) the permission to add/delete directory entries is a completely separate permission.

Don't assume that a single `unlink` completely gets rid of the file: some filesystems (most notably the ones in VMS) have versioned filesystems, and `unlink()` removes only the most recent one (it doesn't remove all the versions because by default the native tools on those platforms remove just the most recent version, too). The portable idiom to remove all the versions of a file is

```
1 while unlink "file";
```

This will terminate if the file is undeleteable for some reason (protected, not there, and so on).

Don't count on a specific environment variable existing in `%ENV`. Don't count on `%ENV` entries being case-sensitive, or even case-preserving. Don't try to clear `%ENV` by saying `%ENV = ();`, or, if you really have to, make it conditional on `$^O ne 'VMS'` since in VMS the `%ENV` table is much more than a per-process key-value string table.

On VMS, some entries in the `%ENV` hash are dynamically created when their key is used on a read if they did not previously exist. The values for `$ENV{HOME}`, `$ENV{TERM}`, `$ENV{PATH}`, and `$ENV{USER}`, are known to be dynamically generated. The specific names that are dynamically generated may vary with the version of the C library on VMS, and more may exist than are documented.

On VMS by default, changes to the %ENV hash persist after perl exits. Subsequent invocations of perl in the same process can inadvertently inherit environment settings that were meant to be temporary.

Don't count on signals or `%SIG` for anything.

Don't count on filename globbing. Use `opendir`, `readdir`, and `closedir` instead.

Don't count on per-program environment variables, or per-program current directories.

Don't count on specific values of `$!`, neither numeric nor especially the string values. Users may switch their locales causing error messages to be translated into their languages. If you can trust a POSIXish environment, you can portably use the symbols defined by the `Errno` module, like `ENOENT`. And don't trust on the values of `$!` at all except immediately after a failed system call.

56.3.5 Command names versus file pathnames

Don't assume that the name used to invoke a command or program with `system` or `exec` can also be used to test for the existence of the file that holds the executable code for that command or program. First, many systems have "internal" commands that are built-in to the shell or OS and while these commands can be invoked, there is no corresponding file. Second, some operating systems (e.g., Cygwin, DJGPP, OS/2, and VOS) have required suffixes for executable files; these suffixes are generally permitted on the command name but are not required. Thus, a command like `"perl"` might exist in a file named `"perl"`, `"perl.exe"`, or `"perl.pm"`, depending on the operating system. The variable `"_exe"` in the `Config` module holds the executable suffix, if any. Third, the VMS port carefully sets up `$^X` and `$Config{perlpath}` so that no further processing is required. This is just as well, because the matching regular expression used below would then have to deal with a possible trailing version number in the VMS file name.

To convert `$^X` to a file pathname, taking account of the requirements of the various operating system possibilities, say:

```
use Config;
my $thisperl = $^X;
if ($^O ne 'VMS')
   {$thisperl .= $Config{_exe} unless $thisperl =~ m/$Config{_exe}$/i;}
```

To convert `$Config{perlpath}` to a file pathname, say:

```
use Config;
my $thisperl = $Config{perlpath};
if ($^O ne 'VMS')
   {$thisperl .= $Config{_exe} unless $thisperl =~ m/$Config{_exe}$/i;}
```

56.3.6 Networking

Don't assume that you can reach the public Internet.

Don't assume that there is only one way to get through firewalls to the public Internet.

Don't assume that you can reach outside world through any other port than 80, or some web proxy. ftp is blocked by many firewalls.

Don't assume that you can send email by connecting to the local SMTP port.

Don't assume that you can reach yourself or any node by the name 'localhost'. The same goes for '127.0.0.1'. You will have to try both.

Don't assume that the host has only one network card, or that it can't bind to many virtual IP addresses.

Don't assume a particular network device name.

Don't assume a particular set of `ioctl()`s will work.

Don't assume that you can ping hosts and get replies.

Don't assume that any particular port (service) will respond.

Don't assume that Sys::Hostname (or any other API or command) returns either a fully qualified hostname or a non-qualified hostname: it all depends on how the system had been configured. Also remember that for things such as DHCP and NAT, the hostname you get back might not be very useful.

All the above "don't":s may look daunting, and they are, but the key is to degrade gracefully if one cannot reach the particular network service one wants. Croaking or hanging do not look very professional.

56.3.7 Interprocess Communication (IPC)

In general, don't directly access the system in code meant to be portable. That means, no system, exec, fork, pipe, ``, qx//, open with a |, nor any of the other things that makes being a Perl hacker worth being.

Commands that launch external processes are generally supported on most platforms (though many of them do not support any type of forking). The problem with using them arises from what you invoke them on. External tools are often named differently on different platforms, may not be available in the same location, might accept different arguments, can behave differently, and often present their results in a platform-dependent way. Thus, you should seldom depend on them to produce consistent results. (Then again, if you're calling *netstat -a*, you probably don't expect it to run on both Unix and CP/M.)

One especially common bit of Perl code is opening a pipe to **sendmail**:

```
open(MAIL, '|/usr/lib/sendmail -t')
    or die "cannot fork sendmail: $!";
```

This is fine for systems programming when sendmail is known to be available. But it is not fine for many non-Unix systems, and even some Unix systems that may not have sendmail installed. If a portable solution is needed, see the various distributions on CPAN that deal with it. Mail::Mailer and Mail::Send in the MailTools distribution are commonly used, and provide several mailing methods, including mail, sendmail, and direct SMTP (via Net::SMTP) if a mail transfer agent is not available. Mail::Sendmail is a standalone module that provides simple, platform-independent mailing.

The Unix System V IPC (msg*(), sem*(), shm*()) is not available even on all Unix platforms.

Do not use either the bare result of pack("N", 10, 20, 30, 40) or bare v-strings (such as v10.20.30.40) to represent IPv4 addresses: both forms just pack the four bytes into network order. That this would be equal to the C language in_addr struct (which is what the socket code internally uses) is not guaranteed. To be portable use the routines of the Socket extension, such as inet_aton(), inet_ntoa(), and sockaddr_in().

The rule of thumb for portable code is: Do it all in portable Perl, or use a module (that may internally implement it with platform-specific code, but exposes a common interface).

56.3.8 External Subroutines (XS)

XS code can usually be made to work with any platform, but dependent libraries, header files, etc., might not be readily available or portable, or the XS code itself might be platform-specific, just as Perl code might be. If the libraries and headers are portable, then it is normally reasonable to make sure the XS code is portable, too.

A different type of portability issue arises when writing XS code: availability of a C compiler on the end-user's system. C brings with it its own portability issues, and writing XS code will expose you to some of those. Writing purely in Perl is an easier way to achieve portability.

56.3.9 Standard Modules

In general, the standard modules work across platforms. Notable exceptions are the `CPAN` module (which currently makes connections to external programs that may not be available), platform-specific modules (like `ExtUtils::MM_VMS`), and DBM modules.

There is no one DBM module available on all platforms. `SDBM_File` and the others are generally available on all Unix and DOSish ports, but not in MacPerl, where only `NDBM_File` and `DB_File` are available.

The good news is that at least some DBM module should be available, and `AnyDBM_File` will use whichever module it can find. Of course, then the code needs to be fairly strict, dropping to the greatest common factor (e.g., not exceeding 1K for each record), so that it will work with any DBM module. See `AnyDBM_File` for more details.

56.3.10 Time and Date

The system's notion of time of day and calendar date is controlled in widely different ways. Don't assume the timezone is stored in `$ENV{TZ}`, and even if it is, don't assume that you can control the timezone through that variable. Don't assume anything about the three-letter timezone abbreviations (for example that MST would be the Mountain Standard Time, it's been known to stand for Moscow Standard Time). If you need to use timezones, express them in some unambiguous format like the exact number of minutes offset from UTC, or the POSIX timezone format.

Don't assume that the epoch starts at 00:00:00, January 1, 1970, because that is OS- and implementation-specific. It is better to store a date in an unambiguous representation. The ISO 8601 standard defines YYYY-MM-DD as the date format, or YYYY-MM-DDTHH:MM:SS (that's a literal `"T"` separating the date from the time). Please do use the ISO 8601 instead of making us guess what date 02/03/04 might be. ISO 8601 even sorts nicely as-is. A text representation (like `"1987-12-18"`) can be easily converted into an OS-specific value using a module like `Date::Parse`. An array of values, such as those returned by `localtime`, can be converted to an OS-specific representation using `Time::Local`.

When calculating specific times, such as for tests in time or date modules, it may be appropriate to calculate an offset for the epoch.

```
require Time::Local;
my $offset = Time::Local::timegm(0, 0, 0, 1, 0, 70);
```

The value for `$offset` in Unix will be `0`, but in Mac OS Classic will be some large number. `$offset` can then be added to a Unix time value to get what should be the proper value on any system.

56.3.11 Character sets and character encoding

Assume very little about character sets.

Assume nothing about numerical values (ord, chr) of characters. Do not use explicit code point ranges (like \xHH-\xHH). However, starting in Perl v5.22, regular expression pattern bracketed character class ranges specified like qr/[\N{U+HH}-\N{U+HH}]/ are portable. You can portably use symbolic character classes like [:print:].

Do not assume that the alphabetic characters are encoded contiguously (in the numeric sense). There may be gaps. Special coding in Perl, however, guarantees that all subsets of qr/[A-Z]/, qr/[a-z]/, and qr/[0-9]/ behave as expected. tr/// behaves the same for these ranges. In patterns, any ranges specified with end points using the \N{...} notations ensures character set portability, but it is a bug in Perl v5.22, that this isn't true of tr///.

Do not assume anything about the ordering of the characters. The lowercase letters may come before or after the uppercase letters; the lowercase and uppercase may be interlaced so that both "a" and "A" come before "b"; the accented and other international characters may be interlaced so that ä comes before "b". Unicode-Collate can be used to sort this all out.

56.3.12 Internationalisation

If you may assume POSIX (a rather large assumption), you may read more about the POSIX locale system from Section 38.1 [perllocale NAME], page 701. The locale system at least attempts to make things a little bit more portable, or at least more convenient and native-friendly for non-English users. The system affects character sets and encoding, and date and time formatting–amongst other things.

If you really want to be international, you should consider Unicode. See Section 83.1 [perluniintro NAME], page 1352 and Section 81.1 [perlunicode NAME], page 1317 for more information.

If you want to use non-ASCII bytes (outside the bytes 0x00..0x7f) in the "source code" of your code, to be portable you have to be explicit about what bytes they are. Someone might for example be using your code under a UTF-8 locale, in which case random native bytes might be illegal ("Malformed UTF-8 ...") This means that for example embedding ISO 8859-1 bytes beyond 0x7f into your strings might cause trouble later. If the bytes are native 8-bit bytes, you can use the bytes pragma. If the bytes are in a string (regular expressions being curious strings), you can often also use the \xHH or more portably, the \N{U+HH} notations instead of embedding the bytes as-is. If you want to write your code in UTF-8, you can use utf8.

56.3.13 System Resources

If your code is destined for systems with severely constrained (or missing!) virtual memory systems then you want to be *especially* mindful of avoiding wasteful constructs such as:

```
    my @lines = <$very_large_file>;            # bad

    while (<$fh>) {$file .= $_}                # sometimes bad
    my $file = join('', <$fh>);                # better
```

The last two constructs may appear unintuitive to most people. The first repeatedly grows a string, whereas the second allocates a large chunk of memory in one go. On some systems, the second is more efficient than the first.

56.3.14 Security

Most multi-user platforms provide basic levels of security, usually implemented at the filesystem level. Some, however, unfortunately do not. Thus the notion of user id, or "home" directory, or even the state of being logged-in, may be unrecognizable on many platforms. If you write programs that are security-conscious, it is usually best to know what type of system you will be running under so that you can write code explicitly for that platform (or class of platforms).

Don't assume the Unix filesystem access semantics: the operating system or the filesystem may be using some ACL systems, which are richer languages than the usual `rwx`. Even if the `rwx` exist, their semantics might be different.

(From the security viewpoint, testing for permissions before attempting to do something is silly anyway: if one tries this, there is potential for race conditions. Someone or something might change the permissions between the permissions check and the actual operation. Just try the operation.)

Don't assume the Unix user and group semantics: especially, don't expect `$<` and `$>` (or `$(` and `$)`) to work for switching identities (or memberships).

Don't assume set-uid and set-gid semantics. (And even if you do, think twice: set-uid and set-gid are a known can of security worms.)

56.3.15 Style

For those times when it is necessary to have platform-specific code, consider keeping the platform-specific code in one place, making porting to other platforms easier. Use the `Config` module and the special variable `$^O` to differentiate platforms, as described in Section 56.5 [PLATFORMS], page 963.

Be careful in the tests you supply with your module or programs. Module code may be fully portable, but its tests might not be. This often happens when tests spawn off other processes or call external programs to aid in the testing, or when (as noted above) the tests assume certain things about the filesystem and paths. Be careful not to depend on a specific output style for errors, such as when checking `$!` after a failed system call. Using `$!` for anything else than displaying it as output is doubtful (though see the `Errno` module for testing reasonably portably for error value). Some platforms expect a certain output format, and Perl on those platforms may have been adjusted accordingly. Most specifically, don't anchor a regex when testing an error value.

56.4 CPAN Testers

Modules uploaded to CPAN are tested by a variety of volunteers on different platforms. These CPAN testers are notified by mail of each new upload, and reply to the list with PASS, FAIL, NA (not applicable to this platform), or UNKNOWN (unknown), along with any relevant notations.

The purpose of the testing is twofold: one, to help developers fix any problems in their code that crop up because of lack of testing on other platforms; two, to provide users with information about whether a given module works on a given platform.

Also see:

- Mailing list: cpan-testers-discuss@perl.org

- Testing results: `http://www.cpantesters.org/`

56.5 PLATFORMS

Perl is built with a `$^O` variable that indicates the operating system it was built on. This was implemented to help speed up code that would otherwise have to `use Config` and use the value of `$Config{osname}`. Of course, to get more detailed information about the system, looking into `%Config` is certainly recommended.

`%Config` cannot always be trusted, however, because it was built at compile time. If perl was built in one place, then transferred elsewhere, some values may be wrong. The values may even have been edited after the fact.

56.5.1 Unix

Perl works on a bewildering variety of Unix and Unix-like platforms (see e.g. most of the files in the `hints/` directory in the source code kit). On most of these systems, the value of `$^O` (hence `$Config{'osname'}`, too) is determined either by lowercasing and stripping punctuation from the first field of the string returned by typing `uname -a` (or a similar command) at the shell prompt or by testing the file system for the presence of uniquely named files such as a kernel or header file. Here, for example, are a few of the more popular Unix flavors:

uname	$^O	$Config{'archname'}
AIX	aix	aix
BSD/OS	bsdos	i386-bsdos
Darwin	darwin	darwin
DYNIX/ptx	dynixptx	i386-dynixptx
FreeBSD	freebsd	freebsd-i386
Haiku	haiku	BePC-haiku
Linux	linux	arm-linux
Linux	linux	armv5tel-linux
Linux	linux	i386-linux
Linux	linux	i586-linux
Linux	linux	ppc-linux
HP-UX	hpux	PA-RISC1.1
IRIX	irix	irix
Mac OS X	darwin	darwin
NeXT 3	next	next-fat
NeXT 4	next	OPENSTEP-Mach
openbsd	openbsd	i386-openbsd
OSF1	dec_osf	alpha-dec_osf
reliantunix-n	svr4	RM400-svr4
SCO_SV	sco_sv	i386-sco_sv
SINIX-N	svr4	RM400-svr4
sn4609	unicos	CRAY_C90-unicos
sn6521	unicosmk	t3e-unicosmk
sn9617	unicos	CRAY_J90-unicos
SunOS	solaris	sun4-solaris

```
SunOS         solaris    i86pc-solaris
SunOS4        sunos      sun4-sunos
```

Because the value of `$Config{archname}` may depend on the hardware architecture, it can vary more than the value of `$^O`.

56.5.2 DOS and Derivatives

Perl has long been ported to Intel-style microcomputers running under systems like PC-DOS, MS-DOS, OS/2, and most Windows platforms you can bring yourself to mention (except for Windows CE, if you count that). Users familiar with *COMMAND.COM* or *CMD.EXE* style shells should be aware that each of these file specifications may have subtle differences:

```
my $filespec0 = "c:/foo/bar/file.txt";
my $filespec1 = "c:\\foo\\bar\\file.txt";
my $filespec2 = 'c:\foo\bar\file.txt';
my $filespec3 = 'c:\\foo\\bar\\file.txt';
```

System calls accept either `/` or `\` as the path separator. However, many command-line utilities of DOS vintage treat `/` as the option prefix, so may get confused by filenames containing `/`. Aside from calling any external programs, `/` will work just fine, and probably better, as it is more consistent with popular usage, and avoids the problem of remembering what to backwhack and what not to.

The DOS FAT filesystem can accommodate only "8.3" style filenames. Under the "case-insensitive, but case-preserving" HPFS (OS/2) and NTFS (NT) filesystems you may have to be careful about case returned with functions like `readdir` or used with functions like `open` or `opendir`.

DOS also treats several filenames as special, such as AUX, PRN, NUL, CON, COM1, LPT1, LPT2, etc. Unfortunately, sometimes these filenames won't even work if you include an explicit directory prefix. It is best to avoid such filenames, if you want your code to be portable to DOS and its derivatives. It's hard to know what these all are, unfortunately.

Users of these operating systems may also wish to make use of scripts such as *pl2bat.bat* or *pl2cmd* to put wrappers around your scripts.

Newline (`\n`) is translated as `\015\012` by STDIO when reading from and writing to files (see Section 56.3.1 [Newlines], page 952). `binmode(FILEHANDLE)` will keep `\n` translated as `\012` for that filehandle. Since it is a no-op on other systems, `binmode` should be used for cross-platform code that deals with binary data. That's assuming you realize in advance that your data is in binary. General-purpose programs should often assume nothing about their data.

The `$^O` variable and the `$Config{archname}` values for various DOSish perls are as follows:

```
OS            $^O        $Config{archname}   ID    Version
--------------------------------------------------------
MS-DOS        dos          ?
PC-DOS        dos          ?
OS/2          os2          ?
Windows 3.1   ?            ?                 0      3 01
Windows 95    MSWin32      MSWin32-x86       1      4 00
```

Windows 98	MSWin32	MSWin32-x86	1	4 10
Windows ME	MSWin32	MSWin32-x86	1	?
Windows NT	MSWin32	MSWin32-x86	2	4 xx
Windows NT	MSWin32	MSWin32-ALPHA	2	4 xx
Windows NT	MSWin32	MSWin32-ppc	2	4 xx
Windows 2000	MSWin32	MSWin32-x86	2	5 00
Windows XP	MSWin32	MSWin32-x86	2	5 01
Windows 2003	MSWin32	MSWin32-x86	2	5 02
Windows Vista	MSWin32	MSWin32-x86	2	6 00
Windows 7	MSWin32	MSWin32-x86	2	6 01
Windows 7	MSWin32	MSWin32-x64	2	6 01
Windows 2008	MSWin32	MSWin32-x86	2	6 01
Windows 2008	MSWin32	MSWin32-x64	2	6 01
Windows CE	MSWin32	?	3	
Cygwin	cygwin	cygwin		

The various MSWin32 Perl's can distinguish the OS they are running on via the value of the fifth element of the list returned from `Win32::GetOSVersion()`. For example:

```
if ($^O eq 'MSWin32') {
    my @os_version_info = Win32::GetOSVersion();
    print +('3.1','95','NT')[$os_version_info[4]],"\n";
}
```

There are also `Win32::IsWinNT()` and `Win32::IsWin95()`; try `perldoc Win32`, and as of libwin32 0.19 (not part of the core Perl distribution) `Win32::GetOSName()`. The very portable `POSIX::uname()` will work too:

```
c:\> perl -MPOSIX -we "print join '|', uname"
Windows NT|moonru|5.0|Build 2195 (Service Pack 2)|x86
```

Also see:

- The djgpp environment for DOS, `http://www.delorie.com/djgpp/` and `perldos`.
- The EMX environment for DOS, OS/2, etc. emx@iaehv.nl, `ftp://hobbes.nmsu.edu/pub/os2/dev/emx/` Also `perlos2`.
- Build instructions for Win32 in `perlwin32`, or under the Cygnus environment in `perlcygwin`.
- The `Win32::*` modules in `Win32`.
- The ActiveState Pages, `http://www.activestate.com/`
- The Cygwin environment for Win32; `README.cygwin` (installed as `perlcygwin`), `http://www.cygwin.com/`
- The U/WIN environment for Win32, `http://www.research.att.com/sw/tools/uwin/`
- Build instructions for OS/2, `perlos2`

56.5.3 VMS

Perl on VMS is discussed in Section 87.1 [perlvms NAME], page 1409 in the Perl distribution.

The official name of VMS as of this writing is OpenVMS.

Interacting with Perl from the Digital Command Language (DCL) shell often requires a different set of quotation marks than Unix shells do. For example:

```
$ perl -e "print ""Hello, world.\n"""
Hello, world.
```

There are several ways to wrap your Perl scripts in DCL `.COM` files, if you are so inclined. For example:

```
$ write sys$output "Hello from DCL!"
$ if p1 .eqs. ""
$ then perl -x 'f$environment("PROCEDURE")
$ else perl -x - 'p1 'p2 'p3 'p4 'p5 'p6 'p7 'p8
$ deck/dollars="__END__"
#!/usr/bin/perl

print "Hello from Perl!\n";

__END__
$ endif
```

Do take care with `$ ASSIGN/nolog/user SYS$COMMAND: SYS$INPUT` if your Perl-in-DCL script expects to do things like `$read = <STDIN>;`.

The VMS operating system has two filesystems, designated by their on-disk structure (ODS) level: ODS-2 and its successor ODS-5. The initial port of Perl to VMS pre-dates ODS-5, but all current testing and development assumes ODS-5 and its capabilities, including case preservation, extended characters in filespecs, and names up to 8192 bytes long.

Perl on VMS can accept either VMS- or Unix-style file specifications as in either of the following:

```
$ perl -ne "print if /perl_setup/i" SYS$LOGIN:LOGIN.COM
$ perl -ne "print if /perl_setup/i" /sys$login/login.com
```

but not a mixture of both as in:

```
$ perl -ne "print if /perl_setup/i" sys$login:/login.com
Can't open sys$login:/login.com: file specification syntax error
```

In general, the easiest path to portability is always to specify filenames in Unix format unless they will need to be processed by native commands or utilities. Because of this latter consideration, the File::Spec module by default returns native format specifications regardless of input format. This default may be reversed so that filenames are always reported in Unix format by specifying the `DECC$FILENAME_UNIX_REPORT` feature logical in the environment.

The file type, or extension, is always present in a VMS-format file specification even if it's zero-length. This means that, by default, `readdir` will return a trailing dot on a file with no extension, so where you would see `"a"` on Unix you'll see `"a."` on VMS. However, the trailing dot may be suppressed by enabling the `DECC$READDIR_DROPDOTNOTYPE` feature in the environment (see the CRTL documentation on feature logical names).

What `\n` represents depends on the type of file opened. It usually represents `\012` but it could also be `\015`, `\012`, `\015\012`, `\000`, `\040`, or nothing depending on the file organization and record format. The VMS::Stdio module provides access to the special fopen() requirements of files with unusual attributes on VMS.

The value of $^O on OpenVMS is "VMS". To determine the architecture that you are running on refer to $Config{'archname'}.

On VMS, perl determines the UTC offset from the SYS$TIMEZONE_DIFFERENTIAL logical name. Although the VMS epoch began at 17-NOV-1858 00:00:00.00, calls to localtime are adjusted to count offsets from 01-JAN-1970 00:00:00.00, just like Unix.

Also see:

- README.vms (installed as README_vms), Section 87.1 [perlvms NAME], page 1409
- vmsperl list, vmsperl-subscribe@perl.org
- vmsperl on the web, http://www.sidhe.org/vmsperl/index.html
- VMS Software Inc. web site, http://www.vmssoftware.com

56.5.4 VOS

Perl on VOS (also known as OpenVOS) is discussed in README.vos in the Perl distribution (installed as perlvos). Perl on VOS can accept either VOS- or Unix-style file specifications as in either of the following:

```
$ perl -ne "print if /perl_setup/i" >system>notices
$ perl -ne "print if /perl_setup/i" /system/notices
```

or even a mixture of both as in:

```
$ perl -ne "print if /perl_setup/i" >system/notices
```

Even though VOS allows the slash character to appear in object names, because the VOS port of Perl interprets it as a pathname delimiting character, VOS files, directories, or links whose names contain a slash character cannot be processed. Such files must be renamed before they can be processed by Perl.

Older releases of VOS (prior to OpenVOS Release 17.0) limit file names to 32 or fewer characters, prohibit file names from starting with a - character, and prohibit file names from containing any character matching tr/ !#%&'()*;<=>?//.

Newer releases of VOS (OpenVOS Release 17.0 or later) support a feature known as extended names. On these releases, file names can contain up to 255 characters, are prohibited from starting with a - character, and the set of prohibited characters is reduced to any character matching tr/#%*<>?//. There are restrictions involving spaces and apostrophes: these characters must not begin or end a name, nor can they immediately precede or follow a period. Additionally, a space must not immediately precede another space or hyphen. Specifically, the following character combinations are prohibited: space-space, space-hyphen, period-space, space-period, period-apostrophe, apostrophe-period, leading or trailing space, and leading or trailing apostrophe. Although an extended file name is limited to 255 characters, a path name is still limited to 256 characters.

The value of $^O on VOS is "vos". To determine the architecture that you are running on without resorting to loading all of %Config you can examine the content of the @INC array like so:

```
if ($^O =~ /vos/) {
    print "I'm on a Stratus box!\n";
} else {
    print "I’m not on a Stratus box!\n";
    die;
}
```

Also see:

- README.vos (installed as perlvos)
- The VOS mailing list.

  There is no specific mailing list for Perl on VOS. You can contact the Stratus Technologies Customer Assistance Center (CAC) for your region, or you can use the contact information located in the distribution files on the Stratus Anonymous FTP site.
- Stratus Technologies on the web at http://www.stratus.com
- VOS Open-Source Software on the web at http://ftp.stratus.com/pub/vos/vos.html

56.5.5 EBCDIC Platforms

v5.22 core Perl runs on z/OS (formerly OS/390). Theoretically it could run on the successors of OS/400 on AS/400 minicomputers as well as VM/ESA, and BS2000 for S/390 Mainframes. Such computers use EBCDIC character sets internally (usually Character Code Set ID 0037 for OS/400 and either 1047 or POSIX-BC for S/390 systems).

The rest of this section may need updating, but we don’t know what it should say. Please email comments to perlbug@perl.org (mailto:perlbug@perl.org).

On the mainframe Perl currently works under the "Unix system services for OS/390" (formerly known as OpenEdition), VM/ESA OpenEdition, or the BS200 POSIX-BC system (BS2000 is supported in Perl 5.6 and greater). See perlos390 for details. Note that for OS/400 there is also a port of Perl 5.8.1/5.10.0 or later to the PASE which is ASCII-based (as opposed to ILE which is EBCDIC-based), see perlos400.

As of R2.5 of USS for OS/390 and Version 2.3 of VM/ESA these Unix sub-systems do not support the #! shebang trick for script invocation. Hence, on OS/390 and VM/ESA Perl scripts can be executed with a header similar to the following simple script:

```
: # use perl
    eval ’exec /usr/local/bin/perl -S $0 ${1+"$@"}’
        if 0;
#!/usr/local/bin/perl     # just a comment really

print "Hello from perl!\n";
```

OS/390 will support the #! shebang trick in release 2.8 and beyond. Calls to system and backticks can use POSIX shell syntax on all S/390 systems.

On the AS/400, if PERL5 is in your library list, you may need to wrap your Perl scripts in a CL procedure to invoke them like so:

```
BEGIN
  CALL PGM(PERL5/PERL) PARM(’/QOpenSys/hello.pl’)
ENDPGM
```

This will invoke the Perl script hello.pl in the root of the QOpenSys file system. On the AS/400 calls to system or backticks must use CL syntax.

On these platforms, bear in mind that the EBCDIC character set may have an effect on what happens with some Perl functions (such as `chr`, `pack`, `print`, `printf`, `ord`, `sort`, `sprintf`, `unpack`), as well as bit-fiddling with ASCII constants using operators like `^`, `&` and `|`, not to mention dealing with socket interfaces to ASCII computers (see Section 56.3.1 [Newlines], page 952).

Fortunately, most web servers for the mainframe will correctly translate the `\n` in the following statement to its ASCII equivalent (`\r` is the same under both Unix and z/OS):

```
print "Content-type: text/html\r\n\r\n";
```

The values of `$^O` on some of these platforms includes:

```
uname         $^O        $Config{'archname'}
--------------------------------------------
OS/390        os390      os390
OS400         os400      os400
POSIX-BC      posix-bc   BS2000-posix-bc
```

Some simple tricks for determining if you are running on an EBCDIC platform could include any of the following (perhaps all):

```
if ("\t" eq "\005")   { print "EBCDIC may be spoken here!\n"; }

if (ord('A') == 193) { print "EBCDIC may be spoken here!\n"; }

if (chr(169) eq 'z') { print "EBCDIC may be spoken here!\n"; }
```

One thing you may not want to rely on is the EBCDIC encoding of punctuation characters since these may differ from code page to code page (and once your module or script is rumoured to work with EBCDIC, folks will want it to work with all EBCDIC character sets).

Also see:

- `perlos390`, `perlos400`, `perlbs2000`, Section 19.1 [perlebcdic NAME], page 266.
- The perl-mvs@perl.org list is for discussion of porting issues as well as general usage issues for all EBCDIC Perls. Send a message body of "subscribe perl-mvs" to majordomo@perl.org.
- AS/400 Perl information at `http://as400.rochester.ibm.com/` as well as on CPAN in the `ports/` directory.

56.5.6 Acorn RISC OS

Because Acorns use ASCII with newlines (`\n`) in text files as `\012` like Unix, and because Unix filename emulation is turned on by default, most simple scripts will probably work "out of the box". The native filesystem is modular, and individual filesystems are free to be case-sensitive or insensitive, and are usually case-preserving. Some native filesystems have name length limits, which file and directory names are silently truncated to fit. Scripts should be aware that the standard filesystem currently has a name length limit of **10** characters, with up to 77 items in a directory, but other filesystems may not impose such limitations.

Native filenames are of the form

```
Filesystem#Special_Field::DiskName.$.Directory.Directory.File
```

where

```
Special_Field is not usually present, but may contain . and $ .
Filesystem =~ m|[A-Za-z0-9_]|
DsicName   =~ m|[A-Za-z0-9_/]|
$ represents the root directory
. is the path separator
@ is the current directory (per filesystem but machine global)
^ is the parent directory
Directory and File =~ m|[^\0- "\.\$\%\&:\@\\^\|\177]+|
```

The default filename translation is roughly `tr|/.|./|;`

Note that `"ADFS::HardDisk.$.File" ne 'ADFS::HardDisk.$.File'` and that the second stage of `$` interpolation in regular expressions will fall foul of the `$.` if scripts are not careful.

Logical paths specified by system variables containing comma-separated search lists are also allowed; hence `System:Modules` is a valid filename, and the filesystem will prefix `Modules` with each section of `System$Path` until a name is made that points to an object on disk. Writing to a new file `System:Modules` would be allowed only if `System$Path` contains a single item list. The filesystem will also expand system variables in filenames if enclosed in angle brackets, so `<System$Dir>.Modules` would look for the file `$ENV{'System$Dir'} . 'Modules'`. The obvious implication of this is that **fully qualified filenames can start with `<>`** and should be protected when `open` is used for input.

Because `.` was in use as a directory separator and filenames could not be assumed to be unique after 10 characters, Acorn implemented the C compiler to strip the trailing `.c` `.h` `.s` and `.o` suffix from filenames specified in source code and store the respective files in subdirectories named after the suffix. Hence files are translated:

```
foo.h           h.foo
C:foo.h         C:h.foo        (logical path variable)
sys/os.h        sys.h.os       (C compiler groks Unix-speak)
10charname.c    c.10charname
10charname.o    o.10charname
11charname_.c   c.11charname   (assuming filesystem truncates at 10)
```

The Unix emulation library's translation of filenames to native assumes that this sort of translation is required, and it allows a user-defined list of known suffixes that it will transpose in this fashion. This may seem transparent, but consider that with these rules `foo/bar/baz.h` and `foo/bar/h/baz` both map to `foo.bar.h.baz`, and that `readdir` and `glob` cannot and do not attempt to emulate the reverse mapping. Other `.`'s in filenames are translated to `/`.

As implied above, the environment accessed through `%ENV` is global, and the convention is that program specific environment variables are of the form `Program$Name`. Each filesystem maintains a current directory, and the current filesystem's current directory is the **global** current directory. Consequently, sociable programs don't change the current directory but rely on full pathnames, and programs (and Makefiles) cannot assume that they can spawn a child process which can change the current directory without affecting its parent (and everyone else for that matter).

Because native operating system filehandles are global and are currently allocated down from 255, with 0 being a reserved value, the Unix emulation library emulates Unix filehandles. Consequently, you can't rely on passing `STDIN`, `STDOUT`, or `STDERR` to your children.

The desire of users to express filenames of the form `<Foo$Dir>.Bar` on the command line unquoted causes problems, too: `` ` ` `` command output capture has to perform a guessing game. It assumes that a string `<[^<>]+\$[^<>]>` is a reference to an environment variable, whereas anything else involving `<` or `>` is redirection, and generally manages to be 99% right. Of course, the problem remains that scripts cannot rely on any Unix tools being available, or that any tools found have Unix-like command line arguments.

Extensions and XS are, in theory, buildable by anyone using free tools. In practice, many don't, as users of the Acorn platform are used to binary distributions. MakeMaker does run, but no available make currently copes with MakeMaker's makefiles; even if and when this should be fixed, the lack of a Unix-like shell will cause problems with makefile rules, especially lines of the form `cd sdbm && make all`, and anything using quoting.

"RISC OS" is the proper name for the operating system, but the value in `$^O` is "riscos" (because we don't like shouting).

56.5.7 Other perls

Perl has been ported to many platforms that do not fit into any of the categories listed above. Some, such as AmigaOS, QNX, Plan 9, and VOS, have been well-integrated into the standard Perl source code kit. You may need to see the `ports/` directory on CPAN for information, and possibly binaries, for the likes of: aos, Atari ST, lynxos, riscos, Novell Netware, Tandem Guardian, *etc.* (Yes, we know that some of these OSes may fall under the Unix category, but we are not a standards body.)

Some approximate operating system names and their `$^O` values in the "OTHER" category include:

```
OS            $^O        $Config{'archname'}
------------------------------------------
Amiga DOS     amigaos    m68k-amigos
```

See also:

- Amiga, `README.amiga` (installed as `perlamiga`).
- A free perl5-based PERL.NLM for Novell Netware is available in precompiled binary and source code form from `http://www.novell.com/` as well as from CPAN.
- Plan 9, `README.plan9`

56.6 FUNCTION IMPLEMENTATIONS

Listed below are functions that are either completely unimplemented or else have been implemented differently on various platforms. Following each description will be, in parentheses, a list of platforms that the description applies to.

The list may well be incomplete, or even wrong in some places. When in doubt, consult the platform-specific README files in the Perl source distribution, and any other documentation resources accompanying a given port.

Be aware, moreover, that even among Unix-ish systems there are variations.

For many functions, you can also query `%Config`, exported by default from the `Config` module. For example, to check whether the platform has the `lstat` call, check `$Config{d_lstat}`. See `Config` for a full description of available variables.

56.6.1 Alphabetical Listing of Perl Functions

-X

`-w` only inspects the read-only file attribute (FILE_ATTRIBUTE_READONLY), which determines whether the directory can be deleted, not whether it can be written to. Directories always have read and write access unless denied by discretionary access control lists (DACLs). (Win32)

`-r`, `-w`, `-x`, and `-o` tell whether the file is accessible, which may not reflect UIC-based file protections. (VMS)

`-s` by name on an open file will return the space reserved on disk, rather than the current extent. `-s` on an open filehandle returns the current size. (RISC OS)

`-R`, `-W`, `-X`, `-O` are indistinguishable from `-r`, `-w`, `-x`, `-o`. (Win32, VMS, RISC OS)

`-g`, `-k`, `-l`, `-u`, `-A` are not particularly meaningful. (Win32, VMS, RISC OS)

`-p` is not particularly meaningful. (VMS, RISC OS)

`-d` is true if passed a device spec without an explicit directory. (VMS)

`-x` (or `-X`) determine if a file ends in one of the executable suffixes. `-S` is meaningless. (Win32)

`-x` (or `-X`) determine if a file has an executable file type. (RISC OS)

alarm

Emulated using timers that must be explicitly polled whenever Perl wants to dispatch "safe signals" and therefore cannot interrupt blocking system calls. (Win32)

atan2

Due to issues with various CPUs, math libraries, compilers, and standards, results for `atan2()` may vary depending on any combination of the above. Perl attempts to conform to the Open Group/IEEE standards for the results returned from `atan2()`, but cannot force the issue if the system Perl is run on does not allow it. (Tru64, HP-UX 10.20)

The current version of the standards for `atan2()` is available at `http://www.opengroup.org/onlinepubs/009695399/functions/atan2.html`.

binmode

Meaningless. (RISC OS)

Reopens file and restores pointer; if function fails, underlying filehandle may be closed, or pointer may be in a different position. (VMS)

The value returned by `tell` may be affected after the call, and the filehandle may be flushed. (Win32)

chmod

Only good for changing "owner" read-write access, "group", and "other" bits are meaningless. (Win32)

Only good for changing "owner" and "other" read-write access. (RISC OS)

Access permissions are mapped onto VOS access-control list changes. (VOS)

The actual permissions set depend on the value of the `CYGWIN` in the SYSTEM environment settings. (Cygwin)

Setting the exec bit on some locations (generally `/sdcard`) will return true but not actually set the bit. (Android)

chown

Not implemented. (Win32, Plan 9, RISC OS)

Does nothing, but won't fail. (Win32)

A little funky, because VOS's notion of ownership is a little funky (VOS).

chroot

Not implemented. (Win32, VMS, Plan 9, RISC OS, VOS)

crypt

May not be available if library or source was not provided when building perl. (Win32)

Not implemented. (Android)

dbmclose

Not implemented. (VMS, Plan 9, VOS)

dbmopen

Not implemented. (VMS, Plan 9, VOS)

dump

Not useful. (RISC OS)

Not supported. (Cygwin, Win32)

Invokes VMS debugger. (VMS)

exec

`exec LIST` without the use of indirect object syntax (`exec PROGRAM LIST`) may fall back to trying the shell if the first `spawn()` fails. (Win32)

Does not automatically flush output handles on some platforms. (SunOS, Solaris, HP-UX)

Not supported. (Symbian OS)

exit

Emulates Unix `exit()` (which considers `exit 1` to indicate an error) by mapping the `1` to `SS$_ABORT` (`44`). This behavior may be overridden with the pragma `use vmsish 'exit'`. As with the CRTL's `exit()` function, `exit 0` is also mapped to an exit status of `SS$_NORMAL` (`1`); this mapping cannot be overridden. Any other argument to `exit()` is used directly as Perl's exit status. On VMS, unless the future POSIX_EXIT mode is enabled, the exit code should always be a valid

VMS exit code and not a generic number. When the POSIX_EXIT mode is enabled, a generic number will be encoded in a method compatible with the C library _POSIX_EXIT macro so that it can be decoded by other programs, particularly ones written in C, like the GNV package. (VMS)

`exit()` resets file pointers, which is a problem when called from a child process (created by `fork()`) in `BEGIN`. A workaround is to use `POSIX::_exit`. (Solaris)

```
    exit unless $Config{archname} =~ /\bsolaris\b/;
    require POSIX and POSIX::_exit(0);
```

fcntl

Not implemented. (Win32)

Some functions available based on the version of VMS. (VMS)

flock

Not implemented (VMS, RISC OS, VOS).

fork

Not implemented. (AmigaOS, RISC OS, VMS)

Emulated using multiple interpreters. See Section 23.1 [perlfork NAME], page 337. (Win32)

Does not automatically flush output handles on some platforms. (SunOS, Solaris, HP-UX)

getlogin

Not implemented. (RISC OS)

getpgrp

Not implemented. (Win32, VMS, RISC OS)

getppid

Not implemented. (Win32, RISC OS)

getpriority

Not implemented. (Win32, VMS, RISC OS, VOS)

getpwnam

Not implemented. (Win32)

Not useful. (RISC OS)

getgrnam

Not implemented. (Win32, VMS, RISC OS)

getnetbyname

Not implemented. (Android, Win32, Plan 9)

getpwuid

Not implemented. (Win32)

Not useful. (RISC OS)

getgrgid

Not implemented. (Win32, VMS, RISC OS)

getnetbyaddr

Not implemented. (Android, Win32, Plan 9)

getprotobynumber

Not implemented. (Android)

getservbyport
getpwent

Not implemented. (Android, Win32)

getgrent

Not implemented. (Android, Win32, VMS)

gethostbyname

`gethostbyname('localhost')` does not work everywhere: you may have to use `gethostbyname('127.0.0.1')`. (Irix 5)

gethostent

Not implemented. (Win32)

getnetent

Not implemented. (Android, Win32, Plan 9)

getprotoent

Not implemented. (Android, Win32, Plan 9)

getservent

Not implemented. (Win32, Plan 9)

seekdir

Not implemented. (Android)

sethostent

Not implemented. (Android, Win32, Plan 9, RISC OS)

setnetent

Not implemented. (Win32, Plan 9, RISC OS)

setprotoent

Not implemented. (Android, Win32, Plan 9, RISC OS)

setservent

Not implemented. (Plan 9, Win32, RISC OS)

endpwent

Not implemented. (Win32)

Either not implemented or a no-op. (Android)

endgrent

Not implemented. (Android, RISC OS, VMS, Win32) endhostent

Not implemented. (Android, Win32)

endnetent

Not implemented. (Android, Win32, Plan 9)

endprotoent

Not implemented. (Android, Win32, Plan 9)

endservent

Not implemented. (Plan 9, Win32)

getsockopt SOCKET,LEVEL,OPTNAME

Not implemented. (Plan 9)

glob

This operator is implemented via the `File::Glob` extension on most platforms. See `File-Glob` for portability information.

gmtime

In theory, `gmtime()` is reliable from -2**63 to 2**63-1. However, because work arounds in the implementation use floating point numbers, it will become inaccurate as the time gets larger. This is a bug and will be fixed in the future.

On VOS, time values are 32-bit quantities.

ioctl FILEHANDLE,FUNCTION,SCALAR

Not implemented. (VMS)

Available only for socket handles, and it does what the `ioctlsocket()` call in the Winsock API does. (Win32)

Available only for socket handles. (RISC OS)

kill

Not implemented, hence not useful for taint checking. (RISC OS)

`kill()` doesn't have the semantics of `raise()`, i.e. it doesn't send a signal to the identified process like it does on Unix platforms. Instead `kill($sig, $pid)` terminates the process identified by `$pid`, and makes it exit immediately with exit status $sig. As in Unix, if $sig is 0 and the specified process exists, it returns true without actually terminating it. (Win32)

`kill(-9, $pid)` will terminate the process specified by `$pid` and recursively all child processes owned by it. This is different from the Unix semantics, where the signal will be delivered to all processes in the same process group as the process specified by $pid. (Win32)

Is not supported for process identification number of 0 or negative numbers. (VMS)

link

Not implemented. (RISC OS, VOS)

Link count not updated because hard links are not quite that hard (They are sort of half-way between hard and soft links). (AmigaOS)

Hard links are implemented on Win32 under NTFS only. They are natively supported on Windows 2000 and later. On Windows NT they are implemented using the Windows POSIX subsystem support and the Perl process will need Administrator or Backup Operator privileges to create hard links.

Available on 64 bit OpenVMS 8.2 and later. (VMS)

localtime

localtime() has the same range as [gmtime], page 976, but because time zone rules change its accuracy for historical and future times may degrade but usually by no more than an hour.

lstat

Not implemented. (RISC OS)

Return values (especially for device and inode) may be bogus. (Win32)

msgctl

msgget

msgsnd

msgrcv

Not implemented. (Android, Win32, VMS, Plan 9, RISC OS, VOS)

open

open to `|-` and `-|` are unsupported. (Win32, RISC OS)

Opening a process does not automatically flush output handles on some platforms. (SunOS, Solaris, HP-UX)

readlink

Not implemented. (Win32, VMS, RISC OS)

rename

Can't move directories between directories on different logical volumes. (Win32)

rewinddir

Will not cause `readdir()` to re-read the directory stream. The entries already read before the `rewinddir()` call will just be returned again from a cache buffer. (Win32)

select

Only implemented on sockets. (Win32, VMS)

Only reliable on sockets. (RISC OS)

Note that the `select FILEHANDLE` form is generally portable.

semctl

semget

semop

Not implemented. (Android, Win32, VMS, RISC OS) setgrent

Not implemented. (Android, VMS, Win32, RISC OS)

setpgrp

Not implemented. (Win32, VMS, RISC OS, VOS)

setpriority

Not implemented. (Win32, VMS, RISC OS, VOS)

setpwent

Not implemented. (Android, Win32, RISC OS)

setsockopt

Not implemented. (Plan 9)

shmctl

shmget

shmread

shmwrite

Not implemented. (Android, Win32, VMS, RISC OS)

sleep

Emulated using synchronization functions such that it can be interrupted by `alarm()`, and limited to a maximum of 4294967 seconds, approximately 49 days. (Win32)

sockatmark

A relatively recent addition to socket functions, may not be implemented even in Unix platforms.

socketpair

Not implemented. (RISC OS)

Available on 64 bit OpenVMS 8.2 and later. (VMS)

stat

Platforms that do not have rdev, blksize, or blocks will return these as ”, so numeric comparison or manipulation of these fields may cause ’not numeric’ warnings.

ctime not supported on UFS (Mac OS X).

ctime is creation time instead of inode change time (Win32).

device and inode are not meaningful. (Win32)

device and inode are not necessarily reliable. (VMS)

mtime, atime and ctime all return the last modification time. Device and inode are not necessarily reliable. (RISC OS)

dev, rdev, blksize, and blocks are not available. inode is not meaningful and will differ between stat calls on the same file. (os2) some versions of cygwin when doing a `stat("foo")` and if not finding it may then attempt to `stat("foo.exe")` (Cygwin)

On Win32 `stat()` needs to open the file to determine the link count and update attributes that may have been changed through hard links. Setting `${^WIN32_SLOPPY_STAT}` to a true value speeds up `stat()` by not performing this operation. (Win32)

symlink

Not implemented. (Win32, RISC OS)

Implemented on 64 bit VMS 8.3. VMS requires the symbolic link to be in Unix syntax if it is intended to resolve to a valid path.

syscall

Not implemented. (Win32, VMS, RISC OS, VOS)

sysopen

The traditional `"0"`, `"1"`, and `"2"` MODEs are implemented with different numeric values on some systems. The flags exported by `Fcntl` (O_RDONLY, O_WRONLY, O_RDWR) should work everywhere though. (Mac OS, OS/390)

system

As an optimization, may not call the command shell specified in `$ENV{PERL5SHELL}`. `system(1, @args)` spawns an external process and immediately returns its process designator, without waiting for it to terminate. Return value may be used subsequently in `wait` or `waitpid`. Failure to `spawn()` a subprocess is indicated by setting `$?` to `"255 << 8"`. `$?` is set in a way compatible with Unix (i.e. the exitstatus of the subprocess is obtained by `"$? 8">>`, as described in the documentation). (Win32)

There is no shell to process metacharacters, and the native standard is to pass a command line terminated by `"\n"` `"\r"` or `"\0"` to the spawned program. Redirection such as `> foo` is performed (if at all) by the run time library of the spawned program. `system` *list* will call the Unix emulation library's `exec` emulation, which attempts to provide emulation of the stdin, stdout, stderr in force in the parent, providing the child program uses a compatible version of the emulation library. *scalar* will call the native command line direct and no such emulation of a child Unix program will exists. Mileage **will** vary. (RISC OS)

`system LIST` without the use of indirect object syntax (`system PROGRAM LIST`) may fall back to trying the shell if the first `spawn()` fails. (Win32)

Does not automatically flush output handles on some platforms. (SunOS, Solaris, HP-UX)

The return value is POSIX-like (shifted up by 8 bits), which only allows room for a made-up value derived from the severity bits of the native 32-bit condition code (unless overridden by `use vmsish 'status'`). If the native condition code is one that has a POSIX value encoded, the POSIX value will be decoded to extract the expected exit value. For more details see [perlvms $?], page 1424. (VMS) telldir

Not implemented. (Android)

times

"cumulative" times will be bogus. On anything other than Windows NT or Windows 2000, "system" time will be bogus, and "user" time is actually the time returned by the `clock()` function in the C runtime library. (Win32)

Not useful. (RISC OS)

truncate

Not implemented. (Older versions of VMS)

Truncation to same-or-shorter lengths only. (VOS)

If a FILEHANDLE is supplied, it must be writable and opened in append mode (i.e., use `open(FH, '>>filename')` or `sysopen(FH,...,O_APPEND|O_RDWR)`. If a filename is supplied, it should not be held open elsewhere. (Win32)

umask

Returns undef where unavailable.

`umask` works but the correct permissions are set only when the file is finally closed. (AmigaOS)

utime

Only the modification time is updated. (VMS, RISC OS)

May not behave as expected. Behavior depends on the C runtime library's implementation of `utime()`, and the filesystem being used. The FAT filesystem typically does not support an "access time" field, and it may limit timestamps to a granularity of two seconds. (Win32)

wait

waitpid

Can only be applied to process handles returned for processes spawned using `system(1, ...)` or pseudo processes created with `fork()`. (Win32)

Not useful. (RISC OS)

56.7 Supported Platforms

The following platforms are known to build Perl 5.12 (as of April 2010, its release date) from the standard source code distribution available at `http://www.cpan.org/src`

Linux (x86, ARM, IA64)
HP-UX

AIX

Win32

Windows 2000
Windows XP
Windows Server 2003
Windows Vista
Windows Server 2008
Windows 7

Cygwin

Some tests are known to fail:

- `ext/XS-APItes/t/call_checker.t` - see `https://rt.perl.org/Ticket/Display.html?id=78502`
- `dist/I18N-Collate/t/I18N-Collate.t`
- `ext/Win32CORE/t/win32core.t` - may fail on recent cygwin installs.

Solaris (x86, SPARC)
OpenVMS

Alpha (7.2 and later)
I64 (8.2 and later)

Symbian

NetBSD

FreeBSD

Debian GNU/kFreeBSD
Haiku

Irix (6.5. What else?)
OpenBSD

Dragonfly BSD
Midnight BSD
QNX Neutrino RTOS (6.5.0)
MirOS BSD
Stratus OpenVOS (17.0 or later)
Caveats:

time_t issues that may or may not be fixed

Symbian (Series 60 v3, 3.2 and 5 - what else?)
Stratus VOS / OpenVOS
AIX

Android

FreeMINT

Perl now builds with FreeMiNT/Atari. It fails a few tests, that needs some investigation.

The FreeMiNT port uses GNU dld for loadable module capabilities. So ensure you have that library installed when building perl.

56.8 EOL Platforms

56.8.1 (Perl 5.20)

The following platforms were supported by a previous version of Perl but have been officially removed from Perl's source code as of 5.20:

AT&T 3b1

56.8.2 (Perl 5.14)

The following platforms were supported up to 5.10. They may still have worked in 5.12, but supporting code has been removed for 5.14:

Windows 95
Windows 98
Windows ME
Windows NT4

56.8.3 (Perl 5.12)

The following platforms were supported by a previous version of Perl but have been officially removed from Perl's source code as of 5.12:

Atari MiNT
Apollo Domain/OS
Apple Mac OS 8/9
Tenon Machten

56.9 Supported Platforms (Perl 5.8)

As of July 2002 (the Perl release 5.8.0), the following platforms were able to build Perl from the standard source code distribution available at `http://www.cpan.org/src/`

```
AIX
BeOS
BSD/OS          (BSDi)
Cygwin
DG/UX
DOS DJGPP       1)
DYNIX/ptx
EPOC R5
FreeBSD
HI-UXMPP        (Hitachi) (5.8.0 worked but we didn't know it)
HP-UX
IRIX
Linux
Mac OS Classic
Mac OS X        (Darwin)
MPE/iX
NetBSD
NetWare
NonStop-UX
```

```
ReliantUNIX     (formerly SINIX)
OpenBSD
OpenVMS         (formerly VMS)
Open UNIX       (Unixware) (since Perl 5.8.1/5.9.0)
OS/2
OS/400          (using the PASE) (since Perl 5.8.1/5.9.0)
PowerUX
POSIX-BC        (formerly BS2000)
QNX
Solaris
SunOS 4
SUPER-UX        (NEC)
Tru64 UNIX      (formerly DEC OSF/1, Digital UNIX)
UNICOS
UNICOS/mk
UTS
VOS / OpenVOS
Win95/98/ME/2K/XP 2)
WinCE
z/OS            (formerly OS/390)
VM/ESA

1) in DOS mode either the DOS or OS/2 ports can be used
2) compilers: Borland, MinGW (GCC), VC6
```

The following platforms worked with the previous releases (5.6 and 5.7), but we did not manage either to fix or to test these in time for the 5.8.0 release. There is a very good chance that many of these will work fine with the 5.8.0.

```
BSD/OS
DomainOS
Hurd
LynxOS
MachTen
PowerMAX
SCO SV
SVR4
Unixware
Windows 3.1
```

Known to be broken for 5.8.0 (but 5.6.1 and 5.7.2 can be used):

```
AmigaOS
```

The following platforms have been known to build Perl from source in the past (5.005_03 and earlier), but we haven't been able to verify their status for the current release, either because the hardware/software platforms are rare or because we don't have an active champion on these platforms–or both. They used to work, though, so go ahead and try compiling them, and let perlbug@perl.org of any trouble.

```
3b1
```

```
A/UX
ConvexOS
CX/UX
DC/OSx
DDE SMES
DOS EMX
Dynix
EP/IX
ESIX
FPS
GENIX
Greenhills
ISC
MachTen 68k
MPC
NEWS-OS
NextSTEP
OpenSTEP
Opus
Plan 9
RISC/os
SCO ODT/OSR
Stellar
SVR2
TI1500
TitanOS
Ultrix
Unisys Dynix
```

The following platforms have their own source code distributions and binaries available via `http://www.cpan.org/ports/`

	Perl release
OS/400 (ILE)	5.005_02
Tandem Guardian	5.004

The following platforms have only binaries available via `http://www.cpan.org/ports/index.html` :

	Perl release
Acorn RISCOS	5.005_02
AOS	5.002
LynxOS	5.004_02

Although we do suggest that you always build your own Perl from the source code, both for maximal configurability and for security, in case you are in a hurry you can check `http://www.cpan.org/ports/index.html` for binary distributions.

56.10 SEE ALSO

`perlaix`, `perlamiga`, `perlbs2000`, `perlce`, `perlcygwin`, `perldos`, Section 19.1 [perlebcdic NAME], page 266, `perlfreebsd`, `perlhurd`, `perlhpux`, `perlirix`, `perlmacos`, `perlmacosx`, `perlnetware`, `perlos2`, `perlos390`, `perlos400`, `perlplan9`, `perlqnx`, `perlsolaris`, `perltru64`, Section 81.1 [perlunicode NAME], page 1317, Section 87.1 [perlvms NAME], page 1409, `perlvos`, `perlwin32`, and `Win32`.

56.11 AUTHORS / CONTRIBUTORS

Abigail <abigail@foad.org>, Charles Bailey <bailey@newman.upenn.edu>, Graham Barr <gbarr@pobox.com>, Tom Christiansen <tchrist@perl.com>, Nicholas Clark <nick@ccl4.org>, Thomas Dorner <Thomas.Dorner@start.de>, Andy Dougherty <doughera@lafayette.edu>, Dominic Dunlop <domo@computer.org>, Neale Ferguson <neale@vma.tabnsw.com.au>, David J. Fiander <davidf@mks.com>, Paul Green <Paul.Green@stratus.com>, M.J.T. Guy <mjtg@cam.ac.uk>, Jarkko Hietaniemi <jhi@iki.fi>, Luther Huffman <lutherh@stratcom.com>, Nick Ing-Simmons <nick@ing-simmons.net>, Andreas J. König <a.koenig@mind.de>, Markus Laker <mlaker@contax.co.uk>, Andrew M. Langmead <aml@world.std.com>, Larry Moore <ljmoore@freespace.net>, Paul Moore <Paul.Moore@uk.origin-it.com>, Chris Nandor <pudge@pobox.com>, Matthias Neeracher <neeracher@mac.com>, Philip Newton <pne@cpan.org>, Gary Ng <71564.1743@CompuServe.COM>, Tom Phoenix <rootbeer@teleport.com>, André Pirard <A.Pirard@ulg.ac.be>, Peter Prymmer <pvhp@forte.com>, Hugo van der Sanden <hv@crypt0.demon.co.uk>, Gurusamy Sarathy <gsar@activestate.com>, Paul J. Schinder <schinder@pobox.com>, Michael G Schwern <schwern@pobox.com>, Dan Sugalski <dan@sidhe.org>, Nathan Torkington <gnat@frii.com>, John Malmberg <wb8tyw@qsl.net>

57 perlpragma

57.1 NAME

perlpragma - how to write a user pragma

57.2 DESCRIPTION

A pragma is a module which influences some aspect of the compile time or run time behaviour of Perl, such as `strict` or `warnings`. With Perl 5.10 you are no longer limited to the built in pragmata; you can now create user pragmata that modify the behaviour of user functions within a lexical scope.

57.3 A basic example

For example, say you need to create a class implementing overloaded mathematical operators, and would like to provide your own pragma that functions much like `use integer;` You'd like this code

```
use MyMaths;

my $l = MyMaths->new(1.2);
my $r = MyMaths->new(3.4);

print "A: ", $l + $r, "\n";

use myint;
print "B: ", $l + $r, "\n";

{
    no myint;
    print "C: ", $l + $r, "\n";
}

print "D: ", $l + $r, "\n";

no myint;
print "E: ", $l + $r, "\n";
```

to give the output

```
A: 4.6
B: 4
C: 4.6
D: 4
E: 4.6
```

i.e., where `use myint;` is in effect, addition operations are forced to integer, whereas by default they are not, with the default behaviour being restored via `no myint;`

The minimal implementation of the package `MyMaths` would be something like this:

```
package MyMaths;
use warnings;
use strict;
use myint();
use overload ’+’ => sub {
    my ($l, $r) = @_;
    # Pass 1 to check up one call level from here
    if (myint::in_effect(1)) {
        int($$l) + int($$r);
    } else {
        $$l + $$r;
    }
};

sub new {
    my ($class, $value) = @_;
    bless \$value, $class;
}

1;
```

Note how we load the user pragma `myint` with an empty list `()` to prevent its `import` being called.

The interaction with the Perl compilation happens inside package `myint`:

```
package myint;

use strict;
use warnings;

sub import {
    $^H{"myint/in_effect"} = 1;
}

sub unimport {
    $^H{"myint/in_effect"} = 0;
}

sub in_effect {
    my $level = shift // 0;
    my $hinthash = (caller($level))[10];
    return $hinthash->{"myint/in_effect"};
}

1;
```

As pragmata are implemented as modules, like any other module, `use myint;` becomes

```
BEGIN {
    require myint;
```

```
        myint->import();
    }
```

and `no myint;` is

```
    BEGIN {
        require myint;
        myint->unimport();
    }
```

Hence the `import` and `unimport` routines are called at **compile time** for the user's code.

User pragmata store their state by writing to the magical hash `%^H`, hence these two routines manipulate it. The state information in `%^H` is stored in the optree, and can be retrieved read-only at runtime with `caller()`, at index 10 of the list of returned results. In the example pragma, retrieval is encapsulated into the routine `in_effect()`, which takes as parameter the number of call frames to go up to find the value of the pragma in the user's script. This uses `caller()` to determine the value of `$^H{"myint/in_effect"}` when each line of the user's script was called, and therefore provide the correct semantics in the subroutine implementing the overloaded addition.

57.4 Key naming

There is only a single `%^H`, but arbitrarily many modules that want to use its scoping semantics. To avoid stepping on each other's toes, they need to be sure to use different keys in the hash. It is therefore conventional for a module to use only keys that begin with the module's name (the name of its main package) and a `"/"` character. After this module-identifying prefix, the rest of the key is entirely up to the module: it may include any characters whatsoever. For example, a module `Foo::Bar` should use keys such as `Foo::Bar/baz` and `Foo::Bar/$%/_!`. Modules following this convention all play nicely with each other.

The Perl core uses a handful of keys in `%^H` which do not follow this convention, because they predate it. Keys that follow the convention won't conflict with the core's historical keys.

57.5 Implementation details

The optree is shared between threads. This means there is a possibility that the optree will outlive the particular thread (and therefore the interpreter instance) that created it, so true Perl scalars cannot be stored in the optree. Instead a compact form is used, which can only store values that are integers (signed and unsigned), strings or `undef` - references and floating point values are stringified. If you need to store multiple values or complex structures, you should serialise them, for example with `pack`. The deletion of a hash key from `%^H` is recorded, and as ever can be distinguished from the existence of a key with value `undef` with `exists`.

Don't attempt to store references to data structures as integers which are retrieved via `caller` and converted back, as this will not be threadsafe. Accesses would be to the structure without locking (which is not safe for Perl's scalars), and either the structure has to leak, or it has to be freed when its creating thread terminates, which may be before the optree referencing it is deleted, if other threads outlive it.

58 perlre

58.1 NAME

perlre - Perl regular expressions

58.2 DESCRIPTION

This page describes the syntax of regular expressions in Perl.

If you haven't used regular expressions before, a quick-start introduction is available in Section 66.1 [perlrequick NAME], page 1116, and a longer tutorial introduction is available in Section 68.1 [perlretut NAME], page 1131.

For reference on how regular expressions are used in matching operations, plus various examples of the same, see discussions of `m//`, `s///`, `qr//` and `??` in Section 48.2.30 [perlop Regexp Quote-Like Operators], page 823.

New in v5.22, Section "'strict' mode" in `re` applies stricter rules than otherwise when compiling regular expression patterns. It can find things that, while legal, may not be what you intended.

58.2.1 Modifiers

Matching operations can have various modifiers. Modifiers that relate to the interpretation of the regular expression inside are listed below. Modifiers that alter the way a regular expression is used by Perl are detailed in Section 48.2.30 [perlop Regexp Quote-Like Operators], page 823 and Section 48.2.32 [perlop Gory details of parsing quoted constructs], page 839.

m

Treat string as multiple lines. That is, change `"^"` and `"$"` from matching the start of the string's first line and the end of its last line to matching the start and end of each line within the string.

s

Treat string as single line. That is, change `"."` to match any character whatsoever, even a newline, which normally it would not match.

Used together, as `/ms`, they let the `"."` match any character whatsoever, while still allowing `"^"` and `"$"` to match, respectively, just after and just before newlines within the string.

i

Do case-insensitive pattern matching.

If locale matching rules are in effect, the case map is taken from the current locale for code points less than 255, and from Unicode rules for larger code points. However, matches that would cross the Unicode rules/non-Unicode rules boundary (ords 255/256) will not succeed. See Section 38.1 [perllocale NAME], page 701.

There are a number of Unicode characters that match multiple characters under `/i`. For example, `LATIN SMALL LIGATURE FI` should match the sequence `fi`.

Perl is not currently able to do this when the multiple characters are in the pattern and are split between groupings, or when one or more are quantified. Thus

```
"\N{LATIN SMALL LIGATURE FI}" =~ /fi/i;          # Matches
"\N{LATIN SMALL LIGATURE FI}" =~ /[fi][fi]/i;    # Doesn't match!
"\N{LATIN SMALL LIGATURE FI}" =~ /fi*/i;         # Doesn't match!

# The below doesn't match, and it isn't clear what $1 and $2 would
# be even if it did!!
"\N{LATIN SMALL LIGATURE FI}" =~ /(f)(i)/i;      # Doesn't match!
```

Perl doesn't match multiple characters in a bracketed character class unless the character that maps to them is explicitly mentioned, and it doesn't match them at all if the character class is inverted, which otherwise could be highly confusing. See Section 61.2.3 [perlrecharclass Bracketed Character Classes], page 1065, and Section 61.2.3.3 [perlrecharclass Negation], page 1068.

x

Extend your pattern's legibility by permitting whitespace and comments. Details in Section 58.2.1.1 [/x], page 991

p

Preserve the string matched such that ${^PREMATCH}, ${^MATCH}, and ${^POSTMATCH} are available for use after matching.

In Perl 5.20 and higher this is ignored. Due to a new copy-on-write mechanism, ${^PREMATCH}, ${^MATCH}, and ${^POSTMATCH} will be available after the match regardless of the modifier.

a, d, l and u

These modifiers, all new in 5.14, affect which character-set rules (Unicode, etc.) are used, as described below in Section 58.2.1.2 [Character set modifiers], page 992.

n

Prevent the grouping metacharacters `()` from capturing. This modifier, new in 5.22, will stop `$1`, `$2`, etc... from being filled in.

```
"hello" =~ /(hi|hello)/;   # $1 is "hello"
"hello" =~ /(hi|hello)/n;  # $1 is undef
```

This is equivalent to putting `?:` at the beginning of every capturing group:

```
"hello" =~ /(?:hi|hello)/; # $1 is undef
```

`/n` can be negated on a per-group basis. Alternatively, named captures may still be used.

```
"hello" =~ /(?-n:(hi|hello))/n;   # $1 is "hello"
"hello" =~ /(?<greet>hi|hello)/n; # $1 is "hello", $+{greet} is
                                  # "hello"
```

Other Modifiers

There are a number of flags that can be found at the end of regular expression constructs that are *not* generic regular expression flags, but apply to the operation being performed, like matching or substitution (`m//` or `s///` respectively).

Flags described further in Section 68.3.15 [perlretut Using regular expressions in Perl], page 1153 are:

```
  c  - keep the current position during repeated matching
  g  - globally match the pattern repeatedly in the string
```

Substitution-specific modifiers described in [perlop s/PATTERN/REPLACEMENT/msixpodualngcer], page 830 are:

```
  e  - evaluate the right-hand side as an expression
  ee - evaluate the right side as a string then eval the result
  o  - pretend to optimize your code, but actually introduce bugs
  r  - perform non-destructive substitution and return the new value
```

Regular expression modifiers are usually written in documentation as e.g., "the `/x` modifier", even though the delimiter in question might not really be a slash. The modifiers `/imsxadlup` may also be embedded within the regular expression itself using the `(?...)` construct, see Section 58.2.4 [Extended Patterns], page 1004 below.

58.2.1.1 /x

`/x` tells the regular expression parser to ignore most whitespace that is neither backslashed nor within a bracketed character class. You can use this to break up your regular expression into (slightly) more readable parts. Also, the `#` character is treated as a metacharacter introducing a comment that runs up to the pattern's closing delimiter, or to the end of the current line if the pattern extends onto the next line. Hence, this is very much like an ordinary Perl code comment. (You can include the closing delimiter within the comment only if you precede it with a backslash, so be careful!)

Use of `/x` means that if you want real whitespace or `#` characters in the pattern (outside a bracketed character class, which is unaffected by `/x`), then you'll either have to escape them (using backslashes or `\Q...\E`) or encode them using octal, hex, or `\N{}` escapes. It is ineffective to try to continue a comment onto the next line by escaping the `\n` with a backslash or `\Q`.

You can use [(?#text)], page 1005 to create a comment that ends earlier than the end of the current line, but `text` also can't contain the closing delimiter unless escaped with a backslash.

Taken together, these features go a long way towards making Perl's regular expressions more readable. Here's an example:

```
    # Delete (most) C comments.
    $program =~ s {
        /\*     # Match the opening delimiter.
        .*?     # Match a minimal number of characters.
        \*/     # Match the closing delimiter.
    } []gsx;
```

Note that anything inside a `\Q...\E` stays unaffected by `/x`. And note that `/x` doesn't affect space interpretation within a single multi-character construct. For example in `\x{...}`, regardless of the `/x` modifier, there can be no spaces. Same for a Section 58.2.2.2 [quantifier], page 997 such as `{3}` or `{5,}`. Similarly, `(?:...)` can't have a space between the `(`, `?`, and `:`. Within any delimiters for such a construct, allowed spaces are not affected by `/x`, and depend on the construct. For example, `\x{...}` can't have spaces because hexadecimal numbers don't have spaces in them. But, Unicode properties can have spaces, so in `\p{...}` there can be spaces that follow the Unicode rules, for which see Section "Properties accessible through \p{} and \P{}" in `perluniprops`.

The set of characters that are deemed whitespace are those that Unicode calls "Pattern White Space", namely:

```
U+0009 CHARACTER TABULATION
U+000A LINE FEED
U+000B LINE TABULATION
U+000C FORM FEED
U+000D CARRIAGE RETURN
U+0020 SPACE
U+0085 NEXT LINE
U+200E LEFT-TO-RIGHT MARK
U+200F RIGHT-TO-LEFT MARK
U+2028 LINE SEPARATOR
U+2029 PARAGRAPH SEPARATOR
```

58.2.1.2 Character set modifiers

`/d`, `/u`, `/a`, and `/l`, available starting in 5.14, are called the character set modifiers; they affect the character set rules used for the regular expression.

The `/d`, `/u`, and `/l` modifiers are not likely to be of much use to you, and so you need not worry about them very much. They exist for Perl's internal use, so that complex regular expression data structures can be automatically serialized and later exactly reconstituted, including all their nuances. But, since Perl can't keep a secret, and there may be rare instances where they are useful, they are documented here.

The `/a` modifier, on the other hand, may be useful. Its purpose is to allow code that is to work mostly on ASCII data to not have to concern itself with Unicode.

Briefly, `/l` sets the character set to that of whatever **L**ocale is in effect at the time of the execution of the pattern match.

`/u` sets the character set to **U**nicode.

`/a` also sets the character set to Unicode, BUT adds several restrictions for **A**SCII-safe matching.

`/d` is the old, problematic, pre-5.14 **D**efault character set behavior. Its only use is to force that old behavior.

At any given time, exactly one of these modifiers is in effect. Their existence allows Perl to keep the originally compiled behavior of a regular expression, regardless of what rules are in effect when it is actually executed. And if it is interpolated into a larger regex, the original's rules continue to apply to it, and only it.

The `/l` and `/u` modifiers are automatically selected for regular expressions compiled within the scope of various pragmas, and we recommend that in general, you use those pragmas instead of specifying these modifiers explicitly. For one thing, the modifiers affect only pattern matching, and do not extend to even any replacement done, whereas using the pragmas give consistent results for all appropriate operations within their scopes. For example,

```
s/foo/\Ubar/il
```

will match "foo" using the locale's rules for case-insensitive matching, but the `/l` does not affect how the `\U` operates. Most likely you want both of them to use locale rules. To do this, instead compile the regular expression within the scope of `use locale`. This both implicitly adds the `/l`, and applies locale rules to the `\U`. The lesson is to `use locale`, and not `/l` explicitly.

Similarly, it would be better to use `use feature 'unicode_strings'` instead of,

```
s/foo/\Lbar/iu
```

to get Unicode rules, as the `\L` in the former (but not necessarily the latter) would also use Unicode rules.

More detail on each of the modifiers follows. Most likely you don't need to know this detail for `/l`, `/u`, and `/d`, and can skip ahead to Section 58.2.1.6 [/a], page 995.

58.2.1.3 /l

means to use the current locale's rules (see Section 38.1 [perllocale NAME], page 701) when pattern matching. For example, `\w` will match the "word" characters of that locale, and "/i" case-insensitive matching will match according to the locale's case folding rules. The locale used will be the one in effect at the time of execution of the pattern match. This may not be the same as the compilation-time locale, and can differ from one match to another if there is an intervening call of the Section 38.5.2 [setlocale() function], page 705.

The only non-single-byte locale Perl supports is (starting in v5.20) UTF-8. This means that code points above 255 are treated as Unicode no matter what locale is in effect (since UTF-8 implies Unicode).

Under Unicode rules, there are a few case-insensitive matches that cross the 255/256 boundary. Except for UTF-8 locales in Perls v5.20 and later, these are disallowed under `/l`. For example, 0xFF (on ASCII platforms) does not caselessly match the character at 0x178, `LATIN CAPITAL LETTER Y WITH DIAERESIS`, because 0xFF may not be `LATIN SMALL LETTER Y WITH DIAERESIS` in the current locale, and Perl has no way of knowing if that character even exists in the locale, much less what code point it is.

In a UTF-8 locale in v5.20 and later, the only visible difference between locale and non-locale in regular expressions should be tainting (see Section 70.1 [perlsec NAME], page 1198).

This modifier may be specified to be the default by `use locale`, but see Section 58.2.1.7 [Which character set modifier is in effect?], page 996.

58.2.1.4 /u

means to use Unicode rules when pattern matching. On ASCII platforms, this means that the code points between 128 and 255 take on their Latin-1 (ISO-8859-1) meanings (which are the same as Unicode's). (Otherwise Perl considers their meanings to be undefined.) Thus, under this modifier, the ASCII platform effectively becomes a Unicode platform; and hence, for example, `\w` will match any of the more than 100_000 word characters in Unicode.

Unlike most locales, which are specific to a language and country pair, Unicode classifies all the characters that are letters *somewhere* in the world as `\w`. For example, your locale might not think that `LATIN SMALL LETTER ETH` is a letter (unless you happen to speak Icelandic), but Unicode does. Similarly, all the characters that are decimal digits somewhere in the world will match `\d`; this is hundreds, not 10, possible matches. And some of those digits look like some of the 10 ASCII digits, but mean a different number, so a human could easily think a number is a different quantity than it really is. For example, `BENGALI DIGIT FOUR` (U+09EA) looks very much like an `ASCII DIGIT EIGHT` (U+0038). And, `\d+`, may match strings of digits that are a mixture from different writing systems, creating a security issue. Section "num()" in `Unicode-UCD` can be used to sort this out. Or the `/a` modifier can be used to force `\d` to match just the ASCII 0 through 9.

Also, under this modifier, case-insensitive matching works on the full set of Unicode characters. The `KELVIN SIGN`, for example matches the letters "k" and "K"; and `LATIN SMALL LIGATURE FF` matches the sequence "ff", which, if you're not prepared, might make it look like a hexadecimal constant, presenting another potential security issue. See `http://unicode.org/reports/tr36` for a detailed discussion of Unicode security issues.

This modifier may be specified to be the default by `use feature 'unicode_strings`, `use locale ':not_characters'`, or `[use 5.012], page 479` (or higher), but see Section 58.2.1.7 [Which character set modifier is in effect?], page 996.

58.2.1.5 /d

This modifier means to use the "Default" native rules of the platform except when there is cause to use Unicode rules instead, as follows:

1. the target string is encoded in UTF-8; or
2. the pattern is encoded in UTF-8; or
3. the pattern explicitly mentions a code point that is above 255 (say by `\x{100}`); or
4. the pattern uses a Unicode name (`\N{...}`); or
5. the pattern uses a Unicode property (`\p{...}` or `\P{...}`); or
6. the pattern uses a Unicode break (`\b{...}` or `\B{...}`); or
7. the pattern uses [(?[ ])], page 1017

Another mnemonic for this modifier is "Depends", as the rules actually used depend on various things, and as a result you can get unexpected results. See Section 81.2.17 [perlunicode The "Unicode Bug"], page 1340. The Unicode Bug has become rather infamous, leading to yet another (printable) name for this modifier, "Dodgy".

Unless the pattern or string are encoded in UTF-8, only ASCII characters can match positively.

Here are some examples of how that works on an ASCII platform:

```
$str =  "\xDF";      # $str is not in UTF-8 format.
$str =~ /^\w/;       # No match, as $str isn't in UTF-8 format.
$str .= "\x{0e0b}";  # Now $str is in UTF-8 format.
$str =~ /^\w/;       # Match! $str is now in UTF-8 format.
chop $str;
$str =~ /^\w/;       # Still a match! $str remains in UTF-8 format.
```

This modifier is automatically selected by default when none of the others are, so yet another name for it is "Default".

Because of the unexpected behaviors associated with this modifier, you probably should only use it to maintain weird backward compatibilities.

58.2.1.6 /a (and /aa)

This modifier stands for ASCII-restrict (or ASCII-safe). This modifier, unlike the others, may be doubled-up to increase its effect.

When it appears singly, it causes the sequences `\d`, `\s`, `\w`, and the Posix character classes to match only in the ASCII range. They thus revert to their pre-5.6, pre-Unicode meanings. Under `/a`, `\d` always means precisely the digits `"0"` to `"9"`; `\s` means the five characters `[ \f\n\r\t]`, and starting in Perl v5.18, the vertical tab; `\w` means the 63 characters `[A-Za-z0-9_]`; and likewise, all the Posix classes such as `[[:print:]]` match only the appropriate ASCII-range characters.

This modifier is useful for people who only incidentally use Unicode, and who do not wish to be burdened with its complexities and security concerns.

With `/a`, one can write `\d` with confidence that it will only match ASCII characters, and should the need arise to match beyond ASCII, you can instead use `\p{Digit}` (or `\p{Word}` for `\w`). There are similar `\p{...}` constructs that can match beyond ASCII both white space (see Section 61.2.2.4 [perlrecharclass Whitespace], page 1062), and Posix classes (see Section 61.2.3.5 [perlrecharclass POSIX Character Classes], page 1069). Thus, this modifier doesn't mean you can't use Unicode, it means that to get Unicode matching you must explicitly use a construct (`\p{}`, `\P{}`) that signals Unicode.

As you would expect, this modifier causes, for example, `\D` to mean the same thing as `[^0-9]`; in fact, all non-ASCII characters match `\D`, `\S`, and `\W`. `\b` still means to match at the boundary between `\w` and `\W`, using the `/a` definitions of them (similarly for `\B`).

Otherwise, `/a` behaves like the `/u` modifier, in that case-insensitive matching uses Unicode rules; for example, `"k"` will match the Unicode `\N{KELVIN SIGN}` under `/i` matching, and code points in the Latin1 range, above ASCII will have Unicode rules when it comes to case-insensitive matching.

To forbid ASCII/non-ASCII matches (like `"k"` with `\N{KELVIN SIGN}`), specify the `"a"` twice, for example `/aai` or `/aia`. (The first occurrence of `"a"` restricts the `\d`, etc., and the second occurrence adds the `/i` restrictions.) But, note that code points outside the ASCII range will use Unicode rules for `/i` matching, so the modifier doesn't really restrict things to just ASCII; it just forbids the intermixing of ASCII and non-ASCII.

To summarize, this modifier provides protection for applications that don't wish to be exposed to all of Unicode. Specifying it twice gives added protection.

This modifier may be specified to be the default by `use re '/a'` or `use re '/aa'`. If you do so, you may actually have occasion to use the `/u` modifier explicitly if there are a few regular expressions where you do want full Unicode rules (but even here, it's best if everything were under feature `"unicode_strings"`, along with the `use re '/aa'`). Also see Section 58.2.1.7 [Which character set modifier is in effect?], page 996.

58.2.1.7 Which character set modifier is in effect?

Which of these modifiers is in effect at any given point in a regular expression depends on a fairly complex set of interactions. These have been designed so that in general you don't have to worry about it, but this section gives the gory details. As explained below in Section 58.2.4 [Extended Patterns], page 1004 it is possible to explicitly specify modifiers that apply only to portions of a regular expression. The innermost always has priority over any outer ones, and one applying to the whole expression has priority over any of the default settings that are described in the remainder of this section.

The `Section ‘‘’/flags’ mode’’ in re` pragma can be used to set default modifiers (including these) for regular expressions compiled within its scope. This pragma has precedence over the other pragmas listed below that also change the defaults.

Otherwise, `Section 38.1 [use locale], page 701` sets the default modifier to `/l`; and `feature`, or `[use 5.012], page 479` (or higher) set the default to `/u` when not in the same scope as either `Section 38.1 [use locale], page 701` or `bytes`. (`Section 38.10 [use locale ’:not_characters’], page 721` also sets the default to `/u`, overriding any plain `use locale`.) Unlike the mechanisms mentioned above, these affect operations besides regular expressions pattern matching, and so give more consistent results with other operators, including using `\U`, `\l`, etc. in substitution replacements.

If none of the above apply, for backwards compatibility reasons, the `/d` modifier is the one in effect by default. As this can lead to unexpected results, it is best to specify which other rule set should be used.

58.2.1.8 Character set modifier behavior prior to Perl 5.14

Prior to 5.14, there were no explicit modifiers, but `/l` was implied for regexes compiled within the scope of `use locale`, and `/d` was implied otherwise. However, interpolating a regex into a larger regex would ignore the original compilation in favor of whatever was in effect at the time of the second compilation. There were a number of inconsistencies (bugs) with the `/d` modifier, where Unicode rules would be used when inappropriate, and vice versa. `\p{}` did not imply Unicode rules, and neither did all occurrences of `\N{}`, until 5.12.

58.2.2 Regular Expressions

58.2.2.1 Metacharacters

The patterns used in Perl pattern matching evolved from those supplied in the Version 8 regex routines. (The routines are derived (distantly) from Henry Spencer's freely redistributable reimplementation of the V8 routines.) See Section 58.2.7 [Version 8 Regular Expressions], page 1024 for details.

In particular the following metacharacters have their standard *egrep*-ish meanings:

```
\        Quote the next metacharacter
^        Match the beginning of the line
.        Match any character (except newline)
$        Match the end of the string (or before newline at the end
         of the string)
|        Alternation
```

```
()        Grouping
[]        Bracketed Character class
```

By default, the "^" character is guaranteed to match only the beginning of the string, the "$" character only the end (or before the newline at the end), and Perl does certain optimizations with the assumption that the string contains only one line. Embedded newlines will not be matched by "^" or "$". You may, however, wish to treat a string as a multi-line buffer, such that the "^" will match after any newline within the string (except if the newline is the last character in the string), and "$" will match before any newline. At the cost of a little more overhead, you can do this by using the /m modifier on the pattern match operator. (Older programs did this by setting $*, but this option was removed in perl 5.10.)

To simplify multi-line substitutions, the "." character never matches a newline unless you use the /s modifier, which in effect tells Perl to pretend the string is a single line–even if it isn't.

58.2.2.2 Quantifiers

The following standard quantifiers are recognized:

```
*           Match 0 or more times
+           Match 1 or more times
?           Match 1 or 0 times
{n}         Match exactly n times
{n,}        Match at least n times
{n,m}       Match at least n but not more than m times
```

(If a curly bracket occurs in any other context and does not form part of a backslashed sequence like \x{...}, it is treated as a regular character. However, a deprecation warning is raised for all such occurrences, and in Perl v5.26, literal uses of a curly bracket will be required to be escaped, say by preceding them with a backslash ("\{") or enclosing them within square brackets ("[{]"). This change will allow for future syntax extensions (like making the lower bound of a quantifier optional), and better error checking of quantifiers.)

The "*" quantifier is equivalent to {0,}, the "+" quantifier to {1,}, and the "?" quantifier to {0,1}. n and m are limited to non-negative integral values less than a preset limit defined when perl is built. This is usually 32766 on the most common platforms. The actual limit can be seen in the error message generated by code such as this:

```
$_ **= $_ , / {$_} / for 2 .. 42;
```

By default, a quantified subpattern is "greedy", that is, it will match as many times as possible (given a particular starting location) while still allowing the rest of the pattern to match. If you want it to match the minimum number of times possible, follow the quantifier with a "?". Note that the meanings don't change, just the "greediness":

```
*?        Match 0 or more times, not greedily
+?        Match 1 or more times, not greedily
??        Match 0 or 1 time, not greedily
{n}?      Match exactly n times, not greedily (redundant)
{n,}?     Match at least n times, not greedily
{n,m}?    Match at least n but not more than m times, not greedily
```

Normally when a quantified subpattern does not allow the rest of the overall pattern to match, Perl will backtrack. However, this behaviour is sometimes undesirable. Thus Perl provides the "possessive" quantifier form as well.

```
*+     Match 0 or more times and give nothing back
++     Match 1 or more times and give nothing back
?+     Match 0 or 1 time and give nothing back
{n}+   Match exactly n times and give nothing back (redundant)
{n,}+  Match at least n times and give nothing back
{n,m}+ Match at least n but not more than m times and give nothing back
```

For instance,

```
'aaaa' =~ /a++a/
```

will never match, as the `a++` will gobble up all the `a`'s in the string and won't leave any for the remaining part of the pattern. This feature can be extremely useful to give perl hints about where it shouldn't backtrack. For instance, the typical "match a double-quoted string" problem can be most efficiently performed when written as:

```
/"(?:[^"\\]++|\\.)*+"/
```

as we know that if the final quote does not match, backtracking will not help. See the independent subexpression [(?>pattern)], page 1015 for more details; possessive quantifiers are just syntactic sugar for that construct. For instance the above example could also be written as follows:

```
/"(?>(?:(?>[^"\\]+)|\\.)*)"/
```

Note that the possessive quantifier modifier can not be be combined with the non-greedy modifier. This is because it would make no sense. Consider the follow equivalency table:

```
Illegal         Legal
------------    ------
X??+            X{0}
X+?+            X{1}
X{min,max}?+    X{min}
```

58.2.2.3 Escape sequences

Because patterns are processed as double-quoted strings, the following also work:

```
\t          tab                   (HT, TAB)
\n          newline               (LF, NL)
\r          return                (CR)
\f          form feed             (FF)
\a          alarm (bell)          (BEL)
\e          escape (think troff)  (ESC)
\cK         control char          (example: VT)
\x{}, \x00  character whose ordinal is the given hexadecimal number
\N{name}    named Unicode character or character sequence
\N{U+263D}  Unicode character     (example: FIRST QUARTER MOON)
\o{}, \000  character whose ordinal is the given octal number
\l          lowercase next char (think vi)
\u          uppercase next char (think vi)
```

```
\L          lowercase until \E (think vi)
\U          uppercase until \E (think vi)
\Q          quote (disable) pattern metacharacters until \E
\E          end either case modification or quoted section, think vi
```

Details are in Section 48.2.29 [perlop Quote and Quote-like Operators], page 818.

58.2.2.4 Character Classes and other Special Escapes

In addition, Perl defines the following:

```
Sequence   Note    Description
 [...]     [1]  Match a character according to the rules of the
                  bracketed character class defined by the "...".
                  Example: [a-z] matches "a" or "b" or "c" ... or "z"
 [[:...:]] [2]  Match a character according to the rules of the POSIX
                  character class "..." within the outer bracketed
                  character class.  Example: [[:upper:]] matches any
                  uppercase character.
 (?[...])  [8]  Extended bracketed character class
 \w        [3]  Match a "word" character (alphanumeric plus "_", plus
                  other connector punctuation chars plus Unicode
                  marks)
 \W        [3]  Match a non-"word" character
 \s        [3]  Match a whitespace character
 \S        [3]  Match a non-whitespace character
 \d        [3]  Match a decimal digit character
 \D        [3]  Match a non-digit character
 \pP       [3]  Match P, named property.  Use \p{Prop} for longer names
 \PP       [3]  Match non-P
 \X        [4]  Match Unicode "eXtended grapheme cluster"
 \C             Match a single C-language char (octet) even if that is
                  part of a larger UTF-8 character.  Thus it breaks up
                  characters into their UTF-8 bytes, so you may end up
                  with malformed pieces of UTF-8.  Unsupported in
                  lookbehind. (Deprecated.)
 \1        [5]  Backreference to a specific capture group or buffer.
                  '1' may actually be any positive integer.
 \g1       [5]  Backreference to a specific or previous group,
 \g{-1}    [5]  The number may be negative indicating a relative
                  previous group and may optionally be wrapped in
                  curly brackets for safer parsing.
 \g{name}  [5]  Named backreference
 \k<name>  [5]  Named backreference
 \K        [6]  Keep the stuff left of the \K, don't include it in $&
 \N        [7]  Any character but \n.  Not affected by /s modifier
 \v        [3]  Vertical whitespace
 \V        [3]  Not vertical whitespace
 \h        [3]  Horizontal whitespace
```

```
\H         [3]  Not horizontal whitespace
\R         [4]  Linebreak
```

[1]

See Section 61.2.3 [perlrecharclass Bracketed Character Classes], page 1065 for details.

[2]

See Section 61.2.3.5 [perlrecharclass POSIX Character Classes], page 1069 for details.

[3]

See Section 61.2.2 [perlrecharclass Backslash sequences], page 1059 for details.

[4]

See Section 60.2.8 [perlrebackslash Misc], page 1056 for details.

[5]

See Section 58.2.2.6 [Capture groups], page 1001 below for details.

[6]

See Section 58.2.4 [Extended Patterns], page 1004 below for details.

[7]

Note that `\N` has two meanings. When of the form `\N{NAME}`, it matches the character or character sequence whose name is `NAME`; and similarly when of the form `\N{U+`*hex*`}`, it matches the character whose Unicode code point is *hex*. Otherwise it matches any character but `\n`.

[8]

See Section 61.2.3.9 [perlrecharclass Extended Bracketed Character Classes], page 1073 for details.

58.2.2.5 Assertions

Perl defines the following zero-width assertions:

```
\b{} Match at Unicode boundary of specified type
\B{} Match where corresponding \b{} doesn't match
\b  Match a word boundary
\B  Match except at a word boundary
\A  Match only at beginning of string
\Z  Match only at end of string, or before newline at the end
\z  Match only at end of string
\G  Match only at pos() (e.g. at the end-of-match position
    of prior m//g)
```

A Unicode boundary (`\b{}`), available starting in v5.22, is a spot between two characters, or before the first character in the string, or after the final character in the string where certain criteria defined by Unicode are met. See [perlrebackslash \b{}, \b, \B{}, \B], page 1054 for details.

A word boundary (\b) is a spot between two characters that has a \w on one side of it and a \W on the other side of it (in either order), counting the imaginary characters off the beginning and end of the string as matching a \W. (Within character classes \b represents backspace rather than a word boundary, just as it normally does in any double-quoted string.) The \A and \Z are just like "^" and "$", except that they won't match multiple times when the /m modifier is used, while "^" and "$" will match at every internal line boundary. To match the actual end of the string and not ignore an optional trailing newline, use \z.

The \G assertion can be used to chain global matches (using m//g), as described in Section 48.2.30 [perlop Regexp Quote-Like Operators], page 823. It is also useful when writing lex-like scanners, when you have several patterns that you want to match against consequent substrings of your string; see the previous reference. The actual location where \G will match can also be influenced by using pos() as an lvalue: see [perlfunc pos], page 428. Note that the rule for zero-length matches (see Section 58.2.9 [Repeated Patterns Matching a Zero-length Substring], page 1026) is modified somewhat, in that contents to the left of \G are not counted when determining the length of the match. Thus the following will not match forever:

```
my $string = 'ABC';
pos($string) = 1;
while ($string =~ /(.\G)/g) {
    print $1;
}
```

It will print 'A' and then terminate, as it considers the match to be zero-width, and thus will not match at the same position twice in a row.

It is worth noting that \G improperly used can result in an infinite loop. Take care when using patterns that include \G in an alternation.

Note also that s/// will refuse to overwrite part of a substitution that has already been replaced; so for example this will stop after the first iteration, rather than iterating its way backwards through the string:

```
$_ = "123456789";
pos = 6;
s/.(?=.\G)/X/g;
print;      # prints 1234X6789, not XXXXX6789
```

58.2.2.6 Capture groups

The bracketing construct (...) creates capture groups (also referred to as capture buffers). To refer to the current contents of a group later on, within the same pattern, use \g1 (or \g{1}) for the first, \g2 (or \g{2}) for the second, and so on. This is called a *backreference*.

```
    >>
There is no limit to the number of captured substrings that you may use.
Groups are numbered with the leftmost open parenthesis being number 1, etc.  If
a group did not match, the associated backreference won't match either. (This
can happen if the group is optional, or in a different branch of an
alternation.)
You can omit the C<"g">, and write C<"\1">, etc, but there are some issues with
```

`this form, described below.`

You can also refer to capture groups relatively, by using a negative number, so that `\g-1` and `\g{-1}` both refer to the immediately preceding capture group, and `\g-2` and `\g{-2}` both refer to the group before it. For example:

```
        /
         (Y)            # group 1
         (              # group 2
            (X)         # group 3
            \g{-1}      # backref to group 3
            \g{-3}      # backref to group 1
         )
        /x
```

would match the same as `/(Y) ( (X) \g3 \g1 )/x`. This allows you to interpolate regexes into larger regexes and not have to worry about the capture groups being renumbered.

You can dispense with numbers altogether and create named capture groups. The notation is `(?<`*`name`*`>...)` to declare and `\g{`*`name`*`}` to reference. (To be compatible with .Net regular expressions, `\g{`*`name`*`}` may also be written as `\k{`*`name`*`}`, `\k<`*`name`*`>` or `\k'`*`name`*`'`.) *name* must not begin with a number, nor contain hyphens. When different groups within the same pattern have the same name, any reference to that name assumes the leftmost defined group. Named groups count in absolute and relative numbering, and so can also be referred to by those numbers. (It's possible to do things with named capture groups that would otherwise require `(??{})`.)

Capture group contents are dynamically scoped and available to you outside the pattern until the end of the enclosing block or until the next successful match, whichever comes first. (See Section 74.2.6 [perlsyn Compound Statements], page 1252.) You can refer to them by absolute number (using `"$1"` instead of `"\g1"`, etc); or by name via the `%+` hash, using `"$+{`*`name`*`}"`.

Braces are required in referring to named capture groups, but are optional for absolute or relative numbered ones. Braces are safer when creating a regex by concatenating smaller strings. For example if you have `qr/$a$b/`, and `$a` contained `"\g1"`, and `$b` contained `"37"`, you would get `/\g137/` which is probably not what you intended.

The `\g` and `\k` notations were introduced in Perl 5.10.0. Prior to that there were no named nor relative numbered capture groups. Absolute numbered groups were referred to using `\1`, `\2`, etc., and this notation is still accepted (and likely always will be). But it leads to some ambiguities if there are more than 9 capture groups, as `\10` could mean either the tenth capture group, or the character whose ordinal in octal is 010 (a backspace in ASCII). Perl resolves this ambiguity by interpreting `\10` as a backreference only if at least 10 left parentheses have opened before it. Likewise `\11` is a backreference only if at least 11 left parentheses have opened before it. And so on. `\1` through `\9` are always interpreted as backreferences. There are several examples below that illustrate these perils. You can avoid the ambiguity by always using `\g{}` or `\g` if you mean capturing groups; and for octal constants always using `\o{}`, or for `\077` and below, using 3 digits padded with leading zeros, since a leading zero implies an octal constant.

The `\`*`digit`* notation also works in certain circumstances outside the pattern. See Section 58.2.8 [Warning on \1 Instead of $1], page 1026 below for details.

Examples:

```
s/ˆ([ˆ ]*) *([ˆ ]*)/$2 $1/;     # swap first two words

/(.)\g1/                        # find first doubled char
     and print "’$1’ is the first doubled character\n";

/(?<char>.)\k<char>/            # ... a different way
     and print "’$+{char}’ is the first doubled character\n";

/(?’char’.)\g1/                 # ... mix and match
     and print "’$1’ is the first doubled character\n";

if (/Time: (..):(..):(..)/) {   # parse out values
    $hours = $1;
    $minutes = $2;
    $seconds = $3;
}

/(.)(.)(.)(.)(.)(.)(.)(.)(.)\g10/   # \g10 is a backreference
/(.)(.)(.)(.)(.)(.)(.)(.)(.)\10/    # \10 is octal
/((.)(.)(.)(.)(.)(.)(.)(.)(.))\10/  # \10 is a backreference
/((.)(.)(.)(.)(.)(.)(.)(.)(.))\010/ # \010 is octal

$a = ’(.)\1’;        # Creates problems when concatenated.
$b = ’(.)\g{1}’;     # Avoids the problems.
"aa" =˜ /${a}/;      # True
"aa" =˜ /${b}/;      # True
"aa0" =˜ /${a}0/;    # False!
"aa0" =˜ /${b}0/;    # True
"aa\x08" =˜ /${a}0/;  # True!
"aa\x08" =˜ /${b}0/;  # False
```

Several special variables also refer back to portions of the previous match. `$+` returns whatever the last bracket match matched. `$&` returns the entire matched string. (At one point `$0` did also, but now it returns the name of the program.) `$‘` returns everything before the matched string. `$’` returns everything after the matched string. And `$ˆN` contains whatever was matched by the most-recently closed group (submatch). `$ˆN` can be used in extended patterns (see below), for example to assign a submatch to a variable.

These special variables, like the `%+` hash and the numbered match variables (`$1`, `$2`, `$3`, etc.) are dynamically scoped until the end of the enclosing block or until the next successful match, whichever comes first. (See Section 74.2.6 [perlsyn Compound Statements], page 1252.)

NOTE: Failed matches in Perl do not reset the match variables, which makes it easier to write code that tests for a series of more specific cases and remembers the best match.

WARNING: If your code is to run on Perl 5.16 or earlier, beware that once Perl sees that you need one of `$&`, `$‘`, or `$’` anywhere in the program, it has to provide them for every pattern match. This may substantially slow your program.

Perl uses the same mechanism to produce $1, $2, etc, so you also pay a price for each pattern that contains capturing parentheses. (To avoid this cost while retaining the grouping behaviour, use the extended regular expression (?: ...) instead.) But if you never use $&, $` or $', then patterns *without* capturing parentheses will not be penalized. So avoid $&, $', and $` if you can, but if you can't (and some algorithms really appreciate them), once you've used them once, use them at will, because you've already paid the price.

Perl 5.16 introduced a slightly more efficient mechanism that notes separately whether each of $`, $&, and $' have been seen, and thus may only need to copy part of the string. Perl 5.20 introduced a much more efficient copy-on-write mechanism which eliminates any slowdown.

As another workaround for this problem, Perl 5.10.0 introduced ${^PREMATCH}, ${^MATCH} and ${^POSTMATCH}, which are equivalent to $`, $& and $', **except** that they are only guaranteed to be defined after a successful match that was executed with the /p (preserve) modifier. The use of these variables incurs no global performance penalty, unlike their punctuation char equivalents, however at the trade-off that you have to tell perl when you want to use them. As of Perl 5.20, these three variables are equivalent to $`, $& and $', and /p is ignored.

58.2.3 Quoting metacharacters

Backslashed metacharacters in Perl are alphanumeric, such as \b, \w, \n. Unlike some other regular expression languages, there are no backslashed symbols that aren't alphanumeric. So anything that looks like \\, \(, \), \[, \], \{, or \} is always interpreted as a literal character, not a metacharacter. This was once used in a common idiom to disable or quote the special meanings of regular expression metacharacters in a string that you want to use for a pattern. Simply quote all non-"word" characters:

```
$pattern =~ s/(\W)/\\$1/g;
```

(If use locale is set, then this depends on the current locale.) Today it is more common to use the quotemeta() function or the \Q metaquoting escape sequence to disable all metacharacters' special meanings like this:

```
/$unquoted\Q$quoted\E$unquoted/
```

Beware that if you put literal backslashes (those not inside interpolated variables) between \Q and \E, double-quotish backslash interpolation may lead to confusing results. If you *need* to use literal backslashes within \Q...\E, consult Section 48.2.32 [perlop Gory details of parsing quoted constructs], page 839.

quotemeta() and \Q are fully described in [perlfunc quotemeta], page 430.

58.2.4 Extended Patterns

Perl also defines a consistent extension syntax for features not found in standard tools like **awk** and **lex**. The syntax for most of these is a pair of parentheses with a question mark as the first thing within the parentheses. The character after the question mark indicates the extension.

The stability of these extensions varies widely. Some have been part of the core language for many years. Others are experimental and may change without warning or be completely removed. Check the documentation on an individual feature to verify its current status.

A question mark was chosen for this and for the minimal-matching construct because 1) question marks are rare in older regular expressions, and 2) whenever you see one, you should stop and "question" exactly what is going on. That's psychology....

`(?#text)`

A comment. The text is ignored. Note that Perl closes the comment as soon as it sees a `)`, so there is no way to put a literal `)` in the comment. The pattern's closing delimiter must be escaped by a backslash if it appears in the comment.

See Section 58.2.1.1 [/x], page 991 for another way to have comments in patterns.

`(?adlupimsx-imsx)`
`(?^alupimsx)`

One or more embedded pattern-match modifiers, to be turned on (or turned off, if preceded by `-`) for the remainder of the pattern or the remainder of the enclosing pattern group (if any).

This is particularly useful for dynamic patterns, such as those read in from a configuration file, taken from an argument, or specified in a table somewhere. Consider the case where some patterns want to be case-sensitive and some do not: The case-insensitive ones merely need to include `(?i)` at the front of the pattern. For example:

```
$pattern = "foobar";
if ( /$pattern/i ) { }

# more flexible:

$pattern = "(?i)foobar";
if ( /$pattern/ ) { }
```

These modifiers are restored at the end of the enclosing group. For example,

```
( (?i) blah ) \s+ \g1
```

will match `blah` in any case, some spaces, and an exact (*including the case*!) repetition of the previous word, assuming the `/x` modifier, and no `/i` modifier outside this group.

These modifiers do not carry over into named subpatterns called in the enclosing group. In other words, a pattern such as `((?i)(?&NAME))` does not change the case-sensitivity of the "NAME" pattern.

Any of these modifiers can be set to apply globally to all regular expressions compiled within the scope of a `use re`. See Section “'/flags' mode” in `re`.

Starting in Perl 5.14, a `"^"` (caret or circumflex accent) immediately after the `"?"` is a shorthand equivalent to `d-imsx`. Flags (except `"d"`) may follow the caret to override it. But a minus sign is not legal with it.

Note that the `a`, `d`, `l`, `p`, and `u` modifiers are special in that they can only be enabled, not disabled, and the `a`, `d`, `l`, and `u` modifiers are mutually exclusive: specifying one de-specifies the others, and a maximum of one (or two `a`'s) may appear in the construct. Thus, for example, `(?-p)` will warn when compiled under `use warnings`; `(?-d:...)` and `(?dl:...)` are fatal errors.

Note also that the `p` modifier is special in that its presence anywhere in a pattern has a global effect.

`(?:pattern)`
`(?adluimsx-imsx:pattern)`
`(?^aluimsx:pattern)`

This is for clustering, not capturing; it groups subexpressions like "()", but doesn't make backreferences as "()" does. So

```
@fields = split(/\b(?:a|b|c)\b/)
```

is like

```
@fields = split(/\b(a|b|c)\b/)
```

but doesn't spit out extra fields. It's also cheaper not to capture characters if you don't need to.

Any letters between `?` and `:` act as flags modifiers as with `(?adluimsx-imsx)`. For example,

```
/(?s-i:more.*than).*million/i
```

is equivalent to the more verbose

```
/(?:(?s-i)more.*than).*million/i
```

Starting in Perl 5.14, a "^" (caret or circumflex accent) immediately after the "?" is a shorthand equivalent to `d-imsx`. Any positive flags (except "d") may follow the caret, so

```
(?^x:foo)
```

is equivalent to

```
(?x-ims:foo)
```

The caret tells Perl that this cluster doesn't inherit the flags of any surrounding pattern, but uses the system defaults (`d-imsx`), modified by any flags specified.

The caret allows for simpler stringification of compiled regular expressions. These look like

```
(?^:pattern)
```

with any non-default flags appearing between the caret and the colon. A test that looks at such stringification thus doesn't need to have the system default flags hard-coded in it, just the caret. If new flags are added to Perl, the meaning of the caret's expansion will change to include the default for those flags, so the test will still work, unchanged.

Specifying a negative flag after the caret is an error, as the flag is redundant.

Mnemonic for `(?^...)`: A fresh beginning since the usual use of a caret is to match at the beginning.

`(?|pattern)`

This is the "branch reset" pattern, which has the special property that the capture groups are numbered from the same starting point in each alternation branch. It is available starting from perl 5.10.0.

Capture groups are numbered from left to right, but inside this construct the numbering is restarted for each branch.

The numbering within each branch will be as normal, and any groups following this construct will be numbered as though the construct contained only one branch, that being the one with the most capture groups in it.

This construct is useful when you want to capture one of a number of alternative matches.

Consider the following pattern. The numbers underneath show in which group the captured content will be stored.

```
    # before  ---------------branch-reset----------- after
    / ( a )  (?| x ( y ) z | (p (q) r) | (t) u (v) ) ( z ) /x
    # 1            2         2  3        2     3     4
```

Be careful when using the branch reset pattern in combination with named captures. Named captures are implemented as being aliases to numbered groups holding the captures, and that interferes with the implementation of the branch reset pattern. If you are using named captures in a branch reset pattern, it's best to use the same names, in the same order, in each of the alternations:

```
   /(?|  (?<a> x ) (?<b> y )
      |  (?<a> z ) (?<b> w )) /x
```

Not doing so may lead to surprises:

```
  "12" =~ /(?| (?<a> \d+ ) | (?<b> \D+))/x;
  say $+ {a};   # Prints '12'
  say $+ {b};   # *Also* prints '12'.
```

The problem here is that both the group named `a` and the group named `b` are aliases for the group belonging to `$1`.

Look-Around Assertions

Look-around assertions are zero-width patterns which match a specific pattern without including it in `$&`. Positive assertions match when their subpattern matches, negative assertions match when their subpattern fails. Look-behind matches text up to the current match position, look-ahead matches text following the current match position.

`(?=pattern)`

A zero-width positive look-ahead assertion. For example, `/\w+(?=\t)/` matches a word followed by a tab, without including the tab in `$&`.

`(?!pattern)`

A zero-width negative look-ahead assertion. For example `/foo(?!bar)/` matches any occurrence of "foo" that isn't followed by "bar". Note however that look-ahead and look-behind are NOT the same thing. You cannot use this for look-behind.

If you are looking for a "bar" that isn't preceded by a "foo", `/(?!foo)bar/` will not do what you want. That's because the `(?!foo)` is just saying that the next thing cannot be "foo"–and it's not, it's a "bar", so "foobar" will match. Use look-behind instead (see below).

`(?<=pattern) \K`

A zero-width positive look-behind assertion. For example, `/(?<=\t)\w+/` matches a word that follows a tab, without including the tab in `$&`. Works only for fixed-width look-behind.

There is a special form of this construct, called `\K` (available since Perl 5.10.0), which causes the regex engine to "keep" everything it had matched prior to the `\K` and not include it in `$&`. This effectively provides variable-length look-behind. The use of `\K` inside of another look-around assertion is allowed, but the behaviour is currently not well defined.

For various reasons `\K` may be significantly more efficient than the equivalent `(?<=...)` construct, and it is especially useful in situations where you want to efficiently remove something following something else in a string. For instance

```
  s/(foo)bar/$1/g;
```

can be rewritten as the much more efficient

```
  s/foo\Kbar//g;
```

`(?<!pattern)`

A zero-width negative look-behind assertion. For example `/(?<!bar)foo/` matches any occurrence of "foo" that does not follow "bar". Works only for fixed-width look-behind.

`(?'NAME'pattern)`
`(?<NAME>pattern)`) >>

A named capture group. Identical in every respect to normal capturing parentheses `()` but for the additional fact that the group can be referred to by name in various regular expression constructs (like `\g{NAME}`) and can be accessed by name after a successful match via `%+` or `%-`. See Section 86.1 [perlvar NAME], page 1375 for more details on the `%+` and `%-` hashes.

If multiple distinct capture groups have the same name then the $+{NAME} will refer to the leftmost defined group in the match.

The forms `(?'NAME'pattern)` and `(?<NAME>pattern)` are equivalent.

NOTE: While the notation of this construct is the same as the similar function in .NET regexes, the behavior is not. In Perl the groups are numbered sequentially regardless of being named or not. Thus in the pattern

```
  /(x)(?<foo>y)(z)/
```

$+{foo} will be the same as $2, and $3 will contain 'z' instead of the opposite which is what a .NET regex hacker might expect.

Currently NAME is restricted to simple identifiers only. In other words, it must match `/^[_A-Za-z][_A-Za-z0-9]*\z/` or its Unicode extension (see `utf8`), though it isn't extended by the locale (see Section 38.1 [perllocale NAME], page 701).

NOTE: In order to make things easier for programmers with experience with the Python or PCRE regex engines, the pattern `(?P<NAME>pattern)` may be used instead of `(?<NAME>pattern)`; however this form does not support the use of single quotes as a delimiter for the name.

`\k<NAME>`

`\k'NAME'`

Named backreference. Similar to numeric backreferences, except that the group is designated by name and not number. If multiple groups have the same name then it refers to the leftmost defined group in the current match.

It is an error to refer to a name not defined by a `(?<NAME>)` earlier in the pattern.

Both forms are equivalent.

NOTE: In order to make things easier for programmers with experience with the Python or PCRE regex engines, the pattern `(?P=NAME)` may be used instead of `\k<NAME>`.

`(?{ code })`

WARNING: Using this feature safely requires that you understand its limitations. Code executed that has side effects may not perform identically from version to version due to the effect of future optimisations in the regex engine. For more information on this, see Section 58.2.12 [Embedded Code Execution Frequency], page 1030.

This zero-width assertion executes any embedded Perl code. It always succeeds, and its return value is set as `$^R`.

In literal patterns, the code is parsed at the same time as the surrounding code. While within the pattern, control is passed temporarily back to the perl parser, until the logically-balancing closing brace is encountered. This is similar to the way that an array index expression in a literal string is handled, for example

```
"abc$array[ 1 + f('[') + g()]def"
```

In particular, braces do not need to be balanced:

```
s/abc(?{ f('{'); })/def/
```

Even in a pattern that is interpolated and compiled at run-time, literal code blocks will be compiled once, at perl compile time; the following prints "ABCD":

```
print "D";
my $qr = qr/(?{ BEGIN { print "A" } })/;
my $foo = "foo";
/$foo$qr(?{ BEGIN { print "B" } })/;
BEGIN { print "C" }
```

In patterns where the text of the code is derived from run-time information rather than appearing literally in a source code /pattern/, the code is compiled at the same time that the pattern is compiled, and for reasons of security, `use re 'eval'` must be in scope. This is to stop user-supplied patterns containing code snippets from being executable.

In situations where you need to enable this with `use re 'eval'`, you should also have taint checking enabled. Better yet, use the carefully constrained evaluation within a Safe compartment. See Section 70.1 [perlsec NAME], page 1198 for details about both these mechanisms.

From the viewpoint of parsing, lexical variable scope and closures,

```
    /AAA(?{ BBB })CCC/
```

behaves approximately like

```
    /AAA/ && do { BBB } && /CCC/
```

Similarly,

```
    qr/AAA(?{ BBB })CCC/
```

behaves approximately like

```
    sub { /AAA/ && do { BBB } && /CCC/ }
```

In particular:

```
    { my $i = 1; $r = qr/(?{ print $i })/ }
    my $i = 2;
    /$r/; # prints "1"
```

Inside a `(?{...})` block, `$_` refers to the string the regular expression is matching against. You can also use `pos()` to know what is the current position of matching within this string.

The code block introduces a new scope from the perspective of lexical variable declarations, but **not** from the perspective of `local` and similar localizing behaviours. So later code blocks within the same pattern will still see the values which were localized in earlier blocks. These accumulated localizations are undone either at the end of a successful match, or if the assertion is backtracked (compare Section 58.2.6 [Backtracking], page 1021). For example,

```
  $_ = 'a' x 8;
  m<
     (?{ $cnt = 0 })               # Initialize $cnt.
     (
       a
       (?{
           local $cnt = $cnt + 1;  # Update $cnt,
                                   # backtracking-safe.
       })
     )*
     aaaa
     (?{ $res = $cnt })            # On success copy to
                                   # non-localized location.
   >x;
```

will initially increment `$cnt` up to 8; then during backtracking, its value will be unwound back to 4, which is the value assigned to `$res`. At the end of the regex execution, $cnt will be wound back to its initial value of 0.

This assertion may be used as the condition in a

```
    (?(condition)yes-pattern|no-pattern)
```

switch. If *not* used in this way, the result of evaluation of `code` is put into the special variable `$^R`. This happens immediately, so `$^R` can be used from other `(?{ code })` assertions inside the same regular expression.

The assignment to `$^R` above is properly localized, so the old value of `$^R` is restored if the assertion is backtracked; compare Section 58.2.6 [Backtracking], page 1021.

Note that the special variable `$^N` is particularly useful with code blocks to capture the results of submatches in variables without having to keep track of the number of nested parentheses. For example:

```
$_ = "The brown fox jumps over the lazy dog";
/the (\S+)(?{ $color = $^N }) (\S+)(?{ $animal = $^N })/i;
print "color = $color, animal = $animal\n";
```

`(??{ code })`

WARNING: Using this feature safely requires that you understand its limitations. Code executed that has side effects may not perform identically from version to version due to the effect of future optimisations in the regex engine. For more information on this, see Section 58.2.12 [Embedded Code Execution Frequency], page 1030.

This is a "postponed" regular subexpression. It behaves in *exactly* the same way as a `(?{ code })` code block as described above, except that its return value, rather than being assigned to `$^R`, is treated as a pattern, compiled if it's a string (or used as-is if its a qr// object), then matched as if it were inserted instead of this construct.

During the matching of this sub-pattern, it has its own set of captures which are valid during the sub-match, but are discarded once control returns to the main pattern. For example, the following matches, with the inner pattern capturing "B" and matching "BB", while the outer pattern captures "A";

```
my $inner = ’(.)\1’;
"ABBA" =~ /^(.)(??{ $inner })\1/;
print $1; # prints "A";
```

Note that this means that there is no way for the inner pattern to refer to a capture group defined outside. (The code block itself can use `$1`, etc., to refer to the enclosing pattern's capture groups.) Thus, although

```
(’a’ x 100)=~/(??{’(.)’ x 100})/
```

will match, it will *not* set $1 on exit.

The following pattern matches a parenthesized group:

```
$re = qr{
           \(
           (?:
              (?> [^()]+ )  # Non-parens without backtracking
            |
              (??{ $re })   # Group with matching parens
           )*
           \)
```

```
        }x;
```

See also [(?*PARNO*)], page 1012 for a different, more efficient way to accomplish the same task.

Executing a postponed regular expression 50 times without consuming any input string will result in a fatal error. The maximum depth is compiled into perl, so changing it requires a custom build.

(?*PARNO*) (?-*PARNO*) (?+*PARNO*) (?R) (?0)

Recursive subpattern. Treat the contents of a given capture buffer in the current pattern as an independent subpattern and attempt to match it at the current position in the string. Information about capture state from the caller for things like backreferences is available to the subpattern, but capture buffers set by the subpattern are not visible to the caller.

Similar to (??{ code }) except that it does not involve executing any code or potentially compiling a returned pattern string; instead it treats the part of the current pattern contained within a specified capture group as an independent pattern that must match at the current position. Also different is the treatment of capture buffers, unlike (??{ code }) recursive patterns have access to their callers match state, so one can use backreferences safely.

PARNO is a sequence of digits (not starting with 0) whose value reflects the paren-number of the capture group to recurse to. (?R) recurses to the beginning of the whole pattern. (?0) is an alternate syntax for (?R). If *PARNO* is preceded by a plus or minus sign then it is assumed to be relative, with negative numbers indicating preceding capture groups and positive ones following. Thus (?-1) refers to the most recently declared group, and (?+1) indicates the next group to be declared. Note that the counting for relative recursion differs from that of relative backreferences, in that with recursion unclosed groups **are** included.

The following pattern matches a function foo() which may contain balanced parentheses as the argument.

```
  $re = qr{ (                    # paren group 1 (full function)
              foo
              (                  # paren group 2 (parens)
                \(
                  (              # paren group 3 (contents of parens)
                  (?:
                   (?> [^()]+ ) # Non-parens without backtracking
                  |
                   (?2)          # Recurse to start of paren group 2
                  )*
                  )
                \)
              )
            )
          }x;
```

If the pattern was used as follows

```
’foo(bar(baz)+baz(bop))’=~/$re/
    and print "\$1 = $1\n",
              "\$2 = $2\n",
              "\$3 = $3\n";
```

the output produced should be the following:

```
$1 = foo(bar(baz)+baz(bop))
$2 = (bar(baz)+baz(bop))
$3 = bar(baz)+baz(bop)
```

If there is no corresponding capture group defined, then it is a fatal error. Recursing deeper than 50 times without consuming any input string will also result in a fatal error. The maximum depth is compiled into perl, so changing it requires a custom build.

The following shows how using negative indexing can make it easier to embed recursive patterns inside of a `qr//` construct for later use:

```
my $parens = qr/(\((?:[^()]++|(?-1))*+\))/;
if (/foo $parens \s+ \+ \s+ bar $parens/x) {
   # do something here...
}
```

Note that this pattern does not behave the same way as the equivalent PCRE or Python construct of the same form. In Perl you can backtrack into a recursed group, in PCRE and Python the recursed into group is treated as atomic. Also, modifiers are resolved at compile time, so constructs like (?i:(?1)) or (?:(?i)(?1)) do not affect how the sub-pattern will be processed.

`(?&NAME)`

Recurse to a named subpattern. Identical to `(?`*`PARNO`*`)` except that the parenthesis to recurse to is determined by name. If multiple parentheses have the same name, then it recurses to the leftmost.

It is an error to refer to a name that is not declared somewhere in the pattern.

NOTE: In order to make things easier for programmers with experience with the Python or PCRE regex engines the pattern `(?P>NAME)` may be used instead of `(?&NAME)`.

`(?(condition)yes-pattern|no-pattern)`
`(?(condition)yes-pattern)`

Conditional expression. Matches `yes-pattern` if `condition` yields a true value, matches `no-pattern` otherwise. A missing pattern always matches.

`(condition)` should be one of: 1) an integer in parentheses (which is valid if the corresponding pair of parentheses matched); 2) a look-ahead/look-behind/evaluate zero-width assertion; 3) a name in angle brackets or single quotes (which is valid if a group with the given name matched); or 4) the special symbol (R) (true when evaluated inside of recursion or eval). Additionally the R may be followed by a number, (which will be true when evaluated when recursing inside of the appropriate group), or by `&NAME`, in which case it will be true only when evaluated during recursion in the named group.

Here’s a summary of the possible predicates:

(1) (2) ...

Checks if the numbered capturing group has matched something.

(<NAME>) (’NAME’)

Checks if a group with the given name has matched something.

(?=...) (?!...) (?<=...) (?<!...)

Checks whether the pattern matches (or does not match, for the ’!’ variants).

(?{ CODE })

Treats the return value of the code block as the condition.

(R)

Checks if the expression has been evaluated inside of recursion.

(R1) (R2) ...

Checks if the expression has been evaluated while executing directly inside of the n-th capture group. This check is the regex equivalent of

```
  if ((caller(0))[3] eq ’subname’) { ... }
```

In other words, it does not check the full recursion stack.

(R&NAME)

Similar to `(R1)`, this predicate checks to see if we’re executing directly inside of the leftmost group with a given name (this is the same logic used by `(?&NAME)` to disambiguate). It does not check the full stack, but only the name of the innermost active recursion.

(DEFINE)

In this case, the yes-pattern is never directly executed, and no no-pattern is allowed. Similar in spirit to `(?{0})` but more efficient. See below for details.

For example:

```
    m{ ( \( )?
       [^()]+
       (?(1) \) )
     }x
```

matches a chunk of non-parentheses, possibly included in parentheses themselves.

A special form is the `(DEFINE)` predicate, which never executes its yes-pattern directly, and does not allow a no-pattern. This allows one to define subpatterns which will be executed only by the recursion mechanism. This way, you can define a set of regular expression rules that can be bundled into any pattern you choose.

It is recommended that for this usage you put the DEFINE block at the end of the pattern, and that you name any subpatterns defined within it.

Also, it's worth noting that patterns defined this way probably will not be as efficient, as the optimizer is not very clever about handling them.

An example of how this might be used is as follows:

```
/(?<NAME>(?&NAME_PAT))(?<ADDR>(?&ADDRESS_PAT))
 (?(DEFINE)
   (?<NAME_PAT>....)
   (?<ADDRESS_PAT>....)
 )/x
```

Note that capture groups matched inside of recursion are not accessible after the recursion returns, so the extra layer of capturing groups is necessary. Thus `$+{NAME_PAT}` would not be defined even though `$+{NAME}` would be.

Finally, keep in mind that subpatterns created inside a DEFINE block count towards the absolute and relative number of captures, so this:

```
my @captures = "a" =~ /(.)                  # First capture
                       (?(DEFINE)
                           (?<EXAMPLE> 1 )  # Second capture
                       )/x;
say scalar @captures;
```

Will output 2, not 1. This is particularly important if you intend to compile the definitions with the `qr//` operator, and later interpolate them in another pattern.

`(?>pattern)`

An "independent" subexpression, one which matches the substring that a *standalone* `pattern` would match if anchored at the given position, and it matches *nothing other than this substring*. This construct is useful for optimizations of what would otherwise be "eternal" matches, because it will not backtrack (see Section 58.2.6 [Backtracking], page 1021). It may also be useful in places where the "grab all you can, and do not give anything back" semantic is desirable.

For example: `^(?>a*)ab` will never match, since `(?>a*)` (anchored at the beginning of string, as above) will match *all* characters `a` at the beginning of string, leaving no `a` for `ab` to match. In contrast, `a*ab` will match the same as `a+b`, since the match of the subgroup `a*` is influenced by the following group `ab` (see Section 58.2.6 [Backtracking], page 1021). In particular, `a*` inside `a*ab` will match fewer characters than a standalone `a*`, since this makes the tail match.

`(?>pattern)` does not disable backtracking altogether once it has matched. It is still possible to backtrack past the construct, but not into it. So `((?>a*)|(?>b*))ar` will still match "bar".

An effect similar to `(?>pattern)` may be achieved by writing `(?=(pattern))\g{-1}`. This matches the same substring as a standalone `a+`, and the following `\g{-1}` eats the matched string; it therefore makes a zero-length assertion into an analogue of `(?>...)`. (The difference between these two constructs is that the second one uses a capturing group, thus shifting ordinals of backreferences in the rest of a regular expression.)

Consider this pattern:

```
    m{ \(
          (
            [^()]+           # x+
          |
            \( [^()]* \)
          )+
       \)
     }x
```

That will efficiently match a nonempty group with matching parentheses two levels deep or less. However, if there is no such group, it will take virtually forever on a long string. That's because there are so many different ways to split a long string into several substrings. This is what `(.+)+` is doing, and `(.+)+` is similar to a subpattern of the above pattern. Consider how the pattern above detects no-match on `((()aaaaaaaaaaaaaaaaaa` in several seconds, but that each extra letter doubles this time. This exponential performance will make it appear that your program has hung. However, a tiny change to this pattern

```
    m{ \(
          (
            (?> [^()]+ )        # change x+ above to (?> x+ )
          |
            \( [^()]* \)
          )+
       \)
     }x
```

which uses `(?>...)` matches exactly when the one above does (verifying this yourself would be a productive exercise), but finishes in a fourth the time when used on a similar string with 1000000 `a`s. Be aware, however, that, when this construct is followed by a quantifier, it currently triggers a warning message under the `use warnings` pragma or `-w` switch saying it `"matches null string many times in regex"`.

On simple groups, such as the pattern `(?> [^()]+ )`, a comparable effect may be achieved by negative look-ahead, as in `[^()]+ (?! [^()] )`. This was only 4 times slower on a string with 1000000 `a`s.

The "grab all you can, and do not give anything back" semantic is desirable in many situations where on the first sight a simple `()*` looks like the correct solution. Suppose we parse text with comments being delimited by `#` followed by some optional (horizontal) whitespace. Contrary to its appearance, `#[ \t]*` *is not* the correct subexpression to match the comment delimiter, because it may "give up" some whitespace if the remainder of the pattern can be made to match that way. The correct answer is either one of these:

```
    (?>#[ \t]*)
    #[ \t]*(?![ \t])
```

For example, to grab non-empty comments into $1, one should use either one of these:

```
/ (?> \# [ \t]* ) (        .+ ) /x;
/     \# [ \t]*   ( [^ \t] .* ) /x;
```

Which one you pick depends on which of these expressions better reflects the above specification of comments.

In some literature this construct is called "atomic matching" or "possessive matching".

Possessive quantifiers are equivalent to putting the item they are applied to inside of one of these constructs. The following equivalences apply:

```
Quantifier Form     Bracketing Form
---------------     ---------------
PAT*+               (?>PAT*)
PAT++               (?>PAT+)
PAT?+               (?>PAT?)
PAT{min,max}+       (?>PAT{min,max})
```

`(?[ ])`

See Section 61.2.3.9 [perlrecharclass Extended Bracketed Character Classes], page 1073.

58.2.5 Special Backtracking Control Verbs

These special patterns are generally of the form `(*VERB:ARG)`. Unless otherwise stated the ARG argument is optional; in some cases, it is forbidden.

Any pattern containing a special backtracking verb that allows an argument has the special behaviour that when executed it sets the current package's `$REGERROR` and `$REGMARK` variables. When doing so the following rules apply:

On failure, the `$REGERROR` variable will be set to the ARG value of the verb pattern, if the verb was involved in the failure of the match. If the ARG part of the pattern was omitted, then `$REGERROR` will be set to the name of the last `(*MARK:NAME)` pattern executed, or to TRUE if there was none. Also, the `$REGMARK` variable will be set to FALSE.

On a successful match, the `$REGERROR` variable will be set to FALSE, and the `$REGMARK` variable will be set to the name of the last `(*MARK:NAME)` pattern executed. See the explanation for the `(*MARK:NAME)` verb below for more details.

NOTE: `$REGERROR` and `$REGMARK` are not magic variables like `$1` and most other regex-related variables. They are not local to a scope, nor readonly, but instead are volatile package variables similar to `$AUTOLOAD`. Use `local` to localize changes to them to a specific scope if necessary.

If a pattern does not contain a special backtracking verb that allows an argument, then `$REGERROR` and `$REGMARK` are not touched at all.

Verbs that take an argument

`(*PRUNE) (*PRUNE:NAME)`

This zero-width pattern prunes the backtracking tree at the current point when backtracked into on failure. Consider the pattern `A (*PRUNE) B`, where A and B are complex patterns. Until the `(*PRUNE)` verb is reached, A may backtrack as necessary to match.

Once it is reached, matching continues in B, which may also backtrack as necessary; however, should B not match, then no further backtracking will take place, and the pattern will fail outright at the current starting position.

The following example counts all the possible matching strings in a pattern (without actually matching any of them).

```
'aaab' =~ /a+b?(?{print "$&\n"; $count++})(*FAIL)/;
print "Count=$count\n";
```

which produces:

```
aaab
aaa
aa
a
aab
aa
a
ab
a
Count=9
```

If we add a (*PRUNE) before the count like the following

```
'aaab' =~ /a+b?(*PRUNE)(?{print "$&\n"; $count++})(*FAIL)/;
print "Count=$count\n";
```

we prevent backtracking and find the count of the longest matching string at each matching starting point like so:

```
aaab
aab
ab
Count=3
```

Any number of (*PRUNE) assertions may be used in a pattern.

See also (?>pattern) and possessive quantifiers for other ways to control backtracking. In some cases, the use of (*PRUNE) can be replaced with a (?>pattern) with no functional difference; however, (*PRUNE) can be used to handle cases that cannot be expressed using a (?>pattern) alone.

(*SKIP) (*SKIP:NAME)

This zero-width pattern is similar to (*PRUNE), except that on failure it also signifies that whatever text that was matched leading up to the (*SKIP) pattern being executed cannot be part of *any* match of this pattern. This effectively means that the regex engine "skips" forward to this position on failure and tries to match again, (assuming that there is sufficient room to match).

The name of the (*SKIP:NAME) pattern has special significance. If a (*MARK:NAME) was encountered while matching, then it is that position which is used as the "skip point". If no (*MARK) of that name was encountered, then the (*SKIP) operator has no effect. When used without a name the "skip point" is where the match point was when executing the (*SKIP) pattern.

Compare the following to the examples in (*PRUNE); note the string is twice as long:

```
 'aaabaaab' =~ /a+b?(*SKIP)(?{print "$&\n"; $count++})(*FAIL)/;
 print "Count=$count\n";
```

outputs

```
    aaab
    aaab
    Count=2
```

Once the 'aaab' at the start of the string has matched, and the (*SKIP) executed, the next starting point will be where the cursor was when the (*SKIP) was executed.

(*MARK:NAME) (*:NAME)

This zero-width pattern can be used to mark the point reached in a string when a certain part of the pattern has been successfully matched. This mark may be given a name. A later (*SKIP) pattern will then skip forward to that point if backtracked into on failure. Any number of (*MARK) patterns are allowed, and the NAME portion may be duplicated.

In addition to interacting with the (*SKIP) pattern, (*MARK:NAME) can be used to "label" a pattern branch, so that after matching, the program can determine which branches of the pattern were involved in the match.

When a match is successful, the $REGMARK variable will be set to the name of the most recently executed (*MARK:NAME) that was involved in the match.

This can be used to determine which branch of a pattern was matched without using a separate capture group for each branch, which in turn can result in a performance improvement, as perl cannot optimize /(?:(x)|(y)|(z))/ as efficiently as something like /(?:x(*MARK:x)|y(*MARK:y)|z(*MARK:z))/.

When a match has failed, and unless another verb has been involved in failing the match and has provided its own name to use, the $REGERROR variable will be set to the name of the most recently executed (*MARK:NAME).

See ⟨undefined⟩ [(*SKIP)], page ⟨undefined⟩ for more details.

As a shortcut (*MARK:NAME) can be written (*:NAME).

(*THEN) (*THEN:NAME)

This is similar to the "cut group" operator :: from Perl 6. Like (*PRUNE), this verb always matches, and when backtracked into on failure, it causes the regex engine to try the next alternation in the innermost enclosing group (capturing or otherwise) that has alternations. The two branches of a `(?(condition)yes-pattern|no-pattern)` do not count as an alternation, as far as `(*THEN)` is concerned.

Its name comes from the observation that this operation combined with the alternation operator (`|`) can be used to create what is essentially a pattern-based if/then/else block:

```
( COND (*THEN) FOO | COND2 (*THEN) BAR | COND3 (*THEN) BAZ )
```

Note that if this operator is used and NOT inside of an alternation then it acts exactly like the `(*PRUNE)` operator.

```
/ A (*PRUNE) B /
```

is the same as

```
/ A (*THEN) B /
```

but

```
/ ( A (*THEN) B | C ) /
```

is not the same as

```
/ ( A (*PRUNE) B | C ) /
```

as after matching the A but failing on the B the `(*THEN)` verb will backtrack and try C; but the `(*PRUNE)` verb will simply fail.

Verbs without an argument

`(*COMMIT)`

This is the Perl 6 "commit pattern" `<commit>` or `:::`. It's a zero-width pattern similar to `(*SKIP)`, except that when backtracked into on failure it causes the match to fail outright. No further attempts to find a valid match by advancing the start pointer will occur again. For example,

```
 'aaabaaab' =~ /a+b?(*COMMIT)(?{print "$&\n"; $count++})(*FAIL)/;
 print "Count=$count\n";
```

outputs

```
    aaab
    Count=1
```

In other words, once the `(*COMMIT)` has been entered, and if the pattern does not match, the regex engine will not try any further matching on the rest of the string.

`(*FAIL) (*F)`

This pattern matches nothing and always fails. It can be used to force the engine to backtrack. It is equivalent to `(?!)`, but easier to read. In fact, `(?!)` gets optimised into `(*FAIL)` internally.

It is probably useful only when combined with `(?{})` or `(??{})`.

`(*ACCEPT)`

This pattern matches nothing and causes the end of successful matching at the point at which the `(*ACCEPT)` pattern was encountered, regardless of whether there is actually more to match in the string. When inside of a nested pattern, such as recursion, or in a subpattern dynamically generated via (??{}), only the innermost pattern is ended immediately.

If the (*ACCEPT) is inside of capturing groups then the groups are marked as ended at the point at which the (*ACCEPT) was encountered. For instance:

```
'AB' =~ /(A (A|B(*ACCEPT)|C) D)(E)/x;
```

will match, and $1 will be AB and $2 will be B, $3 will not be set. If another branch in the inner parentheses was matched, such as in the string 'ACDE', then the D and E would have to be matched as well.

58.2.6 Backtracking

NOTE: This section presents an abstract approximation of regular expression behavior. For a more rigorous (and complicated) view of the rules involved in selecting a match among possible alternatives, see Section 58.2.10 [Combining RE Pieces], page 1027.

A fundamental feature of regular expression matching involves the notion called *backtracking*, which is currently used (when needed) by all regular non-possessive expression quantifiers, namely *, *?, +, +?, {n,m}, and {n,m}?. Backtracking is often optimized internally, but the general principle outlined here is valid.

For a regular expression to match, the *entire* regular expression must match, not just part of it. So if the beginning of a pattern containing a quantifier succeeds in a way that causes later parts in the pattern to fail, the matching engine backs up and recalculates the beginning part–that's why it's called backtracking.

Here is an example of backtracking: Let's say you want to find the word following "foo" in the string "Food is on the foo table.":

```
$_ = "Food is on the foo table.";
if ( /\b(foo)\s+(\w+)/i ) {
    print "$2 follows $1.\n";
}
```

When the match runs, the first part of the regular expression (\b(foo)) finds a possible match right at the beginning of the string, and loads up $1 with "Foo". However, as soon as the matching engine sees that there's no whitespace following the "Foo" that it had saved in $1, it realizes its mistake and starts over again one character after where it had the tentative match. This time it goes all the way until the next occurrence of "foo". The complete regular expression matches this time, and you get the expected output of "table follows foo."

Sometimes minimal matching can help a lot. Imagine you'd like to match everything between "foo" and "bar". Initially, you write something like this:

```
$_ =  "The food is under the bar in the barn.";
if ( /foo(.*)bar/ ) {
    print "got <$1>\n";
}
```

Which perhaps unexpectedly yields:

```
got <d is under the bar in the >
```

That's because `.*` was greedy, so you get everything between the *first* "foo" and the *last* "bar". Here it's more effective to use minimal matching to make sure you get the text between a "foo" and the first "bar" thereafter.

```
  if ( /foo(.*?)bar/ ) { print "got <$1>\n" }
got <d is under the >
```

Here's another example. Let's say you'd like to match a number at the end of a string, and you also want to keep the preceding part of the match. So you write this:

```
$_ = "I have 2 numbers: 53147";
if ( /(.*)(\d*)/ ) {                                # Wrong!
    print "Beginning is <$1>, number is <$2>.\n";
}
```

That won't work at all, because `.*` was greedy and gobbled up the whole string. As `\d*` can match on an empty string the complete regular expression matched successfully.

```
Beginning is <I have 2 numbers: 53147>, number is <>.
```

Here are some variants, most of which don't work:

```
$_ = "I have 2 numbers: 53147";
@pats = qw{
    (.*)(\d*)
    (.*)(\d+)
    (.*?)(\d*)
    (.*?)(\d+)
    (.*)(\d+)$
    (.*?)(\d+)$
    (.*)\b(\d+)$
    (.*\D)(\d+)$
};

for $pat (@pats) {
    printf "%-12s ", $pat;
    if ( /$pat/ ) {
        print "<$1> <$2>\n";
    } else {
        print "FAIL\n";
    }
}
```

That will print out:

```
(.*)(\d*)    <I have 2 numbers: 53147> <>
(.*)(\d+)    <I have 2 numbers: 5314> <7>
(.*?)(\d*)   <> <>
(.*?)(\d+)   <I have > <2>
(.*)(\d+)$   <I have 2 numbers: 5314> <7>
(.*?)(\d+)$  <I have 2 numbers: > <53147>
(.*)\b(\d+)$ <I have 2 numbers: > <53147>
(.*\D)(\d+)$ <I have 2 numbers: > <53147>
```

As you see, this can be a bit tricky. It's important to realize that a regular expression is merely a set of assertions that gives a definition of success. There may be 0, 1, or several different ways that the definition might succeed against a particular string. And if there are multiple ways it might succeed, you need to understand backtracking to know which variety of success you will achieve.

When using look-ahead assertions and negations, this can all get even trickier. Imagine you'd like to find a sequence of non-digits not followed by "123". You might try to write that as

```
$_ = "ABC123";
if ( /^\D*(?!123)/ ) {                # Wrong!
    print "Yup, no 123 in $_\n";
}
```

But that isn't going to match; at least, not the way you're hoping. It claims that there is no 123 in the string. Here's a clearer picture of why that pattern matches, contrary to popular expectations:

```
$x = 'ABC123';
$y = 'ABC445';

print "1: got $1\n" if $x =~ /^(ABC)(?!123)/;
print "2: got $1\n" if $y =~ /^(ABC)(?!123)/;

print "3: got $1\n" if $x =~ /^(\D*)(?!123)/;
print "4: got $1\n" if $y =~ /^(\D*)(?!123)/;
```

This prints

```
2: got ABC
3: got AB
4: got ABC
```

You might have expected test 3 to fail because it seems to a more general purpose version of test 1. The important difference between them is that test 3 contains a quantifier (`\D*`) and so can use backtracking, whereas test 1 will not. What's happening is that you've asked "Is it true that at the start of $x, following 0 or more non-digits, you have something that's not 123?" If the pattern matcher had let `\D*` expand to "ABC", this would have caused the whole pattern to fail.

The search engine will initially match `\D*` with "ABC". Then it will try to match `(?!123)` with "123", which fails. But because a quantifier (`\D*`) has been used in the regular expression, the search engine can backtrack and retry the match differently in the hope of matching the complete regular expression.

The pattern really, *really* wants to succeed, so it uses the standard pattern back-off-and-retry and lets `\D*` expand to just "AB" this time. Now there's indeed something following "AB" that is not "123". It's "C123", which suffices.

We can deal with this by using both an assertion and a negation. We'll say that the first part in $1 must be followed both by a digit and by something that's not "123". Remember that the look-aheads are zero-width expressions–they only look, but don't consume any of the string in their match. So rewriting this way produces what you'd expect; that is, case 5 will fail, but case 6 succeeds:

```
    print "5: got $1\n" if $x =~ /^(\D*)(?=\d)(?!123)/;
    print "6: got $1\n" if $y =~ /^(\D*)(?=\d)(?!123)/;

    6: got ABC
```

In other words, the two zero-width assertions next to each other work as though they're ANDed together, just as you'd use any built-in assertions: `/^$/` matches only if you're at the beginning of the line AND the end of the line simultaneously. The deeper underlying truth is that juxtaposition in regular expressions always means AND, except when you write an explicit OR using the vertical bar. `/ab/` means match "a" AND (then) match "b", although the attempted matches are made at different positions because "a" is not a zero-width assertion, but a one-width assertion.

WARNING: Particularly complicated regular expressions can take exponential time to solve because of the immense number of possible ways they can use backtracking to try for a match. For example, without internal optimizations done by the regular expression engine, this will take a painfully long time to run:

```
    'aaaaaaaaaaaa' =~ /((a{0,5}){0,5})*[c]/
```

And if you used `*`'s in the internal groups instead of limiting them to 0 through 5 matches, then it would take forever–or until you ran out of stack space. Moreover, these internal optimizations are not always applicable. For example, if you put `{0,5}` instead of `*` on the external group, no current optimization is applicable, and the match takes a long time to finish.

A powerful tool for optimizing such beasts is what is known as an "independent group", which does not backtrack (see [(?>pattern)], page 1015). Note also that zero-length look-ahead/look-behind assertions will not backtrack to make the tail match, since they are in "logical" context: only whether they match is considered relevant. For an example where side-effects of look-ahead *might* have influenced the following match, see [(?>pattern)], page 1015.

58.2.7 Version 8 Regular Expressions

In case you're not familiar with the "regular" Version 8 regex routines, here are the pattern-matching rules not described above.

Any single character matches itself, unless it is a *metacharacter* with a special meaning described here or above. You can cause characters that normally function as metacharacters to be interpreted literally by prefixing them with a "\" (e.g., "\." matches a ".", not any character; "\\" matches a "\"). This escape mechanism is also required for the character used as the pattern delimiter.

A series of characters matches that series of characters in the target string, so the pattern `blurfl` would match "blurfl" in the target string.

You can specify a character class, by enclosing a list of characters in `[]`, which will match any character from the list. If the first character after the "[" is "^", the class matches any character not in the list. Within a list, the "-" character specifies a range, so that `a-z` represents all characters between "a" and "z", inclusive. If you want either "-" or "]" itself to be a member of a class, put it at the start of the list (possibly after a "^"), or escape it with a backslash. "-" is also taken literally when it is at the end of the list, just before the closing "]". (The following all specify the same class of three characters: `[-az]`, `[az-]`, and `[a\-z]`. All are different from `[a-z]`, which specifies a class containing twenty-six characters, even on EBCDIC-based character sets.) Also, if you try to use the character classes `\w`, `\W`, `\s`, `\S`, `\d`, or `\D` as endpoints of a range, the `"-"` is understood literally.

Note also that the whole range idea is rather unportable between character sets, except for four situations that Perl handles specially. Any subset of the ranges `[A-Z]`, `[a-z]`, and `[0-9]` are guaranteed to match the expected subset of ASCII characters, no matter what character set the platform is running. The fourth portable way to specify ranges is to use the `\N{...}` syntax to specify either end point of the range. For example, `[\N{U+04}-\N{U+07}]` means to match the Unicode code points `\N{U+04}`, `\N{U+05}`, `\N{U+06}`, and `\N{U+07}`, whatever their native values may be on the platform. Under Section "'strict' mode" in `re` or within a [(?[ ])], page 1017, a warning is raised, if enabled, and the other end point of a range which has a `\N{...}` endpoint is not portably specified. For example,

```
 [\N{U+00}-\x06]    # Warning under "use re 'strict'".
```

It is hard to understand without digging what exactly matches ranges other than subsets of `[A-Z]`, `[a-z]`, and `[0-9]`. A sound principle is to use only ranges that begin from and end at either alphabetics of equal case ([a-e], [A-E]), or digits ([0-9]). Anything else is unsafe or unclear. If in doubt, spell out the range in full.

Characters may be specified using a metacharacter syntax much like that used in C: `"\n"` matches a newline, `"\t"` a tab, `"\r"` a carriage return, `"\f"` a form feed, etc. More generally, *\nnn*, where *nnn* is a string of three octal digits, matches the character whose coded character set value is *nnn*. Similarly, \x*nn*, where *nn* are hexadecimal digits, matches the character whose ordinal is *nn*. The expression \c*x* matches the character control-*x*. Finally, the `"."` metacharacter matches any character except `"\n"` (unless you use `/s`).

You can specify a series of alternatives for a pattern using `"|"` to separate them, so that `fee|fie|foe` will match any of "fee", "fie", or "foe" in the target string (as would `f(e|i|o)e`). The first alternative includes everything from the last pattern delimiter ("(", "(?:", etc. or the beginning of the pattern) up to the first `"|"`, and the last alternative contains everything from the last `"|"` to the next closing pattern delimiter. That's why it's common practice to include alternatives in parentheses: to minimize confusion about where they start and end.

Alternatives are tried from left to right, so the first alternative found for which the entire expression matches, is the one that is chosen. This means that alternatives are not necessarily greedy. For example: when matching `foo|foot` against `"barefoot"`, only the `"foo"` part will match, as that is the first alternative tried, and it successfully matches the target string. (This might not seem important, but it is important when you are capturing matched text using parentheses.)

Also remember that `"|"` is interpreted as a literal within square brackets, so if you write `[fee|fie|foe]` you're really only matching `[feio|]`.

Within a pattern, you may designate subpatterns for later reference by enclosing them in parentheses, and you may refer back to the *n*th subpattern later in the pattern using the metacharacter *\n* or \g*n*. Subpatterns are numbered based on the left to right order of their opening parenthesis. A backreference matches whatever actually matched the subpattern in the string being examined, not the rules for that subpattern. Therefore, `(0|0x)\d*\s\g1\d*` will match "0x1234 0x4321", but not "0x1234 01234", because subpattern 1 matched "0x", even though the rule `0|0x` could potentially match the leading 0 in the second number.

58.2.8 Warning on \1 Instead of $1

Some people get too used to writing things like:

```
$pattern =~ s/(\W)/\\\1/g;
```

This is grandfathered (for \1 to \9) for the RHS of a substitute to avoid shocking the **sed** addicts, but it's a dirty habit to get into. That's because in PerlThink, the righthand side of an `s///` is a double-quoted string. `\1` in the usual double-quoted string means a control-A. The customary Unix meaning of `\1` is kludged in for `s///`. However, if you get into the habit of doing that, you get yourself into trouble if you then add an `/e` modifier.

```
s/(\d+)/ \1 + 1 /eg;            # causes warning under -w
```

Or if you try to do

```
s/(\d+)/\1000/;
```

You can't disambiguate that by saying `\{1}000`, whereas you can fix it with `${1}000`. The operation of interpolation should not be confused with the operation of matching a backreference. Certainly they mean two different things on the *left* side of the `s///`.

58.2.9 Repeated Patterns Matching a Zero-length Substring

WARNING: Difficult material (and prose) ahead. This section needs a rewrite.

Regular expressions provide a terse and powerful programming language. As with most other power tools, power comes together with the ability to wreak havoc.

A common abuse of this power stems from the ability to make infinite loops using regular expressions, with something as innocuous as:

```
'foo' =~ m{ ( o? )* }x;
```

The `o?` matches at the beginning of `'foo'`, and since the position in the string is not moved by the match, `o?` would match again and again because of the `*` quantifier. Another common way to create a similar cycle is with the looping modifier `//g`:

```
@matches = ( 'foo' =~ m{ o? }xg );
```

or

```
print "match: <$&>\n" while 'foo' =~ m{ o? }xg;
```

or the loop implied by split().

However, long experience has shown that many programming tasks may be significantly simplified by using repeated subexpressions that may match zero-length substrings. Here's a simple example being:

```
@chars = split //, $string;           # // is not magic in split
($whitewashed = $string) =~ s/()/ /g; # parens avoid magic s// /
```

Thus Perl allows such constructs, by *forcefully breaking the infinite loop*. The rules for this are different for lower-level loops given by the greedy quantifiers `*+{}`, and for higher-level ones like the `/g` modifier or split() operator.

The lower-level loops are *interrupted* (that is, the loop is broken) when Perl detects that a repeated expression matched a zero-length substring. Thus

```
m{ (?: NON_ZERO_LENGTH | ZERO_LENGTH )* }x;
```

is made equivalent to

```
m{ (?: NON_ZERO_LENGTH )* (?: ZERO_LENGTH )? }x;
```

For example, this program

```
#!perl -l
"aaaaab" =~ /
  (?:
     a                 # non-zero
     |                 # or
    (?{print "hello"}) # print hello whenever this
                       #    branch is tried
    (?=(b))            # zero-width assertion
  )*  # any number of times
 /x;
print $&;
print $1;
```

prints

```
hello
aaaaa
b
```

Notice that "hello" is only printed once, as when Perl sees that the sixth iteration of the outermost (?:)* matches a zero-length string, it stops the *.

The higher-level loops preserve an additional state between iterations: whether the last match was zero-length. To break the loop, the following match after a zero-length match is prohibited to have a length of zero. This prohibition interacts with backtracking (see Section 58.2.6 [Backtracking], page 1021), and so the *second best* match is chosen if the *best* match is of zero length.

For example:

```
 $_ = ’bar’;
 s/\w??/<$&>/g;
```

results in <><b><><a><><r><>. At each position of the string the best match given by non-greedy ?? is the zero-length match, and the *second best* match is what is matched by \w. Thus zero-length matches alternate with one-character-long matches.

Similarly, for repeated m/()/g the second-best match is the match at the position one notch further in the string.

The additional state of being *matched with zero-length* is associated with the matched string, and is reset by each assignment to pos(). Zero-length matches at the end of the previous match are ignored during split.

58.2.10 Combining RE Pieces

Each of the elementary pieces of regular expressions which were described before (such as ab or \Z) could match at most one substring at the given position of the input string. However, in a typical regular expression these elementary pieces are combined into more complicated patterns using combining operators `ST`, `S|T`, `S*` etc. (in these examples `S` and `T` are regular subexpressions).

Such combinations can include alternatives, leading to a problem of choice: if we match a regular expression `a|ab` against `"abc"`, will it match substring `"a"` or `"ab"`? One way to describe which substring is actually matched is the concept of backtracking (see Section 58.2.6 [Backtracking], page 1021). However, this description is too low-level and makes you think in terms of a particular implementation.

Another description starts with notions of "better"/"worse". All the substrings which may be matched by the given regular expression can be sorted from the "best" match to the "worst" match, and it is the "best" match which is chosen. This substitutes the question of "what is chosen?" by the question of "which matches are better, and which are worse?".

Again, for elementary pieces there is no such question, since at most one match at a given position is possible. This section describes the notion of better/worse for combining operators. In the description below `S` and `T` are regular subexpressions.

`ST`

> Consider two possible matches, `AB` and `A'B'`, `A` and `A'` are substrings which can be matched by `S`, `B` and `B'` are substrings which can be matched by `T`.
>
> If `A` is a better match for `S` than `A'`, `AB` is a better match than `A'B'`.
>
> If `A` and `A'` coincide: `AB` is a better match than `AB'` if `B` is a better match for `T` than `B'`.

`S|T`

> When `S` can match, it is a better match than when only `T` can match.
>
> Ordering of two matches for `S` is the same as for `S`. Similar for two matches for `T`.

`S{REPEAT_COUNT}`

> Matches as `SSS...S` (repeated as many times as necessary).

`S{min,max}`

> Matches as `S{max}|S{max-1}|...|S{min+1}|S{min}`.

`S{min,max}?`

> Matches as `S{min}|S{min+1}|...|S{max-1}|S{max}`.

`S?`, `S*`, `S+`

> Same as `S{0,1}`, `S{0,BIG_NUMBER}`, `S{1,BIG_NUMBER}` respectively.

`S??`, `S*?`, `S+?`

> Same as `S{0,1}?`, `S{0,BIG_NUMBER}?`, `S{1,BIG_NUMBER}?` respectively.

`(?>S)`

> Matches the best match for `S` and only that.

`(?=S)`, `(?<=S)`

> Only the best match for `S` is considered. (This is important only if `S` has capturing parentheses, and backreferences are used somewhere else in the whole regular expression.)

`(?!S)`, `(?<!S)`

For this grouping operator there is no need to describe the ordering, since only whether or not `S` can match is important.

`(??{ EXPR })`, `(?`*PARNO*`)`

The ordering is the same as for the regular expression which is the result of EXPR, or the pattern contained by capture group *PARNO*.

`(?(condition)yes-pattern|no-pattern)`

Recall that which of `yes-pattern` or `no-pattern` actually matches is already determined. The ordering of the matches is the same as for the chosen subexpression.

The above recipes describe the ordering of matches *at a given position*. One more rule is needed to understand how a match is determined for the whole regular expression: a match at an earlier position is always better than a match at a later position.

58.2.11 Creating Custom RE Engines

As of Perl 5.10.0, one can create custom regular expression engines. This is not for the faint of heart, as they have to plug in at the C level. See Section 59.1 [perlreapi NAME], page 1032 for more details.

As an alternative, overloaded constants (see `overload`) provide a simple way to extend the functionality of the RE engine, by substituting one pattern for another.

Suppose that we want to enable a new RE escape-sequence `\Y|` which matches at a boundary between whitespace characters and non-whitespace characters. Note that `(?=\S)(?<!\S)|(?!\S)(?<=\S)` matches exactly at these positions, so we want to have each `\Y|` in the place of the more complicated version. We can create a module `customre` to do this:

```
package customre;
use overload;

sub import {
  shift;
  die "No argument to customre::import allowed" if @_;
  overload::constant 'qr' => \&convert;
}

sub invalid { die "/$_[0]/: invalid escape '\\$_[1]'"}

# We must also take care of not escaping the legitimate \\Y|
# sequence, hence the presence of '\\' in the conversion rules.
my %rules = ( '\\' => '\\\\',
              'Y|' => qr/(?=\S)(?<!\S)|(?!\S)(?<=\S)/ );
sub convert {
  my $re = shift;
  $re =~ s{
            \\ ( \\ | Y . )
          }
```

```
            { $rules{$1} or invalid($re,$1) }sgex;
    return $re;
  }
```

Now `use customre` enables the new escape in constant regular expressions, i.e., those without any runtime variable interpolations. As documented in `overload`, this conversion will work only over literal parts of regular expressions. For `\Y|$re\Y|` the variable part of this regular expression needs to be converted explicitly (but only if the special meaning of `\Y|` should be enabled inside $re):

```
  use customre;
  $re = <>;
  chomp $re;
  $re = customre::convert $re;
  /\Y|$re\Y|/;
```

58.2.12 Embedded Code Execution Frequency

The exact rules for how often (??{}) and (?{}) are executed in a pattern are unspecified. In the case of a successful match you can assume that they DWIM and will be executed in left to right order the appropriate number of times in the accepting path of the pattern as would any other meta-pattern. How non-accepting pathways and match failures affect the number of times a pattern is executed is specifically unspecified and may vary depending on what optimizations can be applied to the pattern and is likely to change from version to version.

For instance in

```
"aaabcdeeeee"=~/a(?{print "a"})b(?{print "b"})cde/;
```

the exact number of times "a" or "b" are printed out is unspecified for failure, but you may assume they will be printed at least once during a successful match, additionally you may assume that if "b" is printed, it will be preceded by at least one "a".

In the case of branching constructs like the following:

```
/a(b|(?{ print "a" }))c(?{ print "c" })/;
```

you can assume that the input "ac" will output "ac", and that "abc" will output only "c".

When embedded code is quantified, successful matches will call the code once for each matched iteration of the quantifier. For example:

```
"good" =~ /g(?:o(?{print "o"}))*d/;
```

will output "o" twice.

58.2.13 PCRE/Python Support

As of Perl 5.10.0, Perl supports several Python/PCRE-specific extensions to the regex syntax. While Perl programmers are encouraged to use the Perl-specific syntax, the following are also accepted:

`(?P<NAME>pattern)`

Define a named capture group. Equivalent to `(?<NAME>pattern)`.

`(?P=NAME)`

Backreference to a named capture group. Equivalent to `\g{NAME}`.

`(?P>NAME)`

Subroutine call to a named capture group. Equivalent to `(?&NAME)`.

58.3 BUGS

Many regular expression constructs don't work on EBCDIC platforms.

There are a number of issues with regard to case-insensitive matching in Unicode rules. See `i` under Section 58.2.1 [Modifiers], page 989 above.

This document varies from difficult to understand to completely and utterly opaque. The wandering prose riddled with jargon is hard to fathom in several places.

This document needs a rewrite that separates the tutorial content from the reference content.

58.4 SEE ALSO

Section 66.1 [perlrequick NAME], page 1116.

Section 68.1 [perlretut NAME], page 1131.

Section 48.2.30 [perlop Regexp Quote-Like Operators], page 823.

Section 48.2.32 [perlop Gory details of parsing quoted constructs], page 839.

`perlfaq6`.

[perlfunc pos], page 428.

Section 38.1 [perllocale NAME], page 701.

Section 19.1 [perlebcdic NAME], page 266.

Mastering Regular Expressions by Jeffrey Friedl, published by O'Reilly and Associates.

59 perlreapi

59.1 NAME

perlreapi - Perl regular expression plugin interface

59.2 DESCRIPTION

As of Perl 5.9.5 there is a new interface for plugging and using regular expression engines other than the default one.

Each engine is supposed to provide access to a constant structure of the following format:

```
typedef struct regexp_engine {
    REGEXP* (*comp) (pTHX_
                     const SV * const pattern, const U32 flags);
    I32     (*exec) (pTHX_
                     REGEXP * const rx,
                     char* stringarg,
                     char* strend, char* strbeg,
                     SSize_t minend, SV* sv,
                     void* data, U32 flags);
    char*   (*intuit) (pTHX_
                       REGEXP * const rx, SV *sv,
                       const char * const strbeg,
                       char *strpos, char *strend, U32 flags,
                       struct re_scream_pos_data_s *data);
    SV*     (*checkstr) (pTHX_ REGEXP * const rx);
    void    (*free) (pTHX_ REGEXP * const rx);
    void    (*numbered_buff_FETCH) (pTHX_
                                    REGEXP * const rx,
                                    const I32 paren,
                                    SV * const sv);
    void    (*numbered_buff_STORE) (pTHX_
                                    REGEXP * const rx,
                                    const I32 paren,
                                    SV const * const value);
    I32     (*numbered_buff_LENGTH) (pTHX_
                                     REGEXP * const rx,
                                     const SV * const sv,
                                     const I32 paren);
    SV*     (*named_buff) (pTHX_
                           REGEXP * const rx,
                           SV * const key,
                           SV * const value,
                           U32 flags);
    SV*     (*named_buff_iter) (pTHX_
                                REGEXP * const rx,
                                const SV * const lastkey,
                                const U32 flags);
    SV*     (*qr_package)(pTHX_ REGEXP * const rx);
#ifdef USE_ITHREADS
    void*   (*dupe) (pTHX_ REGEXP * const rx, CLONE_PARAMS *param);
#endif
    REGEXP* (*op_comp) (...);
```

When a regexp is compiled, its `engine` field is then set to point at the appropriate structure, so that when it needs to be used Perl can find the right routines to do so.

In order to install a new regexp handler, `$^H{regcomp}` is set to an integer which (when casted appropriately) resolves to one of these structures. When compiling, the `comp` method is executed, and the resulting `regexp` structure's engine field is expected to point back at the same structure.

The pTHX_ symbol in the definition is a macro used by Perl under threading to provide an extra argument to the routine holding a pointer back to the interpreter that is executing the regexp. So under threading all routines get an extra argument.

59.3 Callbacks

59.3.1 comp

```
    REGEXP* comp(pTHX_ const SV * const pattern, const U32 flags);
```

Compile the pattern stored in `pattern` using the given `flags` and return a pointer to a prepared `REGEXP` structure that can perform the match. See Section 59.4 [The REGEXP structure], page 1041 below for an explanation of the individual fields in the REGEXP struct.

The `pattern` parameter is the scalar that was used as the pattern. Previous versions of Perl would pass two `char*` indicating the start and end of the stringified pattern; the following snippet can be used to get the old parameters:

```
    STRLEN plen;
    char*  exp = SvPV(pattern, plen);
    char* xend = exp + plen;
```

Since any scalar can be passed as a pattern, it's possible to implement an engine that does something with an array (`"ook" =~ [ qw/ eek hlagh / ]`) or with the non-stringified form of a compiled regular expression (`"ook" =~ qr/eek/`). Perl's own engine will always stringify everything using the snippet above, but that doesn't mean other engines have to.

The `flags` parameter is a bitfield which indicates which of the `msixpn` flags the regex was compiled with. It also contains additional info, such as if `use locale` is in effect.

The `eogc` flags are stripped out before being passed to the comp routine. The regex engine does not need to know if any of these are set, as those flags should only affect what Perl does with the pattern and its match variables, not how it gets compiled and executed.

By the time the comp callback is called, some of these flags have already had effect (noted below where applicable). However most of their effect occurs after the comp callback has run, in routines that read the `rx->extflags` field which it populates.

In general the flags should be preserved in `rx->extflags` after compilation, although the regex engine might want to add or delete some of them to invoke or disable some special behavior in Perl. The flags along with any special behavior they cause are documented below:

The pattern modifiers:

`/m` - RXf_PMf_MULTILINE

If this is in `rx->extflags` it will be passed to `Perl_fbm_instr` by `pp_split` which will treat the subject string as a multi-line string.

`/s` - RXf_PMf_SINGLELINE
`/i` - RXf_PMf_FOLD
`/x` - RXf_PMf_EXTENDED

If present on a regex, `"#"` comments will be handled differently by the tokenizer in some cases.

TODO: Document those cases.

`/p` - RXf_PMf_KEEPCOPY

TODO: Document this

Character set

The character set rules are determined by an enum that is contained in this field. This is still experimental and subject to change, but the current interface returns the rules by use of the in-line function `get_regex_charset(const U32 flags)`. The only currently documented value returned from it is REGEX_LOCALE_CHARSET, which is set if `use locale` is in effect. If present in `rx->extflags`, `split` will use the locale dependent definition of whitespace when RXf_SKIPWHITE or RXf_WHITE is in effect. ASCII whitespace is defined as per Section "isSPACE" in `perlapi`, and by the internal macros `is_utf8_space` under UTF-8, and `isSPACE_LC` under `use locale`.

Additional flags:

RXf_SPLIT

This flag was removed in perl 5.18.0. `split ' '` is now special-cased solely in the parser. RXf_SPLIT is still #defined, so you can test for it. This is how it used to work:

If `split` is invoked as `split ' '` or with no arguments (which really means `split(' ', $_)`, see [split], page 453), Perl will set this flag. The regex engine can then check for it and set the SKIPWHITE and WHITE extflags. To do this, the Perl engine does:

```
if (flags & RXf_SPLIT && r->prelen == 1 && r->precomp[0] == ' ')
    r->extflags |= (RXf_SKIPWHITE|RXf_WHITE);
```

These flags can be set during compilation to enable optimizations in the `split` operator.

RXf_SKIPWHITE

This flag was removed in perl 5.18.0. It is still #defined, so you can set it, but doing so will have no effect. This is how it used to work:

If the flag is present in `rx->extflags` `split` will delete whitespace from the start of the subject string before it's operated on. What is considered whitespace depends on if the subject is a UTF-8 string and if the `RXf_PMf_LOCALE` flag is set.

If RXf_WHITE is set in addition to this flag, `split` will behave like `split " "` under the Perl engine.

RXf_START_ONLY

Tells the split operator to split the target string on newlines (`\n`) without invoking the regex engine.

Perl's engine sets this if the pattern is `/^/` (`plen == 1 && *exp == '^'`), even under `/^/s`; see Section 25.1 [split], page 351. Of course a different regex engine might want to use the same optimizations with a different syntax.

RXf_WHITE

Tells the split operator to split the target string on whitespace without invoking the regex engine. The definition of whitespace varies depending on if the target string is a UTF-8 string and on if RXf_PMf_LOCALE is set.

Perl's engine sets this flag if the pattern is `\s+`.

RXf_NULL

Tells the split operator to split the target string on characters. The definition of character varies depending on if the target string is a UTF-8 string.

Perl's engine sets this flag on empty patterns, this optimization makes `split //` much faster than it would otherwise be. It's even faster than `unpack`.

RXf_NO_INPLACE_SUBST

Added in perl 5.18.0, this flag indicates that a regular expression might perform an operation that would interfere with inplace substitution. For instance it might contain lookbehind, or assign to non-magical variables (such as $REGMARK and $REGERROR) during matching. `s///` will skip certain optimisations when this is set.

59.3.2 exec

```
    I32 exec(pTHX_ REGEXP * const rx,
             char *stringarg, char* strend, char* strbeg,
             SSize_t minend, SV* sv,
             void* data, U32 flags);
```

Execute a regexp. The arguments are

rx

The regular expression to execute.

sv

This is the SV to be matched against. Note that the actual char array to be matched against is supplied by the arguments described below; the SV is just used to determine UTF8ness, `pos()` etc.

strbeg

Pointer to the physical start of the string.

strend

Pointer to the character following the physical end of the string (i.e. the \0, if any).

stringarg

Pointer to the position in the string where matching should start; it might not be equal to strbeg (for example in a later iteration of /.../g).

minend

Minimum length of string (measured in bytes from stringarg) that must match; if the engine reaches the end of the match but hasn't reached this position in the string, it should fail.

data

Optimisation data; subject to change.

flags

Optimisation flags; subject to change.

59.3.3 intuit

```
char* intuit(pTHX_
             REGEXP * const rx,
             SV *sv,
             const char * const strbeg,
             char *strpos,
             char *strend,
             const U32 flags,
             struct re_scream_pos_data_s *data);
```

Find the start position where a regex match should be attempted, or possibly if the regex engine should not be run because the pattern can't match. This is called, as appropriate, by the core, depending on the values of the extflags member of the regexp structure.

Arguments:

```
rx:     the regex to match against
sv:     the SV being matched: only used for utf8 flag; the string
        itself is accessed via the pointers below. Note that on
        something like an overloaded SV, SvPOK(sv) may be false
        and the string pointers may point to something unrelated to
        the SV itself.
strbeg: real beginning of string
strpos: the point in the string at which to begin matching
strend: pointer to the byte following the last char of the string
flags   currently unused; set to 0
data:   currently unused; set to NULL
```

59.3.4 checkstr

```
SV* checkstr(pTHX_ REGEXP * const rx);
```

Return a SV containing a string that must appear in the pattern. Used by split for optimising matches.

59.3.5 free

```
void free(pTHX_ REGEXP * const rx);
```

Called by Perl when it is freeing a regexp pattern so that the engine can release any resources pointed to by the `pprivate` member of the `regexp` structure. This is only responsible for freeing private data; Perl will handle releasing anything else contained in the `regexp` structure.

59.3.6 Numbered capture callbacks

Called to get/set the value of `$‘`, `$’`, `$&` and their named equivalents, ${^PREMATCH}, ${^POSTMATCH} and ${^MATCH}, as well as the numbered capture groups (`$1`, `$2`, ...).

The `paren` parameter will be `1` for `$1`, `2` for `$2` and so forth, and have these symbolic values for the special variables:

```
${^PREMATCH}  RX_BUFF_IDX_CARET_PREMATCH
${^POSTMATCH} RX_BUFF_IDX_CARET_POSTMATCH
${^MATCH}     RX_BUFF_IDX_CARET_FULLMATCH
$‘            RX_BUFF_IDX_PREMATCH
$’            RX_BUFF_IDX_POSTMATCH
$&            RX_BUFF_IDX_FULLMATCH
```

Note that in Perl 5.17.3 and earlier, the last three constants were also used for the caret variants of the variables.

The names have been chosen by analogy with `Tie-Scalar` methods names with an additional **LENGTH** callback for efficiency. However named capture variables are currently not tied internally but implemented via magic.

59.3.6.1 numbered_buff_FETCH

```
void numbered_buff_FETCH(pTHX_ REGEXP * const rx, const I32 paren,
                         SV * const sv);
```

Fetch a specified numbered capture. `sv` should be set to the scalar to return, the scalar is passed as an argument rather than being returned from the function because when it’s called Perl already has a scalar to store the value, creating another one would be redundant. The scalar can be set with `sv_setsv`, `sv_setpvn` and friends, see `perlapi`.

This callback is where Perl untaints its own capture variables under taint mode (see Section 70.1 [perlsec NAME], page 1198). See the `Perl_reg_numbered_buff_fetch` function in `regcomp.c` for how to untaint capture variables if that’s something you’d like your engine to do as well.

59.3.6.2 numbered_buff_STORE

```
void    (*numbered_buff_STORE) (pTHX_
                                REGEXP * const rx,
                                const I32 paren,
                                SV const * const value);
```

Set the value of a numbered capture variable. `value` is the scalar that is to be used as the new value. It’s up to the engine to make sure this is used as the new value (or reject it).

Example:

```
if ("ook" =~ /(o*)/) {
    # 'paren' will be '1' and 'value' will be 'ee'
    $1 =~ tr/o/e/;
}
```

Perl's own engine will croak on any attempt to modify the capture variables, to do this in another engine use the following callback (copied from `Perl_reg_numbered_buff_store`):

```
void
Example_reg_numbered_buff_store(pTHX_
                                REGEXP * const rx,
                                const I32 paren,
                                SV const * const value)
{
    PERL_UNUSED_ARG(rx);
    PERL_UNUSED_ARG(paren);
    PERL_UNUSED_ARG(value);

    if (!PL_localizing)
        Perl_croak(aTHX_ PL_no_modify);
}
```

Actually Perl will not *always* croak in a statement that looks like it would modify a numbered capture variable. This is because the STORE callback will not be called if Perl can determine that it doesn't have to modify the value. This is exactly how tied variables behave in the same situation:

```
package CaptureVar;
use parent 'Tie::Scalar';

sub TIESCALAR { bless [] }
sub FETCH { undef }
sub STORE { die "This doesn't get called" }

package main;

tie my $sv => "CaptureVar";
$sv =~ y/a/b/;
```

Because `$sv` is `undef` when the `y///` operator is applied to it, the transliteration won't actually execute and the program won't `die`. This is different to how 5.8 and earlier versions behaved since the capture variables were READONLY variables then; now they'll just die when assigned to in the default engine.

59.3.6.3 numbered_buff_LENGTH

```
I32 numbered_buff_LENGTH (pTHX_
                          REGEXP * const rx,
                          const SV * const sv,
                          const I32 paren);
```

Get the length of a capture variable. There's a special callback for this so that Perl doesn't have to do a FETCH and run length on the result, since the length is (in Perl's case) known from an offset stored in rx->offs, this is much more efficient:

```
I32 s1  = rx->offs[paren].start;
I32 s2  = rx->offs[paren].end;
I32 len = t1 - s1;
```

This is a little bit more complex in the case of UTF-8, see what Perl_reg_numbered_buff_length does with Section "is_utf8_string_loclen" in perlapi.

59.3.7 Named capture callbacks

Called to get/set the value of %+ and %-, as well as by some utility functions in re.

There are two callbacks, named_buff is called in all the cases the FETCH, STORE, DELETE, CLEAR, EXISTS and SCALAR Tie-Hash callbacks would be on changes to %+ and %- and named_buff_iter in the same cases as FIRSTKEY and NEXTKEY.

The flags parameter can be used to determine which of these operations the callbacks should respond to. The following flags are currently defined:

Which Tie-Hash operation is being performed from the Perl level on %+ or %+, if any:

```
RXapif_FETCH
RXapif_STORE
RXapif_DELETE
RXapif_CLEAR
RXapif_EXISTS
RXapif_SCALAR
RXapif_FIRSTKEY
RXapif_NEXTKEY
```

If %+ or %- is being operated on, if any.

```
RXapif_ONE /* %+ */
RXapif_ALL /* %- */
```

If this is being called as re::regname, re::regnames or re::regnames_count, if any. The first two will be combined with RXapif_ONE or RXapif_ALL.

```
RXapif_REGNAME
RXapif_REGNAMES
RXapif_REGNAMES_COUNT
```

Internally %+ and %- are implemented with a real tied interface via Tie-Hash-NamedCapture. The methods in that package will call back into these functions. However the usage of Tie-Hash-NamedCapture for this purpose might change in future releases. For instance this might be implemented by magic instead (would need an extension to mgvtbl).

59.3.7.1 named_buff

```
SV*     (*named_buff) (pTHX_ REGEXP * const rx, SV * const key,
                       SV * const value, U32 flags);
```

59.3.7.2 named_buff_iter

```
SV*     (*named_buff_iter) (pTHX_
                            REGEXP * const rx,
                            const SV * const lastkey,
                            const U32 flags);
```

59.3.8 qr_package

```
SV* qr_package(pTHX_ REGEXP * const rx);
```

The package the qr// magic object is blessed into (as seen by `ref qr//`). It is recommended that engines change this to their package name for identification regardless of if they implement methods on the object.

The package this method returns should also have the internal `Regexp` package in its `@ISA`. `qr//->isa("Regexp")` should always be true regardless of what engine is being used.

Example implementation might be:

```
SV*
Example_qr_package(pTHX_ REGEXP * const rx)
{
    PERL_UNUSED_ARG(rx);
    return newSVpvs("re::engine::Example");
}
```

Any method calls on an object created with `qr//` will be dispatched to the package as a normal object.

```
use re::engine::Example;
my $re = qr//;
$re->meth; # dispatched to re::engine::Example::meth()
```

To retrieve the `REGEXP` object from the scalar in an XS function use the `SvRX` macro, see Section "REGEXP Functions" in `perlapi`.

```
void meth(SV * rv)
PPCODE:
    REGEXP * re = SvRX(sv);
```

59.3.9 dupe

```
void* dupe(pTHX_ REGEXP * const rx, CLONE_PARAMS *param);
```

On threaded builds a regexp may need to be duplicated so that the pattern can be used by multiple threads. This routine is expected to handle the duplication of any private data pointed to by the `pprivate` member of the `regexp` structure. It will be called with the preconstructed new `regexp` structure as an argument, the `pprivate` member will point at the **old** private structure, and it is this routine's responsibility to construct a copy and return a pointer to it (which Perl will then use to overwrite the field as passed to this routine.)

This allows the engine to dupe its private data but also if necessary modify the final structure if it really must.

On unthreaded builds this field doesn't exist.

59.3.10 op_comp

This is private to the Perl core and subject to change. Should be left null.

59.4 The REGEXP structure

The REGEXP struct is defined in `regexp.h`. All regex engines must be able to correctly build such a structure in their Section 59.3.1 [comp], page 1033 routine.

The REGEXP structure contains all the data that Perl needs to be aware of to properly work with the regular expression. It includes data about optimisations that Perl can use to determine if the regex engine should really be used, and various other control info that is needed to properly execute patterns in various contexts, such as if the pattern anchored in some way, or what flags were used during the compile, or if the program contains special constructs that Perl needs to be aware of.

In addition it contains two fields that are intended for the private use of the regex engine that compiled the pattern. These are the `intflags` and `pprivate` members. `pprivate` is a void pointer to an arbitrary structure, whose use and management is the responsibility of the compiling engine. Perl will never modify either of these values.

```
typedef struct regexp {
    /* what engine created this regexp? */
    const struct regexp_engine* engine;

    /* what re is this a lightweight copy of? */
    struct regexp* mother_re;

    /* Information about the match that the Perl core uses to manage
     * things */
    U32 extflags;   /* Flags used both externally and internally */
    I32 minlen;     /* mininum possible number of chars in */
                       string to match */
    I32 minlenret;  /* mininum possible number of chars in $& */
    U32 gofs;       /* chars left of pos that we search from */

    /* substring data about strings that must appear
       in the final match, used for optimisations */
    struct reg_substr_data *substrs;

    U32 nparens;  /* number of capture groups */

    /* private engine specific data */
    U32 intflags;   /* Engine Specific Internal flags */
    void *pprivate; /* Data private to the regex engine which
                       created this object. */

    /* Data about the last/current match. These are modified during
     * matching*/
    U32 lastparen;            /* highest close paren matched ($+) */
```

```
        U32 lastcloseparen;       /* last close paren matched ($^N) */
        regexp_paren_pair *swap;  /* Swap copy of *offs */
        regexp_paren_pair *offs;  /* Array of offsets for (@-) and
                                     (@+) */

        char *subbeg;  /* saved or original string so \digit works
                          forever. */
        SV_SAVED_COPY  /* If non-NULL, SV which is COW from original */
        I32 sublen;    /* Length of string pointed by subbeg */
        I32 suboffset;  /* byte offset of subbeg from logical start of
                           str */
        I32 subcoffset; /* suboffset equiv, but in chars (for @-/@+) */

        /* Information about the match that isn't often used */
        I32 prelen;           /* length of precomp */
        const char *precomp;  /* pre-compilation regular expression */

        char *wrapped;  /* wrapped version of the pattern */
        I32 wraplen;    /* length of wrapped */

        I32 seen_evals;   /* number of eval groups in the pattern - for
                             security checks */
        HV *paren_names;  /* Optional hash of paren names */

        /* Refcount of this regexp */
        I32 refcnt;             /* Refcount of this regexp */
    } regexp;
```

The fields are discussed in more detail below:

59.4.1 engine

This field points at a regexp_engine structure which contains pointers to the subroutines that are to be used for performing a match. It is the compiling routine's responsibility to populate this field before returning the regexp object.

Internally this is set to NULL unless a custom engine is specified in $^H{regcomp}, Perl's own set of callbacks can be accessed in the struct pointed to by RE_ENGINE_PTR.

59.4.2 mother_re

TODO, see http://www.mail-archive.com/perl5-changes@perl.org/msg17328.html

59.4.3 extflags

This will be used by Perl to see what flags the regexp was compiled with, this will normally be set to the value of the flags parameter by the Section 59.3.1 [comp], page 1033 callback. See the Section 59.3.1 [comp], page 1033 documentation for valid flags.

59.4.4 minlen minlenret

The minimum string length (in characters) required for the pattern to match. This is used to prune the search space by not bothering to match any closer to the end of a string than would allow a match. For instance there is no point in even starting the regex engine if the minlen is 10 but the string is only 5 characters long. There is no way that the pattern can match.

`minlenret` is the minimum length (in characters) of the string that would be found in $& after a match.

The difference between `minlen` and `minlenret` can be seen in the following pattern:

```
/ns(?=\d)/
```

where the `minlen` would be 3 but `minlenret` would only be 2 as the \d is required to match but is not actually included in the matched content. This distinction is particularly important as the substitution logic uses the `minlenret` to tell if it can do in-place substitutions (these can result in considerable speed-up).

59.4.5 gofs

Left offset from pos() to start match at.

59.4.6 substrs

Substring data about strings that must appear in the final match. This is currently only used internally by Perl's engine, but might be used in the future for all engines for optimisations.

59.4.7 nparens, lastparen, and lastcloseparen

These fields are used to keep track of how many paren groups could be matched in the pattern, which was the last open paren to be entered, and which was the last close paren to be entered.

59.4.8 intflags

The engine's private copy of the flags the pattern was compiled with. Usually this is the same as `extflags` unless the engine chose to modify one of them.

59.4.9 pprivate

A void* pointing to an engine-defined data structure. The Perl engine uses the `regexp_internal` structure (see Section 64.5.2 [perlreguts Base Structures], page 1112) but a custom engine should use something else.

59.4.10 swap

Unused. Left in for compatibility with Perl 5.10.0.

59.4.11 offs

A `regexp_paren_pair` structure which defines offsets into the string being matched which correspond to the `$&` and `$1`, `$2` etc. captures, the `regexp_paren_pair` struct is defined as follows:

```
typedef struct regexp_paren_pair {
        I32 start;
        I32 end;
    } regexp_paren_pair;
```

If `->offs[num].start` or `->offs[num].end` is `-1` then that capture group did not match. `->offs[0].start/end` represents `$&` (or `${^MATCH}` under `//p`) and `->offs[paren].end` matches `$$paren` where `$paren` = 1>.

59.4.12 precomp prelen

Used for optimisations. `precomp` holds a copy of the pattern that was compiled and `prelen` its length. When a new pattern is to be compiled (such as inside a loop) the internal `regcomp` operator checks if the last compiled `REGEXP`'s `precomp` and `prelen` are equivalent to the new one, and if so uses the old pattern instead of compiling a new one.

The relevant snippet from `Perl_pp_regcomp`:

```
        if (!re || !re->precomp || re->prelen != (I32)len ||
            memNE(re->precomp, t, len))
        /* Compile a new pattern */
```

59.4.13 paren_names

This is a hash used internally to track named capture groups and their offsets. The keys are the names of the buffers the values are dualvars, with the IV slot holding the number of buffers with the given name and the pv being an embedded array of I32. The values may also be contained independently in the data array in cases where named backreferences are used.

59.4.14 substrs

Holds information on the longest string that must occur at a fixed offset from the start of the pattern, and the longest string that must occur at a floating offset from the start of the pattern. Used to do Fast-Boyer-Moore searches on the string to find out if its worth using the regex engine at all, and if so where in the string to search.

59.4.15 subbeg sublen saved_copy suboffset subcoffset

Used during the execution phase for managing search and replace patterns, and for providing the text for `$&`, `$1` etc. `subbeg` points to a buffer (either the original string, or a copy in the case of `RX_MATCH_COPIED(rx)`), and `sublen` is the length of the buffer. The `RX_OFFS` start and end indices index into this buffer.

In the presence of the `REXEC_COPY_STR` flag, but with the addition of the `REXEC_COPY_SKIP_PRE` or `REXEC_COPY_SKIP_POST` flags, an engine can choose not to copy the full buffer (although it must still do so in the presence of `RXf_PMf_KEEPCOPY` or the relevant bits being set in `PL_sawampersand`). In this case, it may set `suboffset` to indicate the number of bytes from the logical start of the buffer to the physical start (i.e. `subbeg`). It should also set `subcoffset`, the number of characters in the offset. The latter is needed to support `@-` and `@+` which work in characters, not bytes.

59.4.16 wrapped wraplen

Stores the string `qr//` stringifies to. The Perl engine for example stores `(?^:eek)` in the case of `qr/eek/`.

When using a custom engine that doesn't support the `(?:)` construct for inline modifiers, it's probably best to have `qr//` stringify to the supplied pattern, note that this will create undesired patterns in cases such as:

```
my $x = qr/a|b/;  # "a|b"
my $y = qr/c/i;   # "c"
my $z = qr/$x$y/; # "a|bc"
```

There's no solution for this problem other than making the custom engine understand a construct like `(?:)`.

59.4.17 `seen_evals`

This stores the number of eval groups in the pattern. This is used for security purposes when embedding compiled regexes into larger patterns with `qr//`.

59.4.18 `refcnt`

The number of times the structure is referenced. When this falls to 0, the regexp is automatically freed by a call to pregfree. This should be set to 1 in each engine's Section 59.3.1 [comp], page 1033 routine.

59.5 HISTORY

Originally part of Section 64.1 [perlreguts NAME], page 1100.

59.6 AUTHORS

Originally written by Yves Orton, expanded by Ævar Arnfjörð Bjarmason.

59.7 LICENSE

60 perlrebackslash

60.1 NAME

perlrebackslash - Perl Regular Expression Backslash Sequences and Escapes

60.2 DESCRIPTION

The top level documentation about Perl regular expressions is found in Section 58.1 [perlre NAME], page 989.

This document describes all backslash and escape sequences. After explaining the role of the backslash, it lists all the sequences that have a special meaning in Perl regular expressions (in alphabetical order), then describes each of them.

Most sequences are described in detail in different documents; the primary purpose of this document is to have a quick reference guide describing all backslash and escape sequences.

60.2.1 The backslash

In a regular expression, the backslash can perform one of two tasks: it either takes away the special meaning of the character following it (for instance, `\|` matches a vertical bar, it's not an alternation), or it is the start of a backslash or escape sequence.

The rules determining what it is are quite simple: if the character following the backslash is an ASCII punctuation (non-word) character (that is, anything that is not a letter, digit, or underscore), then the backslash just takes away any special meaning of the character following it.

If the character following the backslash is an ASCII letter or an ASCII digit, then the sequence may be special; if so, it's listed below. A few letters have not been used yet, so escaping them with a backslash doesn't change them to be special. A future version of Perl may assign a special meaning to them, so if you have warnings turned on, Perl issues a warning if you use such a sequence. [1].

It is however guaranteed that backslash or escape sequences never have a punctuation character following the backslash, not now, and not in a future version of Perl 5. So it is safe to put a backslash in front of a non-word character.

Note that the backslash itself is special; if you want to match a backslash, you have to escape the backslash with a backslash: `/\\/` matches a single backslash.

[1]

There is one exception. If you use an alphanumeric character as the delimiter of your pattern (which you probably shouldn't do for readability reasons), you have to escape the delimiter if you want to match it. Perl won't warn then. See also Section 48.2.32 [perlop Gory details of parsing quoted constructs], page 839.

60.2.2 All the sequences and escapes

Those not usable within a bracketed character class (like `[\da-z]`) are marked as `Not in []`.

```
\000               Octal escape sequence.  See also \o{}.
\1                 Absolute backreference.  Not in [].
\a                 Alarm or bell.
\A                 Beginning of string.  Not in [].
\b{}, \b           Boundary. (\b is a backspace in []).
\B{}, \B           Not a boundary.  Not in [].
\cX                Control-X.
\C                 Single octet, even under UTF-8.  Not in [].
                   (Deprecated)
\d                 Character class for digits.
\D                 Character class for non-digits.
\e                 Escape character.
\E                 Turn off \Q, \L and \U processing.  Not in [].
\f                 Form feed.
\F                 Foldcase till \E.  Not in [].
\g{}, \g1          Named, absolute or relative backreference.
                   Not in [].
\G                 Pos assertion.  Not in [].
\h                 Character class for horizontal whitespace.
\H                 Character class for non horizontal whitespace.
\k{}, \k<>, \k’’   Named backreference.  Not in [].
\K                 Keep the stuff left of \K.  Not in [].
\l                 Lowercase next character.  Not in [].
\L                 Lowercase till \E.  Not in [].
\n                 (Logical) newline character.
\N                 Any character but newline.  Not in [].
\N{}               Named or numbered (Unicode) character or sequence.
\o{}               Octal escape sequence.
\p{}, \pP          Character with the given Unicode property.
\P{}, \PP          Character without the given Unicode property.
\Q                 Quote (disable) pattern metacharacters till \E.  Not
                   in [].
\r                 Return character.
\R                 Generic new line.  Not in [].
\s                 Character class for whitespace.
\S                 Character class for non whitespace.
\t                 Tab character.
\u                 Titlecase next character.  Not in [].
\U                 Uppercase till \E.  Not in [].
\v                 Character class for vertical whitespace.
\V                 Character class for non vertical whitespace.
\w                 Character class for word characters.
\W                 Character class for non-word characters.
\x{}, \x00         Hexadecimal escape sequence.
\X                 Unicode "extended grapheme cluster".  Not in [].
\z                 End of string.  Not in [].
\Z                 End of string.  Not in [].
```

60.2.3 Character Escapes

60.2.3.1 Fixed characters

A handful of characters have a dedicated *character escape*. The following table shows them, along with their ASCII code points (in decimal and hex), their ASCII name, the control escape on ASCII platforms and a short description. (For EBCDIC platforms, see Section 19.7 [perlebcdic OPERATOR DIFFERENCES], page 286.)

Seq.	Code Point		ASCII	Cntrl	Description.
	Dec	Hex			
\a	7	07	BEL	\cG	alarm or bell
\b	8	08	BS	\cH	backspace [1]
\e	27	1B	ESC	\c[	escape character
\f	12	0C	FF	\cL	form feed
\n	10	0A	LF	\cJ	line feed [2]
\r	13	0D	CR	\cM	carriage return
\t	9	09	TAB	\cI	tab

[1]

`\b` is the backspace character only inside a character class. Outside a character class, `\b` alone is a word-character/non-word-character boundary, and `\b{}` is some other type of boundary.

[2]

`\n` matches a logical newline. Perl converts between `\n` and your OS's native newline character when reading from or writing to text files.

60.2.3.2 Example

```
$str =~ /\t/;   # Matches if $str contains a (horizontal) tab.
```

60.2.3.3 Control characters

`\c` is used to denote a control character; the character following `\c` determines the value of the construct. For example the value of `\cA` is `chr(1)`, and the value of `\cb` is `chr(2)`, etc. The gory details are in Section 48.2.30 [perlop Regexp Quote-Like Operators], page 823. A complete list of what `chr(1)`, etc. means for ASCII and EBCDIC platforms is in Section 19.7 [perlebcdic OPERATOR DIFFERENCES], page 286.

Note that `\c\` alone at the end of a regular expression (or doubled-quoted string) is not valid. The backslash must be followed by another character. That is, `\c\`*`X`* means `chr(28) . '`*`X`*`'` for all characters *X*.

To write platform-independent code, you must use `\N{`*`NAME`*`}` instead, like `\N{ESCAPE}` or `\N{U+001B}`, see `charnames`.

Mnemonic: *c*ontrol character.

60.2.3.4 Example

```
$str =~ /\cK/;  # Matches if $str contains a vertical tab (control-K).
```

60.2.3.5 Named or numbered characters and character sequences

Unicode characters have a Unicode name and numeric code point (ordinal) value. Use the `\N{}` construct to specify a character by either of these values. Certain sequences of characters also have names.

To specify by name, the name of the character or character sequence goes between the curly braces.

To specify a character by Unicode code point, use the form `\N{U+`*`code point`*`}`, where *code point* is a number in hexadecimal that gives the code point that Unicode has assigned to the desired character. It is customary but not required to use leading zeros to pad the number to 4 digits. Thus `\N{U+0041}` means `LATIN CAPITAL LETTER A`, and you will rarely see it written without the two leading zeros. `\N{U+0041}` means `"A"` even on EBCDIC machines (where the ordinal value of `"A"` is not 0x41).

It is even possible to give your own names to characters and character sequences. For details, see `charnames`.

(There is an expanded internal form that you may see in debug output: `\N{U+`*`code point`*`.`*`code point`*`...}`. The `...` means any number of these *code point*s separated by dots. This represents the sequence formed by the characters. This is an internal form only, subject to change, and you should not try to use it yourself.)

Mnemonic: *N*amed character.

Note that a character or character sequence expressed as a named or numbered character is considered a character without special meaning by the regex engine, and will match "as is".

60.2.3.6 Example

```
$str =~ /\N{THAI CHARACTER SO SO}/;  # Matches the Thai SO SO character

use charnames 'Cyrillic';            # Loads Cyrillic names.
$str =~ /\N{ZHE}\N{KA}/;             # Match "ZHE" followed by "KA".
```

60.2.3.7 Octal escapes

There are two forms of octal escapes. Each is used to specify a character by its code point specified in octal notation.

One form, available starting in Perl 5.14 looks like `\o{...}`, where the dots represent one or more octal digits. It can be used for any Unicode character.

It was introduced to avoid the potential problems with the other form, available in all Perls. That form consists of a backslash followed by three octal digits. One problem with this form is that it can look exactly like an old-style backreference (see Section 60.2.3.9 [Disambiguation rules between old-style octal escapes and backreferences], page 1050 below.) You can avoid this by making the first of the three digits always a zero, but that makes \077 the largest code point specifiable.

In some contexts, a backslash followed by two or even one octal digits may be interpreted as an octal escape, sometimes with a warning, and because of some bugs, sometimes with surprising results. Also, if you are creating a regex out of smaller snippets concatenated together, and you use fewer than three digits, the beginning of one snippet may be interpreted as adding digits to the ending of the snippet before it. See Section 60.2.6.1 [Absolute referencing], page 1052 for more discussion and examples of the snippet problem.

Note that a character expressed as an octal escape is considered a character without special meaning by the regex engine, and will match "as is".

To summarize, the \o{} form is always safe to use, and the other form is safe to use for code points through \077 when you use exactly three digits to specify them.

Mnemonic: *0*ctal or *o*ctal.

60.2.3.8 Examples (assuming an ASCII platform)

```
$str = "Perl";
$str =~ /\o{120}/;  # Match, "\120" is "P".
$str =~ /\120/;     # Same.
$str =~ /\o{120}+/; # Match, "\120" is "P",
                    # it's repeated at least once.
$str =~ /\120+/;    # Same.
$str =~ /P\053/;    # No match, "\053" is "+" and taken literally.
/\o{23073}/         # Black foreground, white background smiling face.
/\o{4801234567}/    # Raises a warning, and yields chr(4).
```

60.2.3.9 Disambiguation rules between old-style octal escapes and backreferences

Octal escapes of the \000 form outside of bracketed character classes potentially clash with old-style backreferences (see Section 60.2.6.1 [Absolute referencing], page 1052 below). They both consist of a backslash followed by numbers. So Perl has to use heuristics to determine whether it is a backreference or an octal escape. Perl uses the following rules to disambiguate:

1. If the backslash is followed by a single digit, it's a backreference.
2. If the first digit following the backslash is a 0, it's an octal escape.
3. If the number following the backslash is N (in decimal), and Perl already has seen N capture groups, Perl considers this a backreference. Otherwise, it considers it an octal escape. If N has more than three digits, Perl takes only the first three for the octal escape; the rest are matched as is.

   ```
   my $pat  = "(" x 999;
      $pat .= "a";
      $pat .= ")" x 999;
   /^($pat)\1000$/;   #  Matches 'aa'; there are 1000 capture groups.
   /^$pat\1000$/;     #  Matches 'a@0'; there are 999 capture groups
                      #  and \1000 is seen as \100 (a '@') and a '0'.
   ```

You can force a backreference interpretation always by using the \g{...} form. You can the force an octal interpretation always by using the \o{...} form, or for numbers up through \077 (= 63 decimal), by using three digits, beginning with a "0".

60.2.3.10 Hexadecimal escapes

Like octal escapes, there are two forms of hexadecimal escapes, but both start with the sequence `\x`. This is followed by either exactly two hexadecimal digits forming a number, or a hexadecimal number of arbitrary length surrounded by curly braces. The hexadecimal number is the code point of the character you want to express.

Note that a character expressed as one of these escapes is considered a character without special meaning by the regex engine, and will match "as is".

Mnemonic: he*x*adecimal.

60.2.3.11 Examples (assuming an ASCII platform)

```
$str = "Perl";
$str =~ /\x50/;    # Match, "\x50" is "P".
$str =~ /\x50+/;   # Match, "\x50" is "P", it is repeated at least once
$str =~ /P\x2B/;   # No match, "\x2B" is "+" and taken literally.

/\x{2603}\x{2602}/ # Snowman with an umbrella.
                   # The Unicode character 2603 is a snowman,
                   # the Unicode character 2602 is an umbrella.
/\x{263B}/         # Black smiling face.
/\x{263b}/         # Same, the hex digits A - F are case insensitive.
```

60.2.4 Modifiers

A number of backslash sequences have to do with changing the character, or characters following them. `\l` will lowercase the character following it, while `\u` will uppercase (or, more accurately, titlecase) the character following it. They provide functionality similar to the functions `lcfirst` and `ucfirst`.

To uppercase or lowercase several characters, one might want to use `\L` or `\U`, which will lowercase/uppercase all characters following them, until either the end of the pattern or the next occurrence of `\E`, whichever comes first. They provide functionality similar to what the functions `lc` and `uc` provide.

`\Q` is used to quote (disable) pattern metacharacters, up to the next `\E` or the end of the pattern. `\Q` adds a backslash to any character that could have special meaning to Perl. In the ASCII range, it quotes every character that isn't a letter, digit, or underscore. See [perlfunc quotemeta], page 430 for details on what gets quoted for non-ASCII code points. Using this ensures that any character between `\Q` and `\E` will be matched literally, not interpreted as a metacharacter by the regex engine.

`\F` can be used to casefold all characters following, up to the next `\E` or the end of the pattern. It provides the functionality similar to the `fc` function.

Mnemonic: *L*owercase, *U*ppercase, *F*old-case, *Q*uotemeta, *E*nd.

60.2.4.1 Examples

```
$sid     = "sid";
$greg    = "GrEg";
$miranda = "(Miranda)";
$str     =~ /\u$sid/;        # Matches ’Sid’
```

```
$str     =~ /\L$greg/;       # Matches 'greg'
$str     =~ /\Q$miranda\E/;  # Matches '(Miranda)', as if the pattern
                             #   had been written as /\(Miranda\)/
```

60.2.5 Character classes

Perl regular expressions have a large range of character classes. Some of the character classes are written as a backslash sequence. We will briefly discuss those here; full details of character classes can be found in Section 61.1 [perlrecharclass NAME], page 1059.

`\w` is a character class that matches any single *word* character (letters, digits, Unicode marks, and connector punctuation (like the underscore)). `\d` is a character class that matches any decimal digit, while the character class `\s` matches any whitespace character. New in perl 5.10.0 are the classes `\h` and `\v` which match horizontal and vertical whitespace characters.

The exact set of characters matched by `\d`, `\s`, and `\w` varies depending on various pragma and regular expression modifiers. It is possible to restrict the match to the ASCII range by using the `/a` regular expression modifier. See Section 61.1 [perlrecharclass NAME], page 1059.

The uppercase variants (`\W`, `\D`, `\S`, `\H`, and `\V`) are character classes that match, respectively, any character that isn't a word character, digit, whitespace, horizontal whitespace, or vertical whitespace.

Mnemonics: *w*ord, *d*igit, *s*pace, *h*orizontal, *v*ertical.

60.2.5.1 Unicode classes

`\pP` (where `P` is a single letter) and `\p{Property}` are used to match a character that matches the given Unicode property; properties include things like "letter", or "thai character". Capitalizing the sequence to `\PP` and `\P{Property}` make the sequence match a character that doesn't match the given Unicode property. For more details, see Section 61.2.2 [perlrecharclass Backslash sequences], page 1059 and Section 81.2.5 [perlunicode Unicode Character Properties], page 1321.

Mnemonic: *p*roperty.

60.2.6 Referencing

If capturing parenthesis are used in a regular expression, we can refer to the part of the source string that was matched, and match exactly the same thing. There are three ways of referring to such *backreference*: absolutely, relatively, and by name.

60.2.6.1 Absolute referencing

Either `\g`*N* (starting in Perl 5.10.0), or `\`*N* (old-style) where *N* is a positive (unsigned) decimal number of any length is an absolute reference to a capturing group.

N refers to the Nth set of parentheses, so `\g`*N* refers to whatever has been matched by that set of parentheses. Thus `\g1` refers to the first capture group in the regex.

The `\g`*N* form can be equivalently written as `\g{`*N*`}` which avoids ambiguity when building a regex by concatenating shorter strings. Otherwise if you had a regex `qr/$a$b/`, and `$a` contained `"\g1"`, and `$b` contained `"37"`, you would get `/\g137/` which is probably not what you intended.

In the `\N` form, *N* must not begin with a `"0"`, and there must be at least *N* capturing groups, or else *N* is considered an octal escape (but something like `\18` is the same as `\0018`; that is, the octal escape `"\001"` followed by a literal digit `"8"`).

Mnemonic: *g*roup.

60.2.6.2 Examples

```
/(\w+) \g1/;    # Finds a duplicated word, (e.g. "cat cat").
/(\w+) \1/;     # Same thing; written old-style.
/(.)(.)\g2\g1/;  # Match a four letter palindrome (e.g. "ABBA").
```

60.2.6.3 Relative referencing

`\g-`*`N`* (starting in Perl 5.10.0) is used for relative addressing. (It can be written as `\g{-`*`N`*.) It refers to the *N*th group before the `\g{-`*`N`*`}`.

The big advantage of this form is that it makes it much easier to write patterns with references that can be interpolated in larger patterns, even if the larger pattern also contains capture groups.

60.2.6.4 Examples

```
/(A)        # Group 1
 (          # Group 2
   (B)      # Group 3
   \g{-1}   # Refers to group 3 (B)
   \g{-3}   # Refers to group 1 (A)
 )
/x;         # Matches "ABBA".

my $qr = qr /(.)(.)\g{-2}\g{-1}/;  # Matches 'abab', 'cdcd', etc.
/$qr$qr/                           # Matches 'ababcdcd'.
```

60.2.6.5 Named referencing

`\g{`*`name`*`}` (starting in Perl 5.10.0) can be used to back refer to a named capture group, dispensing completely with having to think about capture buffer positions.

To be compatible with .Net regular expressions, `\g{name}` may also be written as `\k{name}`, `\k<name>` or `\k'name'`.

To prevent any ambiguity, *name* must not start with a digit nor contain a hyphen.

60.2.6.6 Examples

```
/(?<word>\w+) \g{word}/ # Finds duplicated word, (e.g. "cat cat")
/(?<word>\w+) \k{word}/ # Same.
/(?<word>\w+) \k<word>/ # Same.
/(?<letter1>.)(?<letter2>.)\g{letter2}\g{letter1}/
                        # Match a four letter palindrome (e.g. "ABBA")
```

60.2.7 Assertions

Assertions are conditions that have to be true; they don't actually match parts of the substring. There are six assertions that are written as backslash sequences.

\A

`\A` only matches at the beginning of the string. If the `/m` modifier isn't used, then `/\A/` is equivalent to `/^/`. However, if the `/m` modifier is used, then `/^/` matches internal newlines, but the meaning of `/\A/` isn't changed by the `/m` modifier. `\A` matches at the beginning of the string regardless whether the `/m` modifier is used.

\z, \Z

`\z` and `\Z` match at the end of the string. If the `/m` modifier isn't used, then `/\Z/` is equivalent to `/$/`; that is, it matches at the end of the string, or one before the newline at the end of the string. If the `/m` modifier is used, then `/$/` matches at internal newlines, but the meaning of `/\Z/` isn't changed by the `/m` modifier. `\Z` matches at the end of the string (or just before a trailing newline) regardless whether the `/m` modifier is used.

`\z` is just like `\Z`, except that it does not match before a trailing newline. `\z` matches at the end of the string only, regardless of the modifiers used, and not just before a newline. It is how to anchor the match to the true end of the string under all conditions.

\G

`\G` is usually used only in combination with the `/g` modifier. If the `/g` modifier is used and the match is done in scalar context, Perl remembers where in the source string the last match ended, and the next time, it will start the match from where it ended the previous time.

`\G` matches the point where the previous match on that string ended, or the beginning of that string if there was no previous match.

Mnemonic: *G*lobal.

\b{}, \b, \B{}, \B

`\b{...}`, available starting in v5.22, matches a boundary (between two characters, or before the first character of the string, or after the final character of the string) based on the Unicode rules for the boundary type specified inside the braces. The currently known boundary types are given a few paragraphs below. `\B{...}` matches at any place between characters where `\b{...}` of the same type doesn't match.

`\b` when not immediately followed by a `"{"` matches at any place between a word (something matched by `\w`) and a non-word character (`\W`); `\B` when not immediately followed by a `"{"` matches at any place between characters where `\b` doesn't match. To get better word matching of natural language text, see `bwb` below.

`\b` and `\B` assume there's a non-word character before the beginning and after the end of the source string; so `\b` will match at the beginning (or end) of the source string if the source string begins (or ends) with a word character. Otherwise, `\B` will match.

Do not use something like `\b=head\d\b` and expect it to match the beginning of a line. It can't, because for there to be a boundary before the non-word "=", there must be a word character immediately previous. All plain `\b` and

\B boundary determinations look for word characters alone, not for non-word characters nor for string ends. It may help to understand how <\b> and <\B> work by equating them as follows:

```
    \b  really means    (?:(?<=\w)(?!\w)|(?<!\w)(?=\w))
    \B  really means    (?:(?<=\w)(?=\w)|(?<!\w)(?!\w))
```

In contrast, \b{...} and \B{...} may or may not match at the beginning and end of the line, depending on the boundary type. These implement the Unicode default boundaries, specified in http://www.unicode.org/reports/tr29/. The boundary types currently available are:

\b{gcb} or \b{g}

This matches a Unicode "Grapheme Cluster Boundary". (Actually Perl always uses the improved "extended" grapheme cluster"). These are explained below under [\X], page 1057. In fact, \X is another way to get the same functionality. It is equivalent to /.+?\b{gcb}/. Use whichever is most convenient for your situation.

\b{sb}

This matches a Unicode "Sentence Boundary". This is an aid to parsing natural language sentences. It gives good, but imperfect results. For example, it thinks that "Mr. Smith" is two sentences. More details are at http://www.unicode.org/reports/tr29/. Note also that it thinks that anything matching [\R], page 1057 (except form feed and vertical tab) is a sentence boundary. \b{sb} works with text designed for word-processors which wrap lines automatically for display, but hard-coded line boundaries are considered to be essentially the ends of text blocks (paragraphs really), and hence the ends of sententces. \b{sb} doesn't do well with text containing embedded newlines, like the source text of the document you are reading. Such text needs to be preprocessed to get rid of the line separators before looking for sentence boundaries. Some people view this as a bug in the Unicode standard, and this behavior is quite subject to change in future Perl versions.

\b{wb}

This matches a Unicode "Word Boundary". This gives better (though not perfect) results for natural language processing than plain \b (without braces) does. For example, it understands that apostrophes can be in the middle of words and that parentheses aren't (see the examples below). More details are at http://www.unicode.org/reports/tr29/.

It is important to realize when you use these Unicode boundaries, that you are taking a risk that a future version of Perl which contains a later version of the Unicode Standard will not work precisely the same way as it did when your code was written. These rules are not considered stable and have been somewhat more subject to change than the rest of the Standard. Unicode reserves the right to change them at will, and Perl reserves the right to update its implementation to Unicode’s new rules. In the past, some changes have been because new characters have been added to the Standard which have different characteristics than all previous characters, so new rules are formulated for handling them. These should not cause any backward compatibility issues. But some changes have changed the treatment of existing characters because the Unicode Technical Committee has decided that the change is warranted for whatever reason. This could be to fix a bug, or because they think better results are obtained with the new rule.

It is also important to realize that these are default boundary definitions, and that implementations may wish to tailor the results for particular purposes and locales.

Unicode defines a fourth boundary type, accessible through the `Unicode-LineBreak` module.

Mnemonic: *b*oundary.

60.2.7.1 Examples

```
 "cat"   =~ /\Acat/;     # Match.
 "cat"   =~ /cat\Z/;     # Match.
 "cat\n" =~ /cat\Z/;     # Match.
 "cat\n" =~ /cat\z/;     # No match.

 "cat"   =~ /\bcat\b/;   # Matches.
 "cats"  =~ /\bcat\b/;   # No match.
 "cat"   =~ /\bcat\B/;   # No match.
 "cats"  =~ /\bcat\B/;   # Match.

 while ("cat dog" =~ /(\w+)/g) {
     print $1;           # Prints ’catdog’
 }
 while ("cat dog" =~ /\G(\w+)/g) {
     print $1;           # Prints ’cat’
 }

 my $s = "He said, \"Is pi 3.14? (I’m not sure).\"";
 print join("|", $s =~ m/ ( .+? \b     ) /xg), "\n";
 print join("|", $s =~ m/ ( .+? \b{wb} ) /xg), "\n";
prints
 He| |said|, "|Is| |pi| |3|.|14|? (|I|’|m| |not| |sure
 He| |said|,| |"|Is| |pi| |3.14|?| |(|I’m| |not| |sure|)|.|"
```

60.2.8 Misc

Here we document the backslash sequences that don’t fall in one of the categories above. These are:

\C

(Deprecated.) `\C` always matches a single octet, even if the source string is encoded in UTF-8 format, and the character to be matched is a multi-octet character. This is very dangerous, because it violates the logical character abstraction and can cause UTF-8 sequences to become malformed.

Use `utf8::encode()` instead.

Mnemonic: o*C*tet.

\K

This appeared in perl 5.10.0. Anything matched left of `\K` is not included in `$&`, and will not be replaced if the pattern is used in a substitution. This lets you write `s/PAT1 \K PAT2/REPL/x` instead of `s/(PAT1) PAT2/${1}REPL/x` or `s/(?<=PAT1) PAT2/REPL/x`.

Mnemonic: *K*eep.

\N

This feature, available starting in v5.12, matches any character that is **not** a newline. It is a short-hand for writing `[^\n]`, and is identical to the `.` metasymbol, except under the `/s` flag, which changes the meaning of `.`, but not `\N`.

Note that `\N{...}` can mean a Section 60.2.3.5 [named or numbered character], page 1049.

Mnemonic: Complement of *n*.

\R

`\R` matches a *generic newline*; that is, anything considered a linebreak sequence by Unicode. This includes all characters matched by `\v` (vertical whitespace), and the multi character sequence `"\x0D\x0A"` (carriage return followed by a line feed, sometimes called the network newline; it's the end of line sequence used in Microsoft text files opened in binary mode). `\R` is equivalent to `(?>\x0D\x0A|\v)`. (The reason it doesn't backtrack is that the sequence is considered inseparable. That means that

```
 "\x0D\x0A" =~ /^\R\x0A$/   # No match
```

fails, because the `\R` matches the entire string, and won't backtrack to match just the `"\x0D"`.) Since `\R` can match a sequence of more than one character, it cannot be put inside a bracketed character class; `/[\R]/` is an error; use `\v` instead. `\R` was introduced in perl 5.10.0.

Note that this does not respect any locale that might be in effect; it matches according to the platform's native character set.

Mnemonic: none really. `\R` was picked because PCRE already uses `\R`, and more importantly because Unicode recommends such a regular expression metacharacter, and suggests `\R` as its notation.

\X

This matches a Unicode *extended grapheme cluster*.

`\X` matches quite well what normal (non-Unicode-programmer) usage would consider a single character. As an example, consider a G with some sort of diacritic mark, such as an arrow. There is no such single character in Unicode, but one can be composed by using a G followed by a Unicode "COMBINING UPWARDS ARROW BELOW", and would be displayed by Unicode-aware software as if it were a single character.

The match is greedy and non-backtracking, so that the cluster is never broken up into smaller components.

See also [\b{gcb}], page 1054.

Mnemonic: e*X*tended Unicode character.

60.2.8.1 Examples

```
$str =~ s/foo\Kbar/baz/g; # Change any 'bar' following a 'foo' to 'baz'
$str =~ s/(.)\K\g1//g;    # Delete duplicated characters.

"\n"   =~ /^\R$/;         # Match, \n   is a generic newline.
"\r"   =~ /^\R$/;         # Match, \r   is a generic newline.
"\r\n" =~ /^\R$/;         # Match, \r\n is a generic newline.

"P\x{307}" =~ /^\X$/     # \X matches a P with a dot above.
```

61 perlrecharclass

61.1 NAME

perlrecharclass - Perl Regular Expression Character Classes

61.2 DESCRIPTION

The top level documentation about Perl regular expressions is found in Section 58.1 [perlre NAME], page 989.

This manual page discusses the syntax and use of character classes in Perl regular expressions.

A character class is a way of denoting a set of characters in such a way that one character of the set is matched. It's important to remember that: matching a character class consumes exactly one character in the source string. (The source string is the string the regular expression is matched against.)

There are three types of character classes in Perl regular expressions: the dot, backslash sequences, and the form enclosed in square brackets. Keep in mind, though, that often the term "character class" is used to mean just the bracketed form. Certainly, most Perl documentation does that.

61.2.1 The dot

The dot (or period), `.` is probably the most used, and certainly the most well-known character class. By default, a dot matches any character, except for the newline. That default can be changed to add matching the newline by using the *single line* modifier: either for the entire regular expression with the `/s` modifier, or locally with `(?s)`. (The `Section 61.2.2.1 [\N], page 1060` backslash sequence, described below, matches any character except newline without regard to the *single line* modifier.)

Here are some examples:

```
"a"  =~  /./       # Match
"."  =~  /./       # Match
""   =~  /./       # No match (dot has to match a character)
"\n" =~  /./       # No match (dot does not match a newline)
"\n" =~  /./s      # Match (global 'single line' modifier)
"\n" =~  /(?s:.)/  # Match (local 'single line' modifier)
"ab" =~  /^.$/     # No match (dot matches one character)
```

61.2.2 Backslash sequences

A backslash sequence is a sequence of characters, the first one of which is a backslash. Perl ascribes special meaning to many such sequences, and some of these are character classes. That is, they match a single character each, provided that the character belongs to the specific set of characters defined by the sequence.

Here's a list of the backslash sequences that are character classes. They are discussed in more detail below. (For the backslash sequences that aren't character classes, see Section 60.1 [perlrebackslash NAME], page 1046.)

```
\d              Match a decimal digit character.
\D              Match a non-decimal-digit character.
\w              Match a "word" character.
\W              Match a non-"word" character.
\s              Match a whitespace character.
\S              Match a non-whitespace character.
\h              Match a horizontal whitespace character.
\H              Match a character that isn't horizontal whitespace.
\v              Match a vertical whitespace character.
\V              Match a character that isn't vertical whitespace.
\N              Match a character that isn't a newline.
\pP, \p{Prop}   Match a character that has the given Unicode property.
\PP, \P{Prop}   Match a character that doesn't have the Unicode property
```

61.2.2.1 \N

`\N`, available starting in v5.12, like the dot, matches any character that is not a newline. The difference is that `\N` is not influenced by the *single line* regular expression modifier (see Section 61.2.1 [The dot], page 1059 above). Note that the form `\N{...}` may mean something completely different. When the `{...}` is a Section 58.2.2.2 [quantifier], page 997, it means to match a non-newline character that many times. For example, `\N{3}` means to match 3 non-newlines; `\N{5,}` means to match 5 or more non-newlines. But if `{...}` is not a legal quantifier, it is presumed to be a named character. See `charnames` for those. For example, none of `\N{COLON}`, `\N{4F}`, and `\N{F4}` contain legal quantifiers, so Perl will try to find characters whose names are respectively `COLON`, `4F`, and `F4`.

61.2.2.2 Digits

`\d` matches a single character considered to be a decimal *digit*. If the `/a` regular expression modifier is in effect, it matches [0-9]. Otherwise, it matches anything that is matched by `\p{Digit}`, which includes [0-9]. (An unlikely possible exception is that under locale matching rules, the current locale might not have `[0-9]` matched by `\d`, and/or might match other characters whose code point is less than 256. The only such locale definitions that are legal would be to match `[0-9]` plus another set of 10 consecutive digit characters; anything else would be in violation of the C language standard, but Perl doesn't currently assume anything in regard to this.)

What this means is that unless the `/a` modifier is in effect `\d` not only matches the digits '0' - '9', but also Arabic, Devanagari, and digits from other languages. This may cause some confusion, and some security issues.

Some digits that `\d` matches look like some of the [0-9] ones, but have different values. For example, BENGALI DIGIT FOUR (U+09EA) looks very much like an ASCII DIGIT EIGHT (U+0038). An application that is expecting only the ASCII digits might be misled, or if the match is `\d+`, the matched string might contain a mixture of digits from different writing systems that look like they signify a number different than they actually do. Section "num()" in `Unicode-UCD` can be used to safely calculate the value, returning `undef` if the input string contains such a mixture.

What `\p{Digit}` means (and hence `\d` except under the `/a` modifier) is `\p{General_Category=Decimal_Number}`, or synonymously, `\p{General_Category=Digit}`.

Starting with Unicode version 4.1, this is the same set of characters matched by `\p{Numeric_Type=Decimal}`. But Unicode also has a different property with a similar name, `\p{Numeric_Type=Digit}`, which matches a completely different set of characters. These characters are things such as `CIRCLED DIGIT ONE` or subscripts, or are from writing systems that lack all ten digits.

The design intent is for `\d` to exactly match the set of characters that can safely be used with "normal" big-endian positional decimal syntax, where, for example 123 means one 'hundred', plus two 'tens', plus three 'ones'. This positional notation does not necessarily apply to characters that match the other type of "digit", `\p{Numeric_Type=Digit}`, and so `\d` doesn't match them.

The Tamil digits (U+0BE6 - U+0BEF) can also legally be used in old-style Tamil numbers in which they would appear no more than one in a row, separated by characters that mean "times 10", "times 100", etc. (See `http://www.unicode.org/notes/tn21`.)

Any character not matched by `\d` is matched by `\D`.

61.2.2.3 Word characters

A `\w` matches a single alphanumeric character (an alphabetic character, or a decimal digit); or a connecting punctuation character, such as an underscore ("_"); or a "mark" character (like some sort of accent) that attaches to one of those. It does not match a whole word. To match a whole word, use `\w+`. This isn't the same thing as matching an English word, but in the ASCII range it is the same as a string of Perl-identifier characters.

If the `/a` modifier is in effect ...

`\w` matches the 63 characters [a-zA-Z0-9_].

otherwise ...

For code points above 255 ...

`\w` matches the same as `\p{Word}` matches in this range. That is, it matches Thai letters, Greek letters, etc. This includes connector punctuation (like the underscore) which connect two words together, or diacritics, such as a `COMBINING TILDE` and the modifier letters, which are generally used to add auxiliary markings to letters.

For code points below 256 ...

if locale rules are in effect ...

`\w` matches the platform's native underscore character plus whatever the locale considers to be alphanumeric.

if Unicode rules are in effect ...

`\w` matches exactly what `\p{Word}` matches.

otherwise ...

`\w` matches [a-zA-Z0-9_].

Which rules apply are determined as described in Section 58.2.1.7 [perlre Which character set modifier is in effect?], page 996.

There are a number of security issues with the full Unicode list of word characters. See `http://unicode.org/reports/tr36`.

Also, for a somewhat finer-grained set of characters that are in programming language identifiers beyond the ASCII range, you may wish to instead use the more customized Section 61.2.2.5 [Unicode Properties], page 1064, `\p{ID_Start}`, `\p{ID_Continue}`, `\p{XID_Start}`, and `\p{XID_Continue}`. See `http://unicode.org/reports/tr31`.

Any character not matched by `\w` is matched by `\W`.

61.2.2.4 Whitespace

`\s` matches any single character considered whitespace.

If the `/a` modifier is in effect ...

> In all Perl versions, `\s` matches the 5 characters [\t\n\f\r]; that is, the horizontal tab, the newline, the form feed, the carriage return, and the space. Starting in Perl v5.18, it also matches the vertical tab, `\cK`. See note `[1]` below for a discussion of this.

otherwise ...

> For code points above 255 ...
>
> > `\s` matches exactly the code points above 255 shown with an "s" column in the table below.
>
> For code points below 256 ...
>
> > if locale rules are in effect ...
> >
> > > `\s` matches whatever the locale considers to be whitespace.
> >
> > if Unicode rules are in effect ...
> >
> > > `\s` matches exactly the characters shown with an "s" column in the table below.
> >
> > otherwise ...
> >
> > > `\s` matches [\t\n\f\r] and, starting in Perl v5.18, the vertical tab, `\cK`. (See note `[1]` below for a discussion of this.) Note that this list doesn't include the non-breaking space.

Which rules apply are determined as described in Section 58.2.1.7 [perlre Which character set modifier is in effect?], page 996.

Any character not matched by `\s` is matched by `\S`.

`\h` matches any character considered horizontal whitespace; this includes the platform's space and tab characters and several others listed in the table below. `\H` matches any character not considered horizontal whitespace. They use the platform's native character set, and do not consider any locale that may otherwise be in use.

`\v` matches any character considered vertical whitespace; this includes the platform's carriage return and line feed characters (newline) plus several other characters, all listed in the table below. `\V` matches any character not considered vertical whitespace. They use the platform's native character set, and do not consider any locale that may otherwise be in use.

`\R` matches anything that can be considered a newline under Unicode rules. It can match a multi-character sequence. It cannot be used inside a bracketed character class;

use \v instead (vertical whitespace). It uses the platform's native character set, and does not consider any locale that may otherwise be in use. Details are discussed in Section 60.1 [perlrebackslash NAME], page 1046.

Note that unlike \s (and \d and \w), \h and \v always match the same characters, without regard to other factors, such as the active locale or whether the source string is in UTF-8 format.

One might think that \s is equivalent to [\h\v]. This is indeed true starting in Perl v5.18, but prior to that, the sole difference was that the vertical tab ("\cK") was not matched by \s.

The following table is a complete listing of characters matched by \s, \h and \v as of Unicode 6.3.

The first column gives the Unicode code point of the character (in hex format), the second column gives the (Unicode) name. The third column indicates by which class(es) the character is matched (assuming no locale is in effect that changes the \s matching).

0x0009	CHARACTER TABULATION	h s	
0x000a	LINE FEED (LF)	vs	
0x000b	LINE TABULATION	vs	[1]
0x000c	FORM FEED (FF)	vs	
0x000d	CARRIAGE RETURN (CR)	vs	
0x0020	SPACE	h s	
0x0085	NEXT LINE (NEL)	vs	[2]
0x00a0	NO-BREAK SPACE	h s	[2]
0x1680	OGHAM SPACE MARK	h s	
0x2000	EN QUAD	h s	
0x2001	EM QUAD	h s	
0x2002	EN SPACE	h s	
0x2003	EM SPACE	h s	
0x2004	THREE-PER-EM SPACE	h s	
0x2005	FOUR-PER-EM SPACE	h s	
0x2006	SIX-PER-EM SPACE	h s	
0x2007	FIGURE SPACE	h s	
0x2008	PUNCTUATION SPACE	h s	
0x2009	THIN SPACE	h s	
0x200a	HAIR SPACE	h s	
0x2028	LINE SEPARATOR	vs	
0x2029	PARAGRAPH SEPARATOR	vs	
0x202f	NARROW NO-BREAK SPACE	h s	
0x205f	MEDIUM MATHEMATICAL SPACE	h s	
0x3000	IDEOGRAPHIC SPACE	h s	

[1]

Prior to Perl v5.18, \s did not match the vertical tab. [^\S\cK] (obscurely) matches what \s traditionally did.

[2]

NEXT LINE and NO-BREAK SPACE may or may not match \s depending on the rules in effect. See Section 61.2.2.4 [the beginning of this section], page 1062.

61.2.2.5 Unicode Properties

\pP and \p{Prop} are character classes to match characters that fit given Unicode properties. One letter property names can be used in the \pP form, with the property name following the \p, otherwise, braces are required. When using braces, there is a single form, which is just the property name enclosed in the braces, and a compound form which looks like \p{name=value}, which means to match if the property "name" for the character has that particular "value". For instance, a match for a number can be written as /\pN/ or as /\p{Number}/, or as /\p{Number=True}/. Lowercase letters are matched by the property *Lowercase_Letter* which has the short form *Ll*. They need the braces, so are written as /\p{Ll}/ or /\p{Lowercase_Letter}/, or /\p{General_Category=Lowercase_Letter}/ (the underscores are optional). /\pLl/ is valid, but means something different. It matches a two character string: a letter (Unicode property \pL), followed by a lowercase l.

If locale rules are not in effect, the use of a Unicode property will force the regular expression into using Unicode rules, if it isn't already.

Note that almost all properties are immune to case-insensitive matching. That is, adding a /i regular expression modifier does not change what they match. There are two sets that are affected. The first set is Uppercase_Letter, Lowercase_Letter, and Titlecase_Letter, all of which match Cased_Letter under /i matching. The second set is Uppercase, Lowercase, and Titlecase, all of which match Cased under /i matching. (The difference between these sets is that some things, such as Roman numerals, come in both upper and lower case, so they are Cased, but aren't considered to be letters, so they aren't Cased_Letters. They're actually Letter_Numbers.) This set also includes its subsets PosixUpper and PosixLower, both of which under /i match PosixAlpha.

For more details on Unicode properties, see Section 81.2.5 [perlunicode Unicode Character Properties], page 1321; for a complete list of possible properties, see Section "Properties accessible through \p{} and \P{}" in perluniprops, which notes all forms that have /i differences. It is also possible to define your own properties. This is discussed in Section 81.2.6 [perlunicode User-Defined Character Properties], page 1329.

Unicode properties are defined (surprise!) only on Unicode code points. Starting in v5.20, when matching against \p and \P, Perl treats non-Unicode code points (those above the legal Unicode maximum of 0x10FFFF) as if they were typical unassigned Unicode code points.

Prior to v5.20, Perl raised a warning and made all matches fail on non-Unicode code points. This could be somewhat surprising:

```
chr(0x110000) =~ \p{ASCII_Hex_Digit=True}     # Fails on Perls < v5.20.
chr(0x110000) =~ \p{ASCII_Hex_Digit=False}    # Also fails on Perls
                                              # < v5.20
```

Even though these two matches might be thought of as complements, until v5.20 they were so only on Unicode code points.

61.2.2.6 Examples

```
"a"  =~  /\w/      # Match, "a" is a 'word' character.
"7"  =~  /\w/      # Match, "7" is a 'word' character as well.
"a"  =~  /\d/      # No match, "a" isn't a digit.
"7"  =~  /\d/      # Match, "7" is a digit.
```

```
" "  =~  /\s/      # Match, a space is whitespace.
"a"  =~  /\D/      # Match, "a" is a non-digit.
"7"  =~  /\D/      # No match, "7" is not a non-digit.
" "  =~  /\S/      # No match, a space is not non-whitespace.

" "  =~  /\h/      # Match, space is horizontal whitespace.
" "  =~  /\v/      # No match, space is not vertical whitespace.
"\r" =~  /\v/      # Match, a return is vertical whitespace.

"a"  =~  /\pL/     # Match, "a" is a letter.
"a"  =~  /\p{Lu}/  # No match, /\p{Lu}/ matches upper case letters.

"\x{0e0b}" =~ /\p{Thai}/  # Match, \x{0e0b} is the character
                          # 'THAI CHARACTER SO SO', and that's in
                          # Thai Unicode class.
"a"  =~  /\P{Lao}/ # Match, as "a" is not a Laotian character.
```

It is worth emphasizing that `\d`, `\w`, etc, match single characters, not complete numbers or words. To match a number (that consists of digits), use `\d+`; to match a word, use `\w+`. But be aware of the security considerations in doing so, as mentioned above.

61.2.3 Bracketed Character Classes

The third form of character class you can use in Perl regular expressions is the bracketed character class. In its simplest form, it lists the characters that may be matched, surrounded by square brackets, like this: `[aeiou]`. This matches one of `a`, `e`, `i`, `o` or `u`. Like the other character classes, exactly one character is matched.* To match a longer string consisting of characters mentioned in the character class, follow the character class with a Section 58.2.2.2 [quantifier], page 997. For instance, `[aeiou]+` matches one or more lowercase English vowels.

Repeating a character in a character class has no effect; it's considered to be in the set only once.

Examples:

```
"e"  =~  /[aeiou]/        # Match, as "e" is listed in the class.
"p"  =~  /[aeiou]/        # No match, "p" is not listed in the class.
"ae" =~  /^[aeiou]$/      # No match, a character class only matches
                          # a single character.
"ae" =~  /^[aeiou]+$/     # Match, due to the quantifier.

-------
```

* There are two exceptions to a bracketed character class matching a single character only. Each requires special handling by Perl to make things work:

- When the class is to match caselessly under `/i` matching rules, and a character that is explicitly mentioned inside the class matches a multiple-character sequence caselessly under Unicode rules, the class will also match that sequence. For example, Unicode says that the letter `LATIN SMALL LETTER SHARP S` should match the sequence `ss` under `/i` rules. Thus,

```
'ss' =~ /\A\N{LATIN SMALL LETTER SHARP S}\z/i             # Matches
'ss' =~ /\A[aeioust\N{LATIN SMALL LETTER SHARP S}]\z/i    # Matches
```

For this to happen, the class must not be inverted (see Section 61.2.3.3 [Negation], page 1068) and the character must be explicitly specified, and not be part of a multi-character range (not even as one of its endpoints). (Section 61.2.3.2 [Character Ranges], page 1067 will be explained shortly.) Therefore,

```
'ss' =~ /\A[\0-\x{ff}]\z/ui       # Doesn't match
'ss' =~ /\A[\0-\N{LATIN SMALL LETTER SHARP S}]\z/ui   # No match
'ss' =~ /\A[\xDF-\xDF]\z/ui   # Matches on ASCII platforms, since
                              # \xDF is LATIN SMALL LETTER SHARP S,
                              # and the range is just a single
                              # element
```

Note that it isn't a good idea to specify these types of ranges anyway.

- Some names known to `\N{...}` refer to a sequence of multiple characters, instead of the usual single character. When one of these is included in the class, the entire sequence is matched. For example,

  ```
  "\N{TAMIL LETTER KA}\N{TAMIL VOWEL SIGN AU}"
                                =~ / ^ [\N{TAMIL SYLLABLE KAU}]  $ /x;
  ```

  matches, because `\N{TAMIL SYLLABLE KAU}` is a named sequence consisting of the two characters matched against. Like the other instance where a bracketed class can match multiple characters, and for similar reasons, the class must not be inverted, and the named sequence may not appear in a range, even one where it is both endpoints. If these happen, it is a fatal error if the character class is within an extended Section 61.2.3.9 [`(?[...])`], page 1073 class; and only the first code point is used (with a `regexp`-type warning raised) otherwise.

61.2.3.1 Special Characters Inside a Bracketed Character Class

Most characters that are meta characters in regular expressions (that is, characters that carry a special meaning like `.`, `*`, or `()` lose their special meaning and can be used inside a character class without the need to escape them. For instance, `[()]` matches either an opening parenthesis, or a closing parenthesis, and the parens inside the character class don't group or capture.

Characters that may carry a special meaning inside a character class are: `\`, `^`, `-`, `[` and `]`, and are discussed below. They can be escaped with a backslash, although this is sometimes not needed, in which case the backslash may be omitted.

The sequence `\b` is special inside a bracketed character class. While outside the character class, `\b` is an assertion indicating a point that does not have either two word characters or two non-word characters on either side, inside a bracketed character class, `\b` matches a backspace character.

The sequences `\a`, `\c`, `\e`, `\f`, `\n`, `\N{`*NAME*`}`, `\N{U+`*hex char*`}`, `\r`, `\t`, and `\x` are also special and have the same meanings as they do outside a bracketed character class.

Also, a backslash followed by two or three octal digits is considered an octal number.

A `[` is not special inside a character class, unless it's the start of a POSIX character class (see Section 61.2.3.5 [POSIX Character Classes], page 1069 below). It normally does not need escaping.

A] is normally either the end of a POSIX character class (see Section 61.2.3.5 [POSIX Character Classes], page 1069 below), or it signals the end of the bracketed character class. If you want to include a] in the set of characters, you must generally escape it.

However, if the] is the *first* (or the second if the first character is a caret) character of a bracketed character class, it does not denote the end of the class (as you cannot have an empty class) and is considered part of the set of characters that can be matched without escaping.

Examples:

```
"+"   =~ /[+?*]/      #  Match, "+" in a character class is not special.
"\cH" =~ /[\b]/       #  Match, \b inside in a character class
                      #  is equivalent to a backspace.
"]"   =~ /[][]/       #  Match, as the character class contains
                      #  both [ and ].
"[]"  =~ /[[]]/       #  Match, the pattern contains a character class
                      #  containing just [, and the character class is
                      #  followed by a ].
```

61.2.3.2 Character Ranges

It is not uncommon to want to match a range of characters. Luckily, instead of listing all characters in the range, one may use the hyphen (-). If inside a bracketed character class you have two characters separated by a hyphen, it's treated as if all characters between the two were in the class. For instance, [0-9] matches any ASCII digit, and [a-m] matches any lowercase letter from the first half of the ASCII alphabet.

Note that the two characters on either side of the hyphen are not necessarily both letters or both digits. Any character is possible, although not advisable. ['-?] contains a range of characters, but most people will not know which characters that means. Furthermore, such ranges may lead to portability problems if the code has to run on a platform that uses a different character set, such as EBCDIC.

If a hyphen in a character class cannot syntactically be part of a range, for instance because it is the first or the last character of the character class, or if it immediately follows a range, the hyphen isn't special, and so is considered a character to be matched literally. If you want a hyphen in your set of characters to be matched and its position in the class is such that it could be considered part of a range, you must escape that hyphen with a backslash.

Examples:

```
[a-z]       #  Matches a character that is a lower case ASCII letter.
[a-fz]      #  Matches any letter between 'a' and 'f' (inclusive) or
            #  the letter 'z'.
[-z]        #  Matches either a hyphen ('-') or the letter 'z'.
[a-f-m]     #  Matches any letter between 'a' and 'f' (inclusive), the
            #  hyphen ('-'), or the letter 'm'.
['-?]       #  Matches any of the characters  '()*+,-./0123456789:;<=>?
            #  (But not on an EBCDIC platform).
[\N{APOSTROPHE}-\N{QUESTION MARK}]
            #  Matches any of the characters  '()*+,-./0123456789:;<=>?
```

```
              #  even on an EBCDIC platform.
[\N{U+27}-\N{U+3F}] # Same. (U+27 is "'", and U+3F is "?")
```

As the final two examples above show, you can achieve portablity to non-ASCII platforms by using the \N{...} form for the range endpoints. These indicate that the specified range is to be interpreted using Unicode values, so [\N{U+27}-\N{U+3F}] means to match \N{U+27}, \N{U+28}, \N{U+29}, ..., \N{U+3D}, \N{U+3E}, and \N{U+3F}, whatever the native code point versions for those are. These are called "Unicode" ranges. If either end is of the \N{...} form, the range is considered Unicode. A regexp warning is raised under "use re 'strict'" if the other endpoint is specified non-portably:

```
[\N{U+00}-\x09]    # Warning under re 'strict'; \x09 is non-portable
[\N{U+00}-\t]      # No warning;
```

Both of the above match the characters \N{U+00} \N{U+01}, ... \N{U+08}, \N{U+09}, but the \x09 looks like it could be a mistake so the warning is raised (under re 'strict') for it.

Perl also guarantees that the ranges A-Z, a-z, 0-9, and any subranges of these match what an English-only speaker would expect them to match on any platform. That is, [A-Z] matches the 26 ASCII uppercase letters; [a-z] matches the 26 lowercase letters; and [0-9] matches the 10 digits. Subranges, like [h-k], match correspondingly, in this case just the four letters "h", "i", "j", and "k". This is the natural behavior on ASCII platforms where the code points (ordinal values) for "h" through "k" are consecutive integers (0x68 through 0x6B). But special handling to achieve this may be needed on platforms with a non-ASCII native character set. For example, on EBCDIC platforms, the code point for "h" is 0x88, "i" is 0x89, "j" is 0x91, and "k" is 0x92. Perl specially treats [h-k] to exclude the seven code points in the gap: 0x8A through 0x90. This special handling is only invoked when the range is a subrange of one of the ASCII uppercase, lowercase, and digit ranges, AND each end of the range is expressed either as a literal, like "A", or as a named character (\N{...}, including the \N{U+... form).

EBCDIC Examples:

```
[i-j]               #  Matches either "i" or "j"
[i-\N{LATIN SMALL LETTER J}]  # Same
[i-\N{U+6A}]        #  Same
[\N{U+69}-\N{U+6A}] #  Same
[\x{89}-\x{91}]     #  Matches 0x89 ("i"), 0x8A .. 0x90, 0x91 ("j")
[i-\x{91}]          #  Same
[\x{89}-j]          #  Same
[i-J]               #  Matches, 0x89 ("i") .. 0xC1 ("J"); special
                    #  handling doesn't apply because range is mixed
                    #  case
```

61.2.3.3 Negation

It is also possible to instead list the characters you do not want to match. You can do so by using a caret (^) as the first character in the character class. For instance, [^a-z] matches any character that is not a lowercase ASCII letter, which therefore includes more than a million Unicode code points. The class is said to be "negated" or "inverted".

This syntax make the caret a special character inside a bracketed character class, but only if it is the first character of the class. So if you want the caret as one of the characters to match, either escape the caret or else don't list it first.

In inverted bracketed character classes, Perl ignores the Unicode rules that normally say that named sequence, and certain characters should match a sequence of multiple characters use under caseless /i matching. Following those rules could lead to highly confusing situations:

```
"ss" =~ /^[^\xDF]+$/ui;   # Matches!
```

This should match any sequences of characters that aren't \xDF nor what \xDF matches under /i. "s" isn't \xDF, but Unicode says that "ss" is what \xDF matches under /i. So which one "wins"? Do you fail the match because the string has ss or accept it because it has an s followed by another s? Perl has chosen the latter. (See note in Section 61.2.3 [Bracketed Character Classes], page 1065 above.)

Examples:

```
"e"  =~  /[^aeiou]/   #  No match, the 'e' is listed.
"x"  =~  /[^aeiou]/   #  Match, as 'x' isn't a lowercase vowel.
"^"  =~  /[^^]/       #  No match, matches anything that isn't a caret.
"^"  =~  /[x^]/       #  Match, caret is not special here.
```

61.2.3.4 Backslash Sequences

You can put any backslash sequence character class (with the exception of \N and \R) inside a bracketed character class, and it will act just as if you had put all characters matched by the backslash sequence inside the character class. For instance, [a-f\d] matches any decimal digit, or any of the lowercase letters between 'a' and 'f' inclusive.

\N within a bracketed character class must be of the forms \N{*name*} or \N{U+*hex char*}, and NOT be the form that matches non-newlines, for the same reason that a dot . inside a bracketed character class loses its special meaning: it matches nearly anything, which generally isn't what you want to happen.

Examples:

```
/[\p{Thai}\d]/     # Matches a character that is either a Thai
                   # character, or a digit.
/[^\p{Arabic}()]/  # Matches a character that is neither an Arabic
                   # character, nor a parenthesis.
```

Backslash sequence character classes cannot form one of the endpoints of a range. Thus, you can't say:

```
/[\p{Thai}-\d]/     # Wrong!
```

61.2.3.5 POSIX Character Classes

POSIX character classes have the form [:class:], where *class* is the name, and the [: and :] delimiters. POSIX character classes only appear *inside* bracketed character classes, and are a convenient and descriptive way of listing a group of characters.

Be careful about the syntax,

```
# Correct:
$string =~ /[[:alpha:]]/
```

```
# Incorrect (will warn):
$string =~ /[:alpha:]/
```

The latter pattern would be a character class consisting of a colon, and the letters `a`, `l`, `p` and `h`.

POSIX character classes can be part of a larger bracketed character class. For example,

```
[01[:alpha:]%]
```

is valid and matches ’0’, ’1’, any alphabetic character, and the percent sign.

Perl recognizes the following POSIX character classes:

```
alpha  Any alphabetical character ("[A-Za-z]").
alnum  Any alphanumeric character ("[A-Za-z0-9]").
ascii  Any character in the ASCII character set.
blank  A GNU extension, equal to a space or a horizontal tab ("\t").
cntrl  Any control character.  See Note [2] below.
digit  Any decimal digit ("[0-9]"), equivalent to "\d".
graph  Any printable character, excluding a space.  See Note [3] below.
lower  Any lowercase character ("[a-z]").
print  Any printable character, including a space.  See Note [4] below.
punct  Any graphical character excluding "word" characters.  Note [5].
space  Any whitespace character. "\s" including the vertical tab
       ("\cK").
upper  Any uppercase character ("[A-Z]").
word   A Perl extension ("[A-Za-z0-9_]"), equivalent to "\w".
xdigit Any hexadecimal digit ("[0-9a-fA-F]").
```

Like the Section 61.2.2.5 [Unicode properties], page 1064, most of the POSIX properties match the same regardless of whether case-insensitive (`/i`) matching is in effect or not. The two exceptions are `[:upper:]` and `[:lower:]`. Under `/i`, they each match the union of `[:upper:]` and `[:lower:]`.

Most POSIX character classes have two Unicode-style `\p` property counterparts. (They are not official Unicode properties, but Perl extensions derived from official Unicode properties.) The table below shows the relation between POSIX character classes and these counterparts.

One counterpart, in the column labelled "ASCII-range Unicode" in the table, matches only characters in the ASCII character set.

The other counterpart, in the column labelled "Full-range Unicode", matches any appropriate characters in the full Unicode character set. For example, `\p{Alpha}` matches not just the ASCII alphabetic characters, but any character in the entire Unicode character set considered alphabetic. An entry in the column labelled "backslash sequence" is a (short) equivalent.

```
[[:...:]]      ASCII-range          Full-range  backslash  Note
                Unicode              Unicode     sequence
-----------------------------------------------------
  alpha      \p{PosixAlpha}       \p{XPosixAlpha}
  alnum      \p{PosixAlnum}       \p{XPosixAlnum}
```

ascii	\p{ASCII}			
blank	\p{PosixBlank}	\p{XPosixBlank}	\h	[1]
		or \p{HorizSpace}		[1]
cntrl	\p{PosixCntrl}	\p{XPosixCntrl}		[2]
digit	\p{PosixDigit}	\p{XPosixDigit}	\d	
graph	\p{PosixGraph}	\p{XPosixGraph}		[3]
lower	\p{PosixLower}	\p{XPosixLower}		
print	\p{PosixPrint}	\p{XPosixPrint}		[4]
punct	\p{PosixPunct}	\p{XPosixPunct}		[5]
	\p{PerlSpace}	\p{XPerlSpace}	\s	[6]
space	\p{PosixSpace}	\p{XPosixSpace}		[6]
upper	\p{PosixUpper}	\p{XPosixUpper}		
word	\p{PosixWord}	\p{XPosixWord}	\w	
xdigit	\p{PosixXDigit}	\p{XPosixXDigit}		

[1]

`\p{Blank}` and `\p{HorizSpace}` are synonyms.

[2]

Control characters don't produce output as such, but instead usually control the terminal somehow: for example, newline and backspace are control characters. On ASCII platforms, in the ASCII range, characters whose code points are between 0 and 31 inclusive, plus 127 (`DEL`) are control characters; on EBCDIC platforms, their counterparts are control characters.

[3]

Any character that is *graphical*, that is, visible. This class consists of all alphanumeric characters and all punctuation characters.

[4]

All printable characters, which is the set of all graphical characters plus those whitespace characters which are not also controls.

[5]

`\p{PosixPunct}` and `[[:punct:]]` in the ASCII range match all non-controls, non-alphanumeric, non-space characters: ``[-!"#$%&'()*+,./:;<=>?@[\\\]^_`{|}~]`` (although if a locale is in effect, it could alter the behavior of `[[:punct:]]`).

The similarly named property, `\p{Punct}`, matches a somewhat different set in the ASCII range, namely `[-!"#%&'()*,./:;?@[\\\]_{}]`. That is, it is missing the nine characters ``[$+<=>^`|~]``. This is because Unicode splits what POSIX considers to be punctuation into two categories, Punctuation and Symbols.

`\p{XPosixPunct}` and (under Unicode rules) `[[:punct:]]`, match what `\p{PosixPunct}` matches in the ASCII range, plus what `\p{Punct}` matches. This is different than strictly matching according to `\p{Punct}`. Another way to say it is that if Unicode rules are in effect, `[[:punct:]]` matches all characters that Unicode considers punctuation, plus all ASCII-range characters that Unicode considers symbols.

[6]

> `\p{XPerlSpace}` and `\p{Space}` match identically starting with Perl v5.18. In earlier versions, these differ only in that in non-locale matching, `\p{XPerlSpace}` did not match the vertical tab, `\cK`. Same for the two ASCII-only range forms.

There are various other synonyms that can be used besides the names listed in the table. For example, `\p{PosixAlpha}` can be written as `\p{Alpha}`. All are listed in Section "Properties accessible through \p{} and \P{}" in `perluniprops`.

Both the `\p` counterparts always assume Unicode rules are in effect. On ASCII platforms, this means they assume that the code points from 128 to 255 are Latin-1, and that means that using them under locale rules is unwise unless the locale is guaranteed to be Latin-1 or UTF-8. In contrast, the POSIX character classes are useful under locale rules. They are affected by the actual rules in effect, as follows:

If the `/a` modifier, is in effect ...

> Each of the POSIX classes matches exactly the same as their ASCII-range counterparts.

otherwise ...

> For code points above 255 ...
>
> > The POSIX class matches the same as its Full-range counterpart.
>
> For code points below 256 ...
>
> > if locale rules are in effect ...
> >
> > > The POSIX class matches according to the locale, except:
> > >
> > > `word`
> > >
> > > > also includes the platform's native underscore character, no matter what the locale is.
> > >
> > > `ascii`
> > >
> > > > on platforms that don't have the POSIX `ascii` extension, this matches just the platform's native ASCII-range characters.
> > >
> > > `blank`
> > >
> > > > on platforms that don't have the POSIX `blank` extension, this matches just the platform's native tab and space characters.
> >
> > if Unicode rules are in effect ...
> >
> > > The POSIX class matches the same as the Full-range counterpart.
> >
> > otherwise ...
> >
> > > The POSIX class matches the same as the ASCII range counterpart.

Which rules apply are determined as described in Section 58.2.1.7 [perlre Which character set modifier is in effect?], page 996.

It is proposed to change this behavior in a future release of Perl so that whether or not Unicode rules are in effect would not change the behavior: Outside of locale, the POSIX classes would behave like their ASCII-range counterparts. If you wish to comment on this proposal, send email to perl5-porters@perl.org.

61.2.3.6 Negation of POSIX character classes

A Perl extension to the POSIX character class is the ability to negate it. This is done by prefixing the class name with a caret (^). Some examples:

POSIX	ASCII-range	Full-range	backslash
	Unicode	Unicode	sequence
[[:^digit:]]	\P{PosixDigit}	\P{XPosixDigit}	\D
[[:^space:]]	\P{PosixSpace}	\P{XPosixSpace}	
	\P{PerlSpace}	\P{XPerlSpace}	\S
[[:^word:]]	\P{PerlWord}	\P{XPosixWord}	\W

The backslash sequence can mean either ASCII- or Full-range Unicode, depending on various factors as described in Section 58.2.1.7 [perlre Which character set modifier is in effect?], page 996.

61.2.3.7 [= =] and [. .]

Perl recognizes the POSIX character classes [=class=] and [.class.], but does not (yet?) support them. Any attempt to use either construct raises an exception.

61.2.3.8 Examples

```
/[[:digit:]]/            # Matches a character that is a digit.
/[01[:lower:]]/          # Matches a character that is either a
                         # lowercase letter, or '0' or '1'.
/[[:digit:][:^xdigit:]]/ # Matches a character that can be anything
                         # except the letters 'a' to 'f' and 'A' to
                         # 'F'.  This is because the main character
                         # class is composed of two POSIX character
                         # classes that are ORed together, one that
                         # matches any digit, and the other that
                         # matches anything that isn't a hex digit.
                         # The OR adds the digits, leaving only the
                         # letters 'a' to 'f' and 'A' to 'F' excluded.
```

61.2.3.9 Extended Bracketed Character Classes

This is a fancy bracketed character class that can be used for more readable and less error-prone classes, and to perform set operations, such as intersection. An example is

```
/(?[ \p{Thai} & \p{Digit} ])/
```

This will match all the digit characters that are in the Thai script.

This is an experimental feature available starting in 5.18, and is subject to change as we gain field experience with it. Any attempt to use it will raise a warning, unless disabled via

```
no warnings "experimental::regex_sets";
```

Comments on this feature are welcome; send email to perl5-porters@perl.org.

We can extend the example above:

```
/(?[ ( \p{Thai} + \p{Lao} ) & \p{Digit} ])/
```

This matches digits that are in either the Thai or Laotian scripts.

Notice the white space in these examples. This construct always has the /x modifier turned on within it.

The available binary operators are:

```
&    intersection
+    union
|    another name for '+', hence means union
-    subtraction (the result matches the set consisting of those
     code points matched by the first operand, excluding any that
     are also matched by the second operand)
^    symmetric difference (the union minus the intersection).  This
     is like an exclusive or, in that the result is the set of code
     points that are matched by either, but not both, of the
     operands.
```

There is one unary operator:

```
!    complement
```

All the binary operators left associate; "&" is higher precedence than the others, which all have equal precedence. The unary operator right associates, and has highest precedence. Thus this follows the normal Perl precedence rules for logical operators. Use parentheses to override the default precedence and associativity.

The main restriction is that everything is a metacharacter. Thus, you cannot refer to single characters by doing something like this:

```
/(?[ a + b ])/ # Syntax error!
```

The easiest way to specify an individual typable character is to enclose it in brackets:

```
/(?[ [a] + [b] ])/
```

(This is the same thing as [ab].) You could also have said the equivalent:

```
/(?[[ a b ]])/
```

(You can, of course, specify single characters by using, \x{...}, \N{...}, etc.)

This last example shows the use of this construct to specify an ordinary bracketed character class without additional set operations. Note the white space within it; /x is turned on even within bracketed character classes, except you can't have comments inside them. Hence,

```
(?[ [#] ])
```

matches the literal character "#". To specify a literal white space character, you can escape it with a backslash, like:

```
/(?[ [ a e i o u \  ] ])/
```

This matches the English vowels plus the SPACE character. All the other escapes accepted by normal bracketed character classes are accepted here as well; but unrecognized escapes that generate warnings in normal classes are fatal errors here.

All warnings from these class elements are fatal, as well as some practices that don't currently warn. For example you cannot say

```
/(?[ [ \xF ] ])/     # Syntax error!
```

You have to have two hex digits after a braceless `\x` (use a leading zero to make two). These restrictions are to lower the incidence of typos causing the class to not match what you thought it would.

If a regular bracketed character class contains a `\p{}` or `\P{}` and is matched against a non-Unicode code point, a warning may be raised, as the result is not Unicode-defined. No such warning will come when using this extended form.

The final difference between regular bracketed character classes and these, is that it is not possible to get these to match a multi-character fold. Thus,

```
/(?[ [\xDF] ])/iu
```

does not match the string `ss`.

You don't have to enclose POSIX class names inside double brackets, hence both of the following work:

```
/(?[ [:word:] - [:lower:] ])/
/(?[ [[:word:]] - [[:lower:]] ])/
```

Any contained POSIX character classes, including things like `\w` and `\D` respect the `/a` (and `/aa`) modifiers.

`(?[ ])` is a regex-compile-time construct. Any attempt to use something which isn't knowable at the time the containing regular expression is compiled is a fatal error. In practice, this means just three limitations:

1. This construct cannot be used within the scope of `use locale` (or the `/l` regex modifier).
2. Any Section 81.2.6 [user-defined property], page 1329 used must be already defined by the time the regular expression is compiled (but note that this construct can be used instead of such properties).
3. A regular expression that otherwise would compile using `/d` rules, and which uses this construct will instead use `/u`. Thus this construct tells Perl that you don't want `/d` rules for the entire regular expression containing it.

Note that skipping white space applies only to the interior of this construct. There must not be any space between any of the characters that form the initial `(?[`. Nor may there be space between the closing `])` characters.

Just as in all regular expressions, the pattern can be built up by including variables that are interpolated at regex compilation time. Care must be taken to ensure that you are getting what you expect. For example:

```
my $thai_or_lao = '\p{Thai} + \p{Lao}';
...
qr/(?[ \p{Digit} & $thai_or_lao ])/;
```

compiles to

```
qr/(?[ \p{Digit} & \p{Thai} + \p{Lao} ])/;
```

But this does not have the effect that someone reading the code would likely expect, as the intersection applies just to `\p{Thai}`, excluding the Laotian. Pitfalls like this can be avoided by parenthesizing the component pieces:

```
my $thai_or_lao = '( \p{Thai} + \p{Lao} )';
```

But any modifiers will still apply to all the components:

```
my $lower = '\p{Lower} + \p{Digit}';
qr/(?[ \p{Greek} & $lower ])/i;
```

matches upper case things. You can avoid surprises by making the components into instances of this construct by compiling them:

```
my $thai_or_lao = qr/(?[ \p{Thai} + \p{Lao} ])/;
my $lower = qr/(?[ \p{Lower} + \p{Digit} ])/;
```

When these are embedded in another pattern, what they match does not change, regardless of parenthesization or what modifiers are in effect in that outer pattern.

Due to the way that Perl parses things, your parentheses and brackets may need to be balanced, even including comments. If you run into any examples, please send them to `perlbug@perl.org`, so that we can have a concrete example for this man page.

We may change it so that things that remain legal uses in normal bracketed character classes might become illegal within this experimental construct. One proposal, for example, is to forbid adjacent uses of the same character, as in `(?[ [aa] ])`. The motivation for such a change is that this usage is likely a typo, as the second `"a"` adds nothing.

62 perlref

62.1 NAME

perlref - Perl references and nested data structures

62.2 NOTE

This is complete documentation about all aspects of references. For a shorter, tutorial introduction to just the essential features, see Section 63.1 [perlreftut NAME], page 1092.

62.3 DESCRIPTION

Before release 5 of Perl it was difficult to represent complex data structures, because all references had to be symbolic–and even then it was difficult to refer to a variable instead of a symbol table entry. Perl now not only makes it easier to use symbolic references to variables, but also lets you have "hard" references to any piece of data or code. Any scalar may hold a hard reference. Because arrays and hashes contain scalars, you can now easily build arrays of arrays, arrays of hashes, hashes of arrays, arrays of hashes of functions, and so on.

Hard references are smart–they keep track of reference counts for you, automatically freeing the thing referred to when its reference count goes to zero. (Reference counts for values in self-referential or cyclic data structures may not go to zero without a little help; see Section 62.3.3 [Circular References], page 1084 for a detailed explanation.) If that thing happens to be an object, the object is destructed. See Section 46.1 [perlobj NAME], page 769 for more about objects. (In a sense, everything in Perl is an object, but we usually reserve the word for references to objects that have been officially "blessed" into a class package.)

Symbolic references are names of variables or other objects, just as a symbolic link in a Unix filesystem contains merely the name of a file. The `*glob` notation is something of a symbolic reference. (Symbolic references are sometimes called "soft references", but please don't call them that; references are confusing enough without useless synonyms.)

In contrast, hard references are more like hard links in a Unix file system: They are used to access an underlying object without concern for what its (other) name is. When the word "reference" is used without an adjective, as in the following paragraph, it is usually talking about a hard reference.

References are easy to use in Perl. There is just one overriding principle: in general, Perl does no implicit referencing or dereferencing. When a scalar is holding a reference, it always behaves as a simple scalar. It doesn't magically start being an array or hash or subroutine; you have to tell it explicitly to do so, by dereferencing it.

That said, be aware that Perl version 5.14 introduces an exception to the rule, for syntactic convenience. Experimental array and hash container function behavior allows array and hash references to be handled by Perl as if they had been explicitly syntactically dereferenced. See Section "Syntactical Enhancements" in `perl5140delta` and Section 25.1 [perlfunc NAME], page 351 for details.

62.3.1 Making References

References can be created in several ways.

1. By using the backslash operator on a variable, subroutine, or value. (This works much like the & (address-of) operator in C.) This typically creates *another* reference to a variable, because there's already a reference to the variable in the symbol table. But the symbol table reference might go away, and you'll still have the reference that the backslash returned. Here are some examples:

   ```
   $scalarref = \$foo;
   $arrayref  = \@ARGV;
   $hashref   = \%ENV;
   $coderef   = \&handler;
   $globref   = \*foo;
   ```

   It isn't possible to create a true reference to an IO handle (filehandle or dirhandle) using the backslash operator. The most you can get is a reference to a typeglob, which is actually a complete symbol table entry. But see the explanation of the `*foo{THING}` syntax below. However, you can still use type globs and globrefs as though they were IO handles.

2. A reference to an anonymous array can be created using square brackets:

   ```
   $arrayref = [1, 2, ['a', 'b', 'c']];
   ```

   Here we've created a reference to an anonymous array of three elements whose final element is itself a reference to another anonymous array of three elements. (The multidimensional syntax described later can be used to access this. For example, after the above, `$arrayref->[2][1]` would have the value `"b"`.)

   Taking a reference to an enumerated list is not the same as using square brackets–instead it's the same as creating a list of references!

   ```
   @list = (\$a, \@b, \%c);
   @list = \($a, @b, %c);      # same thing!
   ```

   As a special case, `\(@foo)` returns a list of references to the contents of `@foo`, not a reference to `@foo` itself. Likewise for `%foo`, except that the key references are to copies (since the keys are just strings rather than full-fledged scalars).

3. A reference to an anonymous hash can be created using curly brackets:

   ```
   $hashref = {
       'Adam'  => 'Eve',
       'Clyde' => 'Bonnie',
   };
   ```

   Anonymous hash and array composers like these can be intermixed freely to produce as complicated a structure as you want. The multidimensional syntax described below works for these too. The values above are literals, but variables and expressions would work just as well, because assignment operators in Perl (even within local() or my()) are executable statements, not compile-time declarations.

   Because curly brackets (braces) are used for several other things including BLOCKs, you may occasionally have to disambiguate braces at the beginning of a statement by putting a `+` or a `return` in front so that Perl realizes the opening brace isn't starting a BLOCK. The economy and mnemonic value of using curlies is deemed worth this occasional extra hassle.

For example, if you wanted a function to make a new hash and return a reference to it, you have these options:

```
sub hashem {        { @_ } }   # silently wrong
sub hashem {       +{ @_ } }   # ok
sub hashem { return { @_ } }   # ok
```

On the other hand, if you want the other meaning, you can do this:

```
sub showem {        { @_ } }   # ambiguous (currently ok,
                               # but may change)
sub showem {       {; @_ } }   # ok
sub showem { { return @_ } }   # ok
```

The leading `+{` and `{;` always serve to disambiguate the expression to mean either the HASH reference, or the BLOCK.

4. A reference to an anonymous subroutine can be created by using `sub` without a subname:

```
$coderef = sub { print "Boink!\n" };
```

Note the semicolon. Except for the code inside not being immediately executed, a `sub {}` is not so much a declaration as it is an operator, like `do{}` or `eval{}`. (However, no matter how many times you execute that particular line (unless you're in an `eval("...")`), $coderef will still have a reference to the *same* anonymous subroutine.)

Anonymous subroutines act as closures with respect to my() variables, that is, variables lexically visible within the current scope. Closure is a notion out of the Lisp world that says if you define an anonymous function in a particular lexical context, it pretends to run in that context even when it's called outside the context.

In human terms, it's a funny way of passing arguments to a subroutine when you define it as well as when you call it. It's useful for setting up little bits of code to run later, such as callbacks. You can even do object-oriented stuff with it, though Perl already provides a different mechanism to do that–see Section 46.1 [perlobj NAME], page 769.

You might also think of closure as a way to write a subroutine template without using eval(). Here's a small example of how closures work:

```
sub newprint {
    my $x = shift;
    return sub { my $y = shift; print "$x, $y!\n"; };
}
$h = newprint("Howdy");
$g = newprint("Greetings");

# Time passes...

&$h("world");
&$g("earthlings");
```

This prints

```
Howdy, world!
```

```
    Greetings, earthlings!
```

Note particularly that $x continues to refer to the value passed into newprint() *despite* "my $x" having gone out of scope by the time the anonymous subroutine runs. That's what a closure is all about.

This applies only to lexical variables, by the way. Dynamic variables continue to work as they have always worked. Closure is not something that most Perl programmers need trouble themselves about to begin with.

5. References are often returned by special subroutines called constructors. Perl objects are just references to a special type of object that happens to know which package it's associated with. Constructors are just special subroutines that know how to create that association. They do so by starting with an ordinary reference, and it remains an ordinary reference even while it's also being an object. Constructors are often named `new()`. You *can* call them indirectly:

```
    $objref = new Doggie( Tail => 'short', Ears => 'long' );
```

But that can produce ambiguous syntax in certain cases, so it's often better to use the direct method invocation approach:

```
    $objref   = Doggie->new(Tail => 'short', Ears => 'long');

    use Term::Cap;
    $terminal = Term::Cap->Tgetent( { OSPEED => 9600 });

    use Tk;
    $main    = MainWindow->new();
    $menubar = $main->Frame(-relief              => "raised",
                            -borderwidth         => 2)
```

6. References of the appropriate type can spring into existence if you dereference them in a context that assumes they exist. Because we haven't talked about dereferencing yet, we can't show you any examples yet.

7. A reference can be created by using a special syntax, lovingly known as the *foo{THING} syntax. *foo{THING} returns a reference to the THING slot in *foo (which is the symbol table entry which holds everything known as foo).

```
    $scalarref = *foo{SCALAR};
    $arrayref  = *ARGV{ARRAY};
    $hashref   = *ENV{HASH};
    $coderef   = *handler{CODE};
    $ioref     = *STDIN{IO};
    $globref   = *foo{GLOB};
    $formatref = *foo{FORMAT};
    $globname  = *foo{NAME};    # "foo"
    $pkgname   = *foo{PACKAGE}; # "main"
```

Most of these are self-explanatory, but `*foo{IO}` deserves special attention. It returns the IO handle, used for file handles (⟨undefined⟩ [perlfunc open], page ⟨undefined⟩), sockets (⟨undefined⟩ [perlfunc socket], page ⟨undefined⟩ and ⟨undefined⟩ [perlfunc socketpair], page ⟨undefined⟩), and directory handles (⟨undefined⟩ [perlfunc opendir], page ⟨undefined⟩). For compatibility with previous versions of Perl, *foo{FILEHANDLE} is a synonym for *foo{IO}, though it is deprecated as of 5.8.0. If deprecation warnings are in effect, it will warn of its use.

*foo{THING} returns undef if that particular THING hasn't been used yet, except in the case of scalars. *foo{SCALAR} returns a reference to an anonymous scalar if $foo hasn't been used yet. This might change in a future release.

*foo{NAME} and *foo{PACKAGE} are the exception, in that they return strings, rather than references. These return the package and name of the typeglob itself, rather than one that has been assigned to it. So, after *foo=*Foo::bar, *foo will become "*Foo::bar" when used as a string, but *foo{PACKAGE} and *foo{NAME} will continue to produce "main" and "foo", respectively.

*foo{IO} is an alternative to the *HANDLE mechanism given in Section 11.2.10 [perldata Typeglobs and Filehandles], page 85 for passing filehandles into or out of subroutines, or storing into larger data structures. Its disadvantage is that it won't create a new filehandle for you. Its advantage is that you have less risk of clobbering more than you want to with a typeglob assignment. (It still conflates file and directory handles, though.) However, if you assign the incoming value to a scalar instead of a typeglob as we do in the examples below, there's no risk of that happening.

```
splutter(*STDOUT);          # pass the whole glob
splutter(*STDOUT{IO});      # pass both file and dir handles

sub splutter {
    my $fh = shift;
    print $fh "her um well a hmmm\n";
}

$rec = get_rec(*STDIN);     # pass the whole glob
$rec = get_rec(*STDIN{IO}); # pass both file and dir handles

sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

62.3.2 Using References

That's it for creating references. By now you're probably dying to know how to use references to get back to your long-lost data. There are several basic methods.

1. Anywhere you'd put an identifier (or chain of identifiers) as part of a variable or subroutine name, you can replace the identifier with a simple scalar variable containing a reference of the correct type:

```
$bar = $$scalarref;
push(@$arrayref, $filename);
$$arrayref[0] = "January";
$$hashref{"KEY"} = "VALUE";
&$coderef(1,2,3);
print $globref "output\n";
```

It's important to understand that we are specifically *not* dereferencing `$arrayref[0]` or `$hashref{"KEY"}` there. The dereference of the scalar variable happens *before* it does any key lookups. Anything more complicated than a simple scalar variable must use methods 2 or 3 below. However, a "simple scalar" includes an identifier that itself uses method 1 recursively. Therefore, the following prints "howdy".

```
$refrefref = \\\"howdy";
print $$$$refrefref;
```

2. Anywhere you'd put an identifier (or chain of identifiers) as part of a variable or subroutine name, you can replace the identifier with a BLOCK returning a reference of the correct type. In other words, the previous examples could be written like this:

```
$bar = ${$scalarref};
push(@{$arrayref}, $filename);
${$arrayref}[0] = "January";
${$hashref}{"KEY"} = "VALUE";
&{$coderef}(1,2,3);
$globref->print("output\n");  # iff IO::Handle is loaded
```

Admittedly, it's a little silly to use the curlies in this case, but the BLOCK can contain any arbitrary expression, in particular, subscripted expressions:

```
&{ $dispatch{$index} }(1,2,3);      # call correct routine
```

Because of being able to omit the curlies for the simple case of `$$x`, people often make the mistake of viewing the dereferencing symbols as proper operators, and wonder about their precedence. If they were, though, you could use parentheses instead of braces. That's not the case. Consider the difference below; case 0 is a short-hand version of case 1, *not* case 2:

```
$$hashref{"KEY"}   = "VALUE";       # CASE 0
${$hashref}{"KEY"} = "VALUE";       # CASE 1
${$hashref{"KEY"}} = "VALUE";       # CASE 2
${$hashref->{"KEY"}} = "VALUE";     # CASE 3
```

Case 2 is also deceptive in that you're accessing a variable called %hashref, not dereferencing through $hashref to the hash it's presumably referencing. That would be case 3.

3. Subroutine calls and lookups of individual array elements arise often enough that it gets cumbersome to use method 2. As a form of syntactic sugar, the examples for method 2 may be written:

```
$arrayref->[0] = "January";   # Array element
$hashref->{"KEY"} = "VALUE";  # Hash element
$coderef->(1,2,3);            # Subroutine call
```

The left side of the arrow can be any expression returning a reference, including a previous dereference. Note that `$array[$x]` is *not* the same thing as `$array->[$x]` here:

```
$array[$x]->{"foo"}->[0] = "January";
```

This is one of the cases we mentioned earlier in which references could spring into existence when in an lvalue context. Before this statement, `$array[$x]` may have been undefined. If so, it's automatically defined with a hash reference so that we can look up {"foo"} in it. Likewise $array[$x]->{"foo"} will automatically get defined with an array reference so that we can look up [0] in it. This process is called *autovivification*.

One more thing here. The arrow is optional *between* brackets subscripts, so you can shrink the above down to

```
$array[$x]{"foo"}[0] = "January";
```

Which, in the degenerate case of using only ordinary arrays, gives you multidimensional arrays just like C's:

```
$score[$x][$y][$z] += 42;
```

Well, okay, not entirely like C's arrays, actually. C doesn't know how to grow its arrays on demand. Perl does.

4. If a reference happens to be a reference to an object, then there are probably methods to access the things referred to, and you should probably stick to those methods unless you're in the class package that defines the object's methods. In other words, be nice, and don't violate the object's encapsulation without a very good reason. Perl does not enforce encapsulation. We are not totalitarians here. We do expect some basic civility though.

Using a string or number as a reference produces a symbolic reference, as explained above. Using a reference as a number produces an integer representing its storage location in memory. The only useful thing to be done with this is to compare two references numerically to see whether they refer to the same location.

```
if ($ref1 == $ref2) {  # cheap numeric compare of references
    print "refs 1 and 2 refer to the same thing\n";
}
```

Using a reference as a string produces both its referent's type, including any package blessing as described in Section 46.1 [perlobj NAME], page 769, as well as the numeric address expressed in hex. The ref() operator returns just the type of thing the reference is pointing to, without the address. See [perlfunc ref], page 435 for details and examples of its use.

The bless() operator may be used to associate the object a reference points to with a package functioning as an object class. See Section 46.1 [perlobj NAME], page 769.

A typeglob may be dereferenced the same way a reference can, because the dereference syntax always indicates the type of reference desired. So ${*foo} and ${\$foo} both indicate the same scalar variable.

Here's a trick for interpolating a subroutine call into a string:

```
print "My sub returned @{[mysub(1,2,3)]} that time.\n";
```

The way it works is that when the @{...} is seen in the double-quoted string, it's evaluated as a block. The block creates a reference to an anonymous array containing the results of the call to mysub(1,2,3). So the whole block returns a reference to an array, which is then dereferenced by @{...} and stuck into the double-quoted string. This chicanery is also useful for arbitrary expressions:

```
print "That yields @{[$n + 5]} widgets\n";
```

Similarly, an expression that returns a reference to a scalar can be dereferenced via ${...}. Thus, the above expression may be written as:

```
print "That yields ${\($n + 5)} widgets\n";
```

62.3.3 Circular References

It is possible to create a "circular reference" in Perl, which can lead to memory leaks. A circular reference occurs when two references contain a reference to each other, like this:

```
my $foo = {};
my $bar = { foo => $foo };
$foo->{bar} = $bar;
```

You can also create a circular reference with a single variable:

```
my $foo;
$foo = \$foo;
```

In this case, the reference count for the variables will never reach 0, and the references will never be garbage-collected. This can lead to memory leaks.

Because objects in Perl are implemented as references, it's possible to have circular references with objects as well. Imagine a TreeNode class where each node references its parent and child nodes. Any node with a parent will be part of a circular reference.

You can break circular references by creating a "weak reference". A weak reference does not increment the reference count for a variable, which means that the object can go out of scope and be destroyed. You can weaken a reference with the `weaken` function exported by the `Scalar-Util` module.

Here's how we can make the first example safer:

```
use Scalar::Util 'weaken';

my $foo = {};
my $bar = { foo => $foo };
$foo->{bar} = $bar;

weaken $foo->{bar};
```

The reference from `$foo` to `$bar` has been weakened. When the `$bar` variable goes out of scope, it will be garbage-collected. The next time you look at the value of the `$foo->{bar}` key, it will be `undef`.

This action at a distance can be confusing, so you should be careful with your use of weaken. You should weaken the reference in the variable that will go out of scope *first*. That way, the longer-lived variable will contain the expected reference until it goes out of scope.

62.3.4 Symbolic references

We said that references spring into existence as necessary if they are undefined, but we didn't say what happens if a value used as a reference is already defined, but *isn't* a hard reference. If you use it as a reference, it'll be treated as a symbolic reference. That is, the value of the scalar is taken to be the *name* of a variable, rather than a direct link to a (possibly) anonymous value.

People frequently expect it to work like this. So it does.

```
$name = "foo";
$$name = 1;                 # Sets $foo
${$name} = 2;               # Sets $foo
```

```
${$name x 2} = 3;           # Sets $foofoo
$name->[0] = 4;             # Sets $foo[0]
@$name = ();                # Clears @foo
&$name();                   # Calls &foo()
$pack = "THAT";
${"${pack}::$name"} = 5;    # Sets $THAT::foo without eval
```

This is powerful, and slightly dangerous, in that it's possible to intend (with the utmost sincerity) to use a hard reference, and accidentally use a symbolic reference instead. To protect against that, you can say

```
use strict 'refs';
```

and then only hard references will be allowed for the rest of the enclosing block. An inner block may countermand that with

```
no strict 'refs';
```

Only package variables (globals, even if localized) are visible to symbolic references. Lexical variables (declared with my()) aren't in a symbol table, and thus are invisible to this mechanism. For example:

```
local $value = 10;
$ref = "value";
{
    my $value = 20;
    print $$ref;
}
```

This will still print 10, not 20. Remember that local() affects package variables, which are all "global" to the package.

62.3.5 Not-so-symbolic references

Brackets around a symbolic reference can simply serve to isolate an identifier or variable name from the rest of an expression, just as they always have within a string. For example,

```
$push = "pop on ";
print "${push}over";
```

has always meant to print "pop on over", even though push is a reserved word. This is generalized to work the same without the enclosing double quotes, so that

```
print ${push} . "over";
```

and even

```
print ${ push } . "over";
```

will have the same effect. This construct is *not* considered to be a symbolic reference when you're using strict refs:

```
use strict 'refs';
${ bareword };      # Okay, means $bareword.
${ "bareword" };    # Error, symbolic reference.
```

Similarly, because of all the subscripting that is done using single words, the same rule applies to any bareword that is used for subscripting a hash. So now, instead of writing

```
$array{ "aaa" }{ "bbb" }{ "ccc" }
```

you can write just

```
$array{ aaa }{ bbb }{ ccc }
```

and not worry about whether the subscripts are reserved words. In the rare event that you do wish to do something like

```
$array{ shift }
```

you can force interpretation as a reserved word by adding anything that makes it more than a bareword:

```
$array{ shift() }
$array{ +shift }
$array{ shift @_ }
```

The `use warnings` pragma or the **-w** switch will warn you if it interprets a reserved word as a string. But it will no longer warn you about using lowercase words, because the string is effectively quoted.

62.3.6 Pseudo-hashes: Using an array as a hash

Pseudo-hashes have been removed from Perl. The 'fields' pragma remains available.

62.3.7 Function Templates

As explained above, an anonymous function with access to the lexical variables visible when that function was compiled, creates a closure. It retains access to those variables even though it doesn't get run until later, such as in a signal handler or a Tk callback.

Using a closure as a function template allows us to generate many functions that act similarly. Suppose you wanted functions named after the colors that generated HTML font changes for the various colors:

```
print "Be ", red("careful"), "with that ", green("light");
```

The red() and green() functions would be similar. To create these, we'll assign a closure to a typeglob of the name of the function we're trying to build.

```
@colors = qw(red blue green yellow orange purple violet);
for my $name (@colors) {
    no strict 'refs';       # allow symbol table manipulation
    *$name = *{uc $name} = sub { "<FONT COLOR='$name'>@_</FONT>" };
}
```

Now all those different functions appear to exist independently. You can call red(), RED(), blue(), BLUE(), green(), etc. This technique saves on both compile time and memory use, and is less error-prone as well, since syntax checks happen at compile time. It's critical that any variables in the anonymous subroutine be lexicals in order to create a proper closure. That's the reasons for the `my` on the loop iteration variable.

This is one of the only places where giving a prototype to a closure makes much sense. If you wanted to impose scalar context on the arguments of these functions (probably not a wise idea for this particular example), you could have written it this way instead:

```
*$name = sub ($) { "<FONT COLOR='$name'>$_[0]</FONT>" };
```

However, since prototype checking happens at compile time, the assignment above happens too late to be of much use. You could address this by putting the whole loop of assignments within a BEGIN block, forcing it to occur during compilation.

Access to lexicals that change over time–like those in the `for` loop above, basically aliases to elements from the surrounding lexical scopes– only works with anonymous subs, not with named subroutines. Generally said, named subroutines do not nest properly and should only be declared in the main package scope.

This is because named subroutines are created at compile time so their lexical variables get assigned to the parent lexicals from the first execution of the parent block. If a parent scope is entered a second time, its lexicals are created again, while the nested subs still reference the old ones.

Anonymous subroutines get to capture each time you execute the `sub` operator, as they are created on the fly. If you are accustomed to using nested subroutines in other programming languages with their own private variables, you'll have to work at it a bit in Perl. The intuitive coding of this type of thing incurs mysterious warnings about "will not stay shared" due to the reasons explained above. For example, this won't work:

```
sub outer {
    my $x = $_[0] + 35;
    sub inner { return $x * 19 }   # WRONG
    return $x + inner();
}
```

A work-around is the following:

```
sub outer {
    my $x = $_[0] + 35;
    local *inner = sub { return $x * 19 };
    return $x + inner();
}
```

Now inner() can only be called from within outer(), because of the temporary assignments of the anonymous subroutine. But when it does, it has normal access to the lexical variable $x from the scope of outer() at the time outer is invoked.

This has the interesting effect of creating a function local to another function, something not normally supported in Perl.

62.4 WARNING

You may not (usefully) use a reference as the key to a hash. It will be converted into a string:

```
$x{ \$a } = $a;
```

If you try to dereference the key, it won't do a hard dereference, and you won't accomplish what you're attempting. You might want to do something more like

```
$r = \@a;
$x{ $r } = $r;
```

And then at least you can use the values(), which will be real refs, instead of the keys(), which won't.

The standard Tie::RefHash module provides a convenient workaround to this.

62.5 Postfix Dereference Syntax

Beginning in v5.20.0, a postfix syntax for using references is available. It behaves as described in Section 62.3.2 [Using References], page 1081, but instead of a prefixed sigil, a postfixed sigil-and-star is used.

For example:

```
  $r = \@a;
  @b = $r->@*; # equivalent to @$r or @{ $r }

  $r = [ 1, [ 2, 3 ], 4 ];
  $r->[1]->@*;  # equivalent to @{ $r->[1] }
```

This syntax must be enabled with `use feature 'postderef'`. It is experimental, and will warn by default unless `no warnings 'experimental::postderef'` is in effect.

Postfix dereference should work in all circumstances where block (circumfix) dereference worked, and should be entirely equivalent. This syntax allows dereferencing to be written and read entirely left-to-right. The following equivalencies are defined:

```
$sref->$*;  # same as  ${ $sref }
$aref->@*;  # same as  @{ $aref }
$aref->$#*; # same as $#{ $aref }
$href->%*;  # same as  %{ $href }
$cref->&*;  # same as  &{ $cref }
$gref->**;  # same as  *{ $gref }
```

Note especially that `$cref->&*` is *not* equivalent to `$cref->()`, and can serve different purposes.

Glob elements can be extracted through the postfix dereferencing feature:

```
$gref->*{SCALAR}; # same as *{ $gref }{SCALAR}
```

Postfix array and scalar dereferencing *can* be used in interpolating strings (double quotes or the `qq` operator), but only if the additional `postderef_qq` feature is enabled.

62.5.1 Postfix Reference Slicing

Value slices of arrays and hashes may also be taken with postfix dereferencing notation, with the following equivalencies:

```
$aref->@[ ... ];  # same as @$aref[ ... ]
$href->@{ ... };  # same as @$href{ ... }
```

Postfix key/value pair slicing, added in 5.20.0 and documented in Section 11.2.9.1 [the Key/Value Hash Slices section of perldata], page 85, also behaves as expected:

```
$aref->%[ ... ];  # same as %$aref[ ... ]
$href->%{ ... };  # same as %$href{ ... }
```

As with postfix array, postfix value slice dereferencing *can* be used in interpolating strings (double quotes or the `qq` operator), but only if the additional `postderef_qq feature` is enabled.

62.6 Assigning to References

Beginning in v5.22.0, the referencing operator can be assigned to. It performs an aliasing operation, so that the variable name referenced on the left-hand side becomes an alias for the thing referenced on the right-hand side:

```
\$a = \$b; # $a and $b now point to the same scalar
\&foo = \&bar; # foo() now means bar()
```

This syntax must be enabled with `use feature 'refaliasing'`. It is experimental, and will warn by default unless `no warnings 'experimental::refaliasing'` is in effect.

These forms may be assigned to, and cause the right-hand side to be evaluated in scalar context:

```
\$scalar
\@array
\%hash
\&sub
\my $scalar
\my @array
\my %hash
\state $scalar # or @array, etc.
\our $scalar   # etc.
\local $scalar # etc.
\local our $scalar # etc.
\$some_array[$index]
\$some_hash{$key}
\local $some_array[$index]
\local $some_hash{$key}
condition ? \$this : \$that[0] # etc.
```

Slicing operations and parentheses cause the right-hand side to be evaluated in list context:

```
\@array[5..7]
(\@array[5..7])
\(@array[5..7])
\@hash{'foo','bar'}
(\@hash{'foo','bar'})
\(@hash{'foo','bar'})
(\$scalar)
\($scalar)
\(my $scalar)
\my($scalar)
(\@array)
(\%hash)
(\&sub)
\(&sub)
\($foo, @bar, %baz)
(\$foo, \@bar, \%baz)
```

Each element on the right-hand side must be a reference to a datum of the right type. Parentheses immediately surrounding an array (and possibly also `my/state/our/local`) will make each element of the array an alias to the corresponding scalar referenced on the right-hand side:

```
\(@a) = \(@b); # @a and @b now have the same elements
\my(@a) = \(@b); # likewise
\(my @a) = \(@b); # likewise
push @a, 3; # but now @a has an extra element that @b lacks
\(@a) = (\$a, \$b, \$c); # @a now contains $a, $b, and $c
```

Combining that form with `local` and putting parentheses immediately around a hash are forbidden (because it is not clear what they should do):

```
\local(@array) = foo(); # WRONG
\(%hash)       = bar(); # wRONG
```

Assignment to references and non-references may be combined in lists and conditional ternary expressions, as long as the values on the right-hand side are the right type for each element on the left, though this may make for obfuscated code:

```
(my $tom, \my $dick, \my @harry) = (\1, \2, [1..3]);
# $tom is now \1
# $dick is now 2 (read-only)
# @harry is (1,2,3)

my $type = ref $thingy;
($type ? $type == 'ARRAY' ? \@foo : \$bar : $baz) = $thingy;
```

The `foreach` loop can also take a reference constructor for its loop variable, though the syntax is limited to one of the following, with an optional `my`, `state`, or `our` after the backslash:

```
\$s
\@a
\%h
\&c
```

No parentheses are permitted. This feature is particularly useful for arrays-of-arrays, or arrays-of-hashes:

```
foreach \my @a (@array_of_arrays) {
    frobnicate($a[0], $a[-1]);
}

foreach \my %h (@array_of_hashes) {
    $h{gelastic}++ if $h{type} == 'funny';
}
```

CAVEAT: Aliasing does not work correctly with closures. If you try to alias lexical variables from an inner subroutine or `eval`, the aliasing will only be visible within that inner sub, and will not affect the outer subroutine where the variables are declared. This bizarre behavior is subject to change.

62.7 SEE ALSO

Besides the obvious documents, source code can be instructive. Some pathological examples of the use of references can be found in the `t/op/ref.t` regression test in the Perl source directory.

See also Section 17.1 [perldsc NAME], page 246 and Section 39.1 [perllol NAME], page 725 for how to use references to create complex data structures, and Section 47.1 [perlootut NAME], page 786 and Section 46.1 [perlobj NAME], page 769 for how to use them to create objects.

63 perlreftut

63.1 NAME

perlreftut - Mark's very short tutorial about references

63.2 DESCRIPTION

One of the most important new features in Perl 5 was the capability to manage complicated data structures like multidimensional arrays and nested hashes. To enable these, Perl 5 introduced a feature called 'references', and using references is the key to managing complicated, structured data in Perl. Unfortunately, there's a lot of funny syntax to learn, and the main manual page can be hard to follow. The manual is quite complete, and sometimes people find that a problem, because it can be hard to tell what is important and what isn't.

Fortunately, you only need to know 10% of what's in the main page to get 90% of the benefit. This page will show you that 10%.

63.3 Who Needs Complicated Data Structures?

One problem that comes up all the time is needing a hash whose values are lists. Perl has hashes, of course, but the values have to be scalars; they can't be lists.

Why would you want a hash of lists? Let's take a simple example: You have a file of city and country names, like this:

```
Chicago, USA
Frankfurt, Germany
Berlin, Germany
Washington, USA
Helsinki, Finland
New York, USA
```

and you want to produce an output like this, with each country mentioned once, and then an alphabetical list of the cities in that country:

```
Finland: Helsinki.
Germany: Berlin, Frankfurt.
USA:  Chicago, New York, Washington.
```

The natural way to do this is to have a hash whose keys are country names. Associated with each country name key is a list of the cities in that country. Each time you read a line of input, split it into a country and a city, look up the list of cities already known to be in that country, and append the new city to the list. When you're done reading the input, iterate over the hash as usual, sorting each list of cities before you print it out.

If hash values couldn't be lists, you lose. You'd probably have to combine all the cities into a single string somehow, and then when time came to write the output, you'd have to break the string into a list, sort the list, and turn it back into a string. This is messy and error-prone. And it's frustrating, because Perl already has perfectly good lists that would solve the problem if only you could use them.

63.4 The Solution

By the time Perl 5 rolled around, we were already stuck with this design: Hash values must be scalars. The solution to this is references.

A reference is a scalar value that *refers to* an entire array or an entire hash (or to just about anything else). Names are one kind of reference that you're already familiar with. Think of the President of the United States: a messy, inconvenient bag of blood and bones. But to talk about him, or to represent him in a computer program, all you need is the easy, convenient scalar string "Barack Obama".

References in Perl are like names for arrays and hashes. They're Perl's private, internal names, so you can be sure they're unambiguous. Unlike "Barack Obama", a reference only refers to one thing, and you always know what it refers to. If you have a reference to an array, you can recover the entire array from it. If you have a reference to a hash, you can recover the entire hash. But the reference is still an easy, compact scalar value.

You can't have a hash whose values are arrays; hash values can only be scalars. We're stuck with that. But a single reference can refer to an entire array, and references are scalars, so you can have a hash of references to arrays, and it'll act a lot like a hash of arrays, and it'll be just as useful as a hash of arrays.

We'll come back to this city-country problem later, after we've seen some syntax for managing references.

63.5 Syntax

There are just two ways to make a reference, and just two ways to use it once you have it.

63.5.1 Making References

63.5.1.1 Make Rule 1

If you put a \ in front of a variable, you get a reference to that variable.

```
$aref = \@array;         # $aref now holds a reference to @array
$href = \%hash;          # $href now holds a reference to %hash
$sref = \$scalar;        # $sref now holds a reference to $scalar
```

Once the reference is stored in a variable like $aref or $href, you can copy it or store it just the same as any other scalar value:

```
$xy = $aref;             # $xy now holds a reference to @array
$p[3] = $href;           # $p[3] now holds a reference to %hash
$z = $p[3];              # $z now holds a reference to %hash
```

These examples show how to make references to variables with names. Sometimes you want to make an array or a hash that doesn't have a name. This is analogous to the way you like to be able to use the string "\n" or the number 80 without having to store it in a named variable first.

Make Rule 2

`[ ITEMS ]` makes a new, anonymous array, and returns a reference to that array. `{ ITEMS }` makes a new, anonymous hash, and returns a reference to that hash.

```
$aref = [ 1, "foo", undef, 13 ];
```

```
# $aref now holds a reference to an array

$href = { APR => 4, AUG => 8 };
# $href now holds a reference to a hash
```

The references you get from rule 2 are the same kind of references that you get from rule 1:

```
# This:
$aref = [ 1, 2, 3 ];

# Does the same as this:
@array = (1, 2, 3);
$aref = \@array;
```

The first line is an abbreviation for the following two lines, except that it doesn't create the superfluous array variable `@array`.

If you write just `[]`, you get a new, empty anonymous array. If you write just `{}`, you get a new, empty anonymous hash.

63.5.2 Using References

What can you do with a reference once you have it? It's a scalar value, and we've seen that you can store it as a scalar and get it back again just like any scalar. There are just two more ways to use it:

63.5.2.1 Use Rule 1

You can always use an array reference, in curly braces, in place of the name of an array. For example, `@{$aref}` instead of `@array`.

Here are some examples of that:

Arrays:

```
@a              @{$aref}                An array
reverse @a      reverse @{$aref}        Reverse the array
$a[3]           ${$aref}[3]             An element of the array
$a[3] = 17;     ${$aref}[3] = 17        Assigning an element
```

On each line are two expressions that do the same thing. The left-hand versions operate on the array `@a`. The right-hand versions operate on the array that is referred to by `$aref`. Once they find the array they're operating on, both versions do the same things to the arrays.

Using a hash reference is *exactly* the same:

```
%h              %{$href}              A hash
keys %h         keys %{$href}         Get the keys from the hash
$h{'red'}       ${$href}{'red'}       An element of the hash
$h{'red'} = 17  ${$href}{'red'} = 17  Assigning an element
```

Whatever you want to do with a reference, **Use Rule 1** tells you how to do it. You just write the Perl code that you would have written for doing the same thing to a regular array or hash, and then replace the array or hash name with `{$reference}`. "How do I loop over an array when all I have is a reference?" Well, to loop over an array, you would write

```
for my $element (@array) {
   ...
}
```

so replace the array name, @array, with the reference:

```
for my $element (@{$aref}) {
   ...
}
```

"How do I print out the contents of a hash when all I have is a reference?" First write the code for printing out a hash:

```
for my $key (keys %hash) {
  print "$key => $hash{$key}\n";
}
```

And then replace the hash name with the reference:

```
for my $key (keys %{$href}) {
  print "$key => ${$href}{$key}\n";
}
```

63.5.2.2 Use Rule 2

Use Rule 1 is all you really need, because it tells you how to do absolutely everything you ever need to do with references. But the most common thing to do with an array or a hash is to extract a single element, and the **Use Rule 1** notation is cumbersome. So there is an abbreviation.

${$aref}[3] is too hard to read, so you can write $aref->[3] instead.

${$href}{red} is too hard to read, so you can write $href->{red} instead.

If $aref holds a reference to an array, then $aref->[3] is the fourth element of the array. Don't confuse this with $aref[3], which is the fourth element of a totally different array, one deceptively named @aref. $aref and @aref are unrelated the same way that $item and @item are.

Similarly, $href->{'red'} is part of the hash referred to by the scalar variable $href, perhaps even one with no name. $href{'red'} is part of the deceptively named %href hash. It's easy to forget to leave out the ->, and if you do, you'll get bizarre results when your program gets array and hash elements out of totally unexpected hashes and arrays that weren't the ones you wanted to use.

63.5.3 An Example

Let's see a quick example of how all this is useful.

First, remember that [1, 2, 3] makes an anonymous array containing (1, 2, 3), and gives you a reference to that array.

Now think about

```
@a = ( [1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]
     );
```

@a is an array with three elements, and each one is a reference to another array.

`$a[1]` is one of these references. It refers to an array, the array containing `(4, 5, 6)`, and because it is a reference to an array, **Use Rule 2** says that we can write `$a[1]->[2]` to get the third element from that array. `$a[1]->[2]` is the 6. Similarly, `$a[0]->[1]` is the 2. What we have here is like a two-dimensional array; you can write `$a[ROW]->[COLUMN]` to get or set the element in any row and any column of the array.

The notation still looks a little cumbersome, so there's one more abbreviation:

63.5.4 Arrow Rule

In between two **subscripts**, the arrow is optional.

Instead of `$a[1]->[2]`, we can write `$a[1][2]`; it means the same thing. Instead of `$a[0]->[1]` = 23, we can write `$a[0][1]` = 23; it means the same thing.

Now it really looks like two-dimensional arrays!

You can see why the arrows are important. Without them, we would have had to write `${$a[1]}[2]` instead of `$a[1][2]`. For three-dimensional arrays, they let us write `$x[2][3][5]` instead of the unreadable `${${$x[2]}[3]}[5]`.

63.6 Solution

Here's the answer to the problem I posed earlier, of reformatting a file of city and country names.

```
 1   my %table;

 2   while (<>) {
 3    chomp;
 4     my ($city, $country) = split /, /;
 5     $table{$country} = [] unless exists $table{$country};
 6     push @{$table{$country}}, $city;
 7   }

 8   foreach $country (sort keys %table) {
 9     print "$country: ";
10     my @cities = @{$table{$country}};
11     print join ', ', sort @cities;
12     print ".\n";
13   }
```

The program has two pieces: Lines 2–7 read the input and build a data structure, and lines 8-13 analyze the data and print out the report. We're going to have a hash, `%table`, whose keys are country names, and whose values are references to arrays of city names. The data structure will look like this:

```
   %table
+-------+---+
|       |   |   +-----------+--------+
|Germany| *---->| Frankfurt | Berlin |
|       |   |   +-----------+--------+
+-------+---+
|       |   |   +----------+
|Finland| *---->| Helsinki |
|       |   |   +----------+
+-------+---+
|       |   |   +---------+------------+----------+
|  USA  | *---->| Chicago | Washington | New York |
|       |   |   +---------+------------+----------+
+-------+---+
```

We'll look at output first. Supposing we already have this structure, how do we print it out?

```
 8   foreach $country (sort keys %table) {
 9     print "$country: ";
10     my @cities = @{$table{$country}};
11     print join ', ', sort @cities;
12     print ".\n";
13   }
```

`%table` is an ordinary hash, and we get a list of keys from it, sort the keys, and loop over the keys as usual. The only use of references is in line 10. `$table{$country}` looks up the key `$country` in the hash and gets the value, which is a reference to an array of cities in that country. **Use Rule 1** says that we can recover the array by saying `@{$table{$country}}`. Line 10 is just like

```
@cities = @array;
```

except that the name `array` has been replaced by the reference `{$table{$country}}`. The `@` tells Perl to get the entire array. Having gotten the list of cities, we sort it, join it, and print it out as usual.

Lines 2-7 are responsible for building the structure in the first place. Here they are again:

```
2   while (<>) {
3    chomp;
4     my ($city, $country) = split /, /;
5     $table{$country} = [] unless exists $table{$country};
6     push @{$table{$country}}, $city;
7   }
```

Lines 2-4 acquire a city and country name. Line 5 looks to see if the country is already present as a key in the hash. If it's not, the program uses the `[]` notation (**Make Rule 2**) to manufacture a new, empty anonymous array of cities, and installs a reference to it into the hash under the appropriate key.

Line 6 installs the city name into the appropriate array. `$table{$country}` now holds a reference to the array of cities seen in that country so far. Line 6 is exactly like

```
push @array, $city;
```

except that the name `array` has been replaced by the reference `{$table{$country}}`. The `push` adds a city name to the end of the referred-to array.

There's one fine point I skipped. Line 5 is unnecessary, and we can get rid of it.

```
2   while (<>) {
3    chomp;
4     my ($city, $country) = split /, /;
5   ####  $table{$country} = [] unless exists $table{$country};
6     push @{$table{$country}}, $city;
7   }
```

If there's already an entry in `%table` for the current `$country`, then nothing is different. Line 6 will locate the value in `$table{$country}`, which is a reference to an array, and push `$city` into the array. But what does it do when `$country` holds a key, say `Greece`, that is not yet in `%table`?

This is Perl, so it does the exact right thing. It sees that you want to push `Athens` onto an array that doesn't exist, so it helpfully makes a new, empty, anonymous array for you, installs it into `%table`, and then pushes `Athens` onto it. This is called 'autovivification'– bringing things to life automatically. Perl saw that the key wasn't in the hash, so it created a new hash entry automatically. Perl saw that you wanted to use the hash value as an array, so it created a new empty array and installed a reference to it in the hash automatically. And as usual, Perl made the array one element longer to hold the new city name.

63.7 The Rest

I promised to give you 90% of the benefit with 10% of the details, and that means I left out 90% of the details. Now that you have an overview of the important parts, it should be easier to read the Section 62.1 [perlref NAME], page 1077 manual page, which discusses 100% of the details.

Some of the highlights of Section 62.1 [perlref NAME], page 1077:

- You can make references to anything, including scalars, functions, and other references.
- In **Use Rule 1**, you can omit the curly brackets whenever the thing inside them is an atomic scalar variable like `$aref`. For example, `@$aref` is the same as `@{$aref}`, and `$$aref[1]` is the same as `${$aref}[1]`. If you're just starting out, you may want to adopt the habit of always including the curly brackets.
- This doesn't copy the underlying array:

  ```
  $aref2 = $aref1;
  ```

  You get two references to the same array. If you modify `$aref1->[23]` and then look at `$aref2->[23]` you'll see the change.

  To copy the array, use

  ```
  $aref2 = [@{$aref1}];
  ```

  This uses `[...]` notation to create a new anonymous array, and `$aref2` is assigned a reference to the new array. The new array is initialized with the contents of the array referred to by `$aref1`.

  Similarly, to copy an anonymous hash, you can use

  ```
  $href2 = {%{$href1}};
  ```

- To see if a variable contains a reference, use the `ref` function. It returns true if its argument is a reference. Actually it's a little better than that: It returns `HASH` for hash references and `ARRAY` for array references.

- If you try to use a reference like a string, you get strings like

  ```
  ARRAY(0x80f5dec)   or    HASH(0x826afc0)
  ```

  If you ever see a string that looks like this, you'll know you printed out a reference by mistake.

  A side effect of this representation is that you can use `eq` to see if two references refer to the same thing. (But you should usually use `==` instead because it's much faster.)
- You can use a string as if it were a reference. If you use the string `"foo"` as an array reference, it's taken to be a reference to the array `@foo`. This is called a *soft reference* or *symbolic reference*. The declaration `use strict 'refs'` disables this feature, which can cause all sorts of trouble if you use it by accident.

You might prefer to go on to Section 39.1 [perllol NAME], page 725 instead of Section 62.1 [perlref NAME], page 1077; it discusses lists of lists and multidimensional arrays in detail. After that, you should move on to Section 17.1 [perldsc NAME], page 246; it's a Data Structure Cookbook that shows recipes for using and printing out arrays of hashes, hashes of arrays, and other kinds of data.

63.8 Summary

Everyone needs compound data structures, and in Perl the way you get them is with references. There are four important rules for managing references: Two for making references and two for using them. Once you know these rules you can do most of the important things you need to do with references.

63.9 Credits

Author: Mark Jason Dominus, Plover Systems (`mjd-perl-ref+@plover.com`)

This article originally appeared in *The Perl Journal* (http://www.tpj.com/) volume 3, #2. Reprinted with permission.

The original title was *Understand References Today*.

63.9.1 Distribution Conditions

64 perlreguts

64.1 NAME

perlreguts - Description of the Perl regular expression engine.

64.2 DESCRIPTION

This document is an attempt to shine some light on the guts of the regex engine and how it works. The regex engine represents a significant chunk of the perl codebase, but is relatively poorly understood. This document is a meagre attempt at addressing this situation. It is derived from the author's experience, comments in the source code, other papers on the regex engine, feedback on the perl5-porters mail list, and no doubt other places as well.

NOTICE! It should be clearly understood that the behavior and structures discussed in this represents the state of the engine as the author understood it at the time of writing. It is **NOT** an API definition, it is purely an internals guide for those who want to hack the regex engine, or understand how the regex engine works. Readers of this document are expected to understand perl's regex syntax and its usage in detail. If you want to learn about the basics of Perl's regular expressions, see Section 58.1 [perlre NAME], page 989. And if you want to replace the regex engine with your own, see Section 59.1 [perlreapi NAME], page 1032.

64.3 OVERVIEW

64.3.1 A quick note on terms

There is some debate as to whether to say "regexp" or "regex". In this document we will use the term "regex" unless there is a special reason not to, in which case we will explain why.

When speaking about regexes we need to distinguish between their source code form and their internal form. In this document we will use the term "pattern" when we speak of their textual, source code form, and the term "program" when we speak of their internal representation. These correspond to the terms *S-regex* and *B-regex* that Mark Jason Dominus employs in his paper on "Rx" ([1] in Section 64.9 [REFERENCES], page 1114).

64.3.2 What is a regular expression engine?

A regular expression engine is a program that takes a set of constraints specified in a mini-language, and then applies those constraints to a target string, and determines whether or not the string satisfies the constraints. See Section 58.1 [perlre NAME], page 989 for a full definition of the language.

In less grandiose terms, the first part of the job is to turn a pattern into something the computer can efficiently use to find the matching point in the string, and the second part is performing the search itself.

To do this we need to produce a program by parsing the text. We then need to execute the program to find the point in the string that matches. And we need to do the whole thing efficiently.

64.3.3 Structure of a Regexp Program

64.3.3.1 High Level

Although it is a bit confusing and some people object to the terminology, it is worth taking a look at a comment that has been in `regexp.h` for years:

This is essentially a linear encoding of a nondeterministic finite-state machine (aka syntax charts or "railroad normal form" in parsing technology).

The term "railroad normal form" is a bit esoteric, with "syntax diagram/charts", or "railroad diagram/charts" being more common terms. Nevertheless it provides a useful mental image of a regex program: each node can be thought of as a unit of track, with a single entry and in most cases a single exit point (there are pieces of track that fork, but statistically not many), and the whole forms a layout with a single entry and single exit point. The matching process can be thought of as a car that moves along the track, with the particular route through the system being determined by the character read at each possible connector point. A car can fall off the track at any point but it may only proceed as long as it matches the track.

Thus the pattern `/foo(?:\w+|\d+|\s+)bar/` can be thought of as the following chart:

```
     [start]
        |
      <foo>
        |
  +-----+-----+
  |     |     |
<\w+> <\d+> <\s+>
  |     |     |
  +-----+-----+
        |
      <bar>
        |
      [end]
```

The truth of the matter is that perl's regular expressions these days are much more complex than this kind of structure, but visualising it this way can help when trying to get your bearings, and it matches the current implementation pretty closely.

To be more precise, we will say that a regex program is an encoding of a graph. Each node in the graph corresponds to part of the original regex pattern, such as a literal string or a branch, and has a pointer to the nodes representing the next component to be matched. Since "node" and "opcode" already have other meanings in the perl source, we will call the nodes in a regex program "regops".

The program is represented by an array of `regnode` structures, one or more of which represent a single regop of the program. Struct `regnode` is the smallest struct needed, and has a field structure which is shared with all the other larger structures.

The "next" pointers of all regops except `BRANCH` implement concatenation; a "next" pointer with a `BRANCH` on both ends of it is connecting two alternatives. [Here we have one of the subtle syntax dependencies: an individual `BRANCH` (as opposed to a collection of them) is never concatenated with anything because of operator precedence.]

The operand of some types of regop is a literal string; for others, it is a regop leading into a sub-program. In particular, the operand of a `BRANCH` node is the first regop of the branch.

NOTE: As the railroad metaphor suggests, this is **not** a tree structure: the tail of the branch connects to the thing following the set of `BRANCH`es. It is a like a single line of railway track that splits as it goes into a station or railway yard and rejoins as it comes out the other side.

64.3.3.2 Regops

The base structure of a regop is defined in `regexp.h` as follows:

```
struct regnode {
    U8  flags;    /* Various purposes, sometimes overridden */
    U8  type;     /* Opcode value as specified by regnodes.h */
    U16 next_off; /* Offset in size regnode */
};
```

Other larger `regnode`-like structures are defined in `regcomp.h`. They are almost like subclasses in that they have the same fields as `regnode`, with possibly additional fields following in the structure, and in some cases the specific meaning (and name) of some of base fields are overridden. The following is a more complete description.

`regnode_1`
`regnode_2`

`regnode_1` structures have the same header, followed by a single four-byte argument; `regnode_2` structures contain two two-byte arguments instead:

```
regnode_1                U32 arg1;
regnode_2                U16 arg1;  U16 arg2;
```

`regnode_string`

`regnode_string` structures, used for literal strings, follow the header with a one-byte length and then the string data. Strings are padded on the end with zero bytes so that the total length of the node is a multiple of four bytes:

```
regnode_string           char string[1];
                         U8 str_len; /* overrides flags */
```

`regnode_charclass`

Bracketed character classes are represented by `regnode_charclass` structures, which have a four-byte argument and then a 32-byte (256-bit) bitmap indicating which characters in the Latin1 range are included in the class.

```
regnode_charclass        U32 arg1;
                         char bitmap[ANYOF_BITMAP_SIZE];
```

Various flags whose names begin with `ANYOF_` are used for special situations. Above Latin1 matches and things not known until run-time are stored in Section 64.5.2.1 [Perl's pprivate structure], page 1113.

`regnode_charclass_posixl`

There is also a larger form of a char class structure used to represent POSIX char classes under `/l` matching, called `regnode_charclass_posixl` which has an additional 32-bit bitmap indicating which POSIX char classes have been included.

```
regnode_charclass_posixl U32 arg1;
                         char bitmap[ANYOF_BITMAP_SIZE];
                         U32 classflags;
```

`regnodes.h` defines an array called `regarglen[]` which gives the size of each opcode in units of `size regnode` (4-byte). A macro is used to calculate the size of an `EXACT` node based on its `str_len` field.

The regops are defined in `regnodes.h` which is generated from `regcomp.sym` by `regcomp.pl`. Currently the maximum possible number of distinct regops is restricted to 256, with about a quarter already used.

A set of macros makes accessing the fields easier and more consistent. These include `OP()`, which is used to determine the type of a `regnode`-like structure; `NEXT_OFF()`, which is the offset to the next node (more on this later); `ARG()`, `ARG1()`, `ARG2()`, `ARG_SET()`, and equivalents for reading and setting the arguments; and `STR_LEN()`, `STRING()` and `OPERAND()` for manipulating strings and regop bearing types.

64.3.3.3 What regop is next?

There are three distinct concepts of "next" in the regex engine, and it is important to keep them clear.

- There is the "next regnode" from a given regnode, a value which is rarely useful except that sometimes it matches up in terms of value with one of the others, and that sometimes the code assumes this to always be so.

- There is the "next regop" from a given regop/regnode. This is the regop physically located after the current one, as determined by the size of the current regop. This is often useful, such as when dumping the structure we use this order to traverse. Sometimes the code assumes that the "next regnode" is the same as the "next regop", or in other words assumes that the sizeof a given regop type is always going to be one regnode large.

- There is the "regnext" from a given regop. This is the regop which is reached by jumping forward by the value of `NEXT_OFF()`, or in a few cases for longer jumps by the `arg1` field of the `regnode_1` structure. The subroutine `regnext()` handles this transparently. This is the logical successor of the node, which in some cases, like that of the `BRANCH` regop, has special meaning.

64.4 Process Overview

Broadly speaking, performing a match of a string against a pattern involves the following steps:

A. Compilation

1. Parsing for size
2. Parsing for construction
3. Peep-hole optimisation and analysis

B. Execution

4. Start position and no-match optimisations
5. Program execution

Where these steps occur in the actual execution of a perl program is determined by whether the pattern involves interpolating any string variables. If interpolation occurs, then compilation happens at run time. If it does not, then compilation is performed at compile time. (The `/o` modifier changes this, as does `qr//` to a certain extent.) The engine doesn't really care that much.

64.4.1 Compilation

This code resides primarily in `regcomp.c`, along with the header files `regcomp.h`, `regexp.h` and `regnodes.h`.

Compilation starts with `pregcomp()`, which is mostly an initialisation wrapper which farms work out to two other routines for the heavy lifting: the first is `reg()`, which is the start point for parsing; the second, `study_chunk()`, is responsible for optimisation.

Initialisation in `pregcomp()` mostly involves the creation and data-filling of a special structure, `RExC_state_t` (defined in `regcomp.c`). Almost all internally-used routines in `regcomp.h` take a pointer to one of these structures as their first argument, with the name `pRExC_state`. This structure is used to store the compilation state and contains many fields. Likewise there are many macros which operate on this variable: anything that looks like `RExC_xxxx` is a macro that operates on this pointer/structure.

64.4.1.1 Parsing for size

In this pass the input pattern is parsed in order to calculate how much space is needed for each regop we would need to emit. The size is also used to determine whether long jumps will be required in the program.

This stage is controlled by the macro `SIZE_ONLY` being set.

The parse proceeds pretty much exactly as it does during the construction phase, except that most routines are short-circuited to change the size field `RExC_size` and not do anything else.

64.4.1.2 Parsing for construction

Once the size of the program has been determined, the pattern is parsed again, but this time for real. Now `SIZE_ONLY` will be false, and the actual construction can occur.

`reg()` is the start of the parse process. It is responsible for parsing an arbitrary chunk of pattern up to either the end of the string, or the first closing parenthesis it encounters in the pattern. This means it can be used to parse the top-level regex, or any section inside of a grouping parenthesis. It also handles the "special parens" that perl's regexes have. For instance when parsing `/x(?:foo)y/` `reg()` will at one point be called to parse from the "?" symbol up to and including the ")".

Additionally, reg() is responsible for parsing the one or more branches from the pattern, and for "finishing them off" by correctly setting their next pointers. In order to do the parsing, it repeatedly calls out to regbranch(), which is responsible for handling up to the first | symbol it sees.

regbranch() in turn calls regpiece() which handles "things" followed by a quantifier. In order to parse the "things", regatom() is called. This is the lowest level routine, which parses out constant strings, character classes, and the various special symbols like $. If regatom() encounters a "(" character it in turn calls reg().

The routine regtail() is called by both reg() and regbranch() in order to "set the tail pointer" correctly. When executing and we get to the end of a branch, we need to go to the node following the grouping parens. When parsing, however, we don't know where the end will be until we get there, so when we do we must go back and update the offsets as appropriate. regtail is used to make this easier.

A subtlety of the parsing process means that a regex like /foo/ is originally parsed into an alternation with a single branch. It is only afterwards that the optimiser converts single branch alternations into the simpler form.

64.4.1.3 Parse Call Graph and a Grammar

The call graph looks like this:

```
reg()                        # parse a top level regex, or inside of
                             # parens
    regbranch()              # parse a single branch of an alternation
        regpiece()           # parse a pattern followed by a quantifier
            regatom()        # parse a simple pattern
                regclass()   #   used to handle a class
                reg()        #   used to handle a parenthesised
                             #   subpattern
                ....
        ...
        regtail()            # finish off the branch
    ...
    regtail()                # finish off the branch sequence. Tie each
                             # branch's tail to the tail of the
                             # sequence
                             # (NEW) In Debug mode this is
                             # regtail_study().
```

A grammar form might be something like this:

```
atom  : constant | class
quant : '*' | '+' | '?' | '{min,max}'
_branch: piece
       | piece _branch
       | nothing
branch: _branch
      | _branch '|' branch
group : '(' branch ')'
```

```
    _piece: atom | group
    piece : _piece
          | _piece quant
```

64.4.1.4 Parsing complications

The implication of the above description is that a pattern containing nested parentheses will result in a call graph which cycles through `reg()`, `regbranch()`, `regpiece()`, `regatom()`, `reg()`, `regbranch()` *etc* multiple times, until the deepest level of nesting is reached. All the above routines return a pointer to a `regnode`, which is usually the last regnode added to the program. However, one complication is that reg() returns NULL for parsing `(?:)` syntax for embedded modifiers, setting the flag `TRYAGAIN`. The `TRYAGAIN` propagates upwards until it is captured, in some cases by `regatom()`, but otherwise unconditionally by `regbranch()`. Hence it will never be returned by `regbranch()` to `reg()`. This flag permits patterns such as `(?i)+` to be detected as errors (*Quantifier follows nothing in regex; marked by* `<--` *HERE in m/(?i)*+ `<--` *HERE /*).

Another complication is that the representation used for the program differs if it needs to store Unicode, but it's not always possible to know for sure whether it does until midway through parsing. The Unicode representation for the program is larger, and cannot be matched as efficiently. (See Section 64.5.1 [Unicode and Localisation Support], page 1112 below for more details as to why.) If the pattern contains literal Unicode, it's obvious that the program needs to store Unicode. Otherwise, the parser optimistically assumes that the more efficient representation can be used, and starts sizing on this basis. However, if it then encounters something in the pattern which must be stored as Unicode, such as an `\x{...}` escape sequence representing a character literal, then this means that all previously calculated sizes need to be redone, using values appropriate for the Unicode representation. Currently, all regular expression constructions which can trigger this are parsed by code in `regatom()`.

To avoid wasted work when a restart is needed, the sizing pass is abandoned - `regatom()` immediately returns NULL, setting the flag `RESTART_UTF8`. (This action is encapsulated using the macro `REQUIRE_UTF8`.) This restart request is propagated up the call chain in a similar fashion, until it is "caught" in `Perl_re_op_compile()`, which marks the pattern as containing Unicode, and restarts the sizing pass. It is also possible for constructions within run-time code blocks to turn out to need Unicode representation., which is signalled by `S_compile_runtime_code()` returning false to `Perl_re_op_compile()`.

The restart was previously implemented using a `longjmp` in `regatom()` back to a `setjmp` in `Perl_re_op_compile()`, but this proved to be problematic as the latter is a large function containing many automatic variables, which interact badly with the emergent control flow of `setjmp`.

64.4.1.5 Debug Output

In the 5.9.x development version of perl you can `use re Debug => 'PARSE'` to see some trace information about the parse process. We will start with some simple patterns and build up to more complex patterns.

So when we parse `/foo/` we see something like the following table. The left shows what is being parsed, and the number indicates where the next regop would go. The stuff on the right is the trace output of the graph. The names are chosen to be short to make it less dense on the screen. 'tsdy' is a special form of `regtail()` which does some extra analysis.

```
>foo<             1    reg
                         brnc
                           piec
                             atom
><                4      tsdy~ EXACT <foo> (EXACT) (1)
                             ~ attach to END (3) offset to 2
```

The resulting program then looks like:

```
1: EXACT <foo>(3)
3: END(0)
```

As you can see, even though we parsed out a branch and a piece, it was ultimately only an atom. The final program shows us how things work. We have an `EXACT` regop, followed by an `END` regop. The number in parens indicates where the `regnext` of the node goes. The `regnext` of an `END` regop is unused, as `END` regops mean we have successfully matched. The number on the left indicates the position of the regop in the regnode array.

Now let's try a harder pattern. We will add a quantifier, so now we have the pattern `/foo+/`. We will see that `regbranch()` calls `regpiece()` twice.

```
>foo+<            1    reg
                         brnc
                           piec
                             atom
>o+<              3        piec
                             atom
><                6        tail~ EXACT <fo> (1)
                  7      tsdy~ EXACT <fo> (EXACT) (1)
                             ~ PLUS (END) (3)
                             ~ attach to END (6) offset to 3
```

And we end up with the program:

```
1: EXACT <fo>(3)
3: PLUS(6)
4:   EXACT <o>(0)
6: END(0)
```

Now we have a special case. The `EXACT` regop has a `regnext` of 0. This is because if it matches it should try to match itself again. The `PLUS` regop handles the actual failure of the `EXACT` regop and acts appropriately (going to regnode 6 if the `EXACT` matched at least once, or failing if it didn't).

Now for something much more complex: `/x(?:foo*|b[a][rR])(foo|bar)$/`

```
>x(?:foo*|b...    1    reg
                         brnc
                           piec
                             atom
>(?:foo*|b[...    3        piec
                             atom
>?:foo*|b[a...                 reg
>foo*|b[a][...                   brnc
                                   piec
                                     atom
>o*|b[a][rR...    5                piec
                                     atom
>|b[a][rR])...    8                tail~ EXACT <fo> (3)
>b[a][rR])(...    9              brnc
                 10                piec
                                     atom
>[a][rR])(f...   12                piec
                                     atom
>a][rR])(fo...                         clas
>[rR])(foo|...   14                tail~ EXACT <b> (10)
                                   piec
                                     atom
>rR])(foo|b...                         clas
>)(foo|bar)...   25                tail~ EXACT <a> (12)
                                 tail~ BRANCH (3)
                 26              tsdy~ BRANCH (END) (9)
                                     ~ attach to TAIL (25) offset to 16
                                 tsdy~ EXACT <fo> (EXACT) (4)
                                     ~ STAR (END) (6)
                                     ~ attach to TAIL (25) offset to 19
                                 tsdy~ EXACT <b> (EXACT) (10)
                                     ~ EXACT <a> (EXACT) (12)
                                     ~ ANYOF[Rr] (END) (14)
                                     ~ attach to TAIL (25) offset to 11
>(foo|bar)$<               tail~ EXACT <x> (1)
                           piec
                             atom
>foo|bar)$<                    reg
                 28              brnc
                                   piec
                                     atom
>|bar)$<         31              tail~ OPEN1 (26)
>bar)$<                          brnc
                 32                piec
                                     atom
>)$<             34              tail~ BRANCH (28)
                 36              tsdy~ BRANCH (END) (31)
                                    ~ attach to CLOSE1 (34) offset to 3
                                 tsdy~ EXACT <foo> (EXACT) (29)
                                    ~ attach to CLOSE1 (34) offset to 5
                                 tsdy~ EXACT <bar> (EXACT) (32)
                                    ~ attach to CLOSE1 (34) offset to 2
>$<                        tail~ BRANCH (3)
```

```
                               ~ BRANCH (9)
                               ~ TAIL (25)
                           piec
                             atom
><               37        tail~ OPEN1 (26)
                               ~ BRANCH (28)
                               ~ BRANCH (31)
                               ~ CLOSE1 (34)
                 38      tsdy~ EXACT <x> (EXACT) (1)
                             ~ BRANCH (END) (3)
                             ~ BRANCH (END) (9)
                             ~ TAIL (END) (25)
                             ~ OPEN1 (END) (26)
                             ~ BRANCH (END) (28)
                             ~ BRANCH (END) (31)
                             ~ CLOSE1 (END) (34)
                             ~ EOL (END) (36)
                             ~ attach to END (37) offset to 1
```

Resulting in the program

```
 1: EXACT <x>(3)
 3: BRANCH(9)
 4:   EXACT <fo>(6)
 6:   STAR(26)
 7:     EXACT <o>(0)
 9: BRANCH(25)
10:   EXACT <ba>(14)
12:   OPTIMIZED (2 nodes)
14:   ANYOF[Rr](26)
25: TAIL(26)
26: OPEN1(28)
28:   TRIE-EXACT(34)
      [StS:1 Wds:2 Cs:6 Uq:5 #Sts:7 Mn:3 Mx:3 Stcls:bf]
        <foo>
        <bar>
30:   OPTIMIZED (4 nodes)
34: CLOSE1(36)
36: EOL(37)
37: END(0)
```

Here we can see a much more complex program, with various optimisations in play. At regnode 10 we see an example where a character class with only one character in it was turned into an `EXACT` node. We can also see where an entire alternation was turned into a `TRIE-EXACT` node. As a consequence, some of the regnodes have been marked as optimised away. We can see that the `$` symbol has been converted into an `EOL` regop, a special piece of code that looks for `\n` or the end of the string.

The next pointer for `BRANCH`es is interesting in that it points at where execution should go if the branch fails. When executing, if the engine tries to traverse from a branch to a `regnext` that isn't a branch then the engine will know that the entire set of branches has failed.

64.4.1.6 Peep-hole Optimisation and Analysis

The regular expression engine can be a weighty tool to wield. On long strings and complex patterns it can end up having to do a lot of work to find a match, and even more to decide that no match is possible. Consider a situation like the following pattern.

```
'ababababababababababab' =~ /(a|b)*z/
```

The `(a|b)*` part can match at every char in the string, and then fail every time because there is no `z` in the string. So obviously we can avoid using the regex engine unless there is a `z` in the string. Likewise in a pattern like:

```
/foo(\w+)bar/
```

In this case we know that the string must contain a `foo` which must be followed by `bar`. We can use Fast Boyer-Moore matching as implemented in `fbm_instr()` to find the location of these strings. If they don't exist then we don't need to resort to the much more expensive regex engine. Even better, if they do exist then we can use their positions to reduce the search space that the regex engine needs to cover to determine if the entire pattern matches.

There are various aspects of the pattern that can be used to facilitate optimisations along these lines:

- anchored fixed strings
- floating fixed strings
- minimum and maximum length requirements
- start class
- Beginning/End of line positions

Another form of optimisation that can occur is the post-parse "peep-hole" optimisation, where inefficient constructs are replaced by more efficient constructs. The `TAIL` regops which are used during parsing to mark the end of branches and the end of groups are examples of this. These regops are used as place-holders during construction and "always match" so they can be "optimised away" by making the things that point to the `TAIL` point to the thing that `TAIL` points to, thus "skipping" the node.

Another optimisation that can occur is that of "`EXACT` merging" which is where two consecutive `EXACT` nodes are merged into a single regop. An even more aggressive form of this is that a branch sequence of the form `EXACT BRANCH ... EXACT` can be converted into a `TRIE-EXACT` regop.

All of this occurs in the routine `study_chunk()` which uses a special structure `scan_data_t` to store the analysis that it has performed, and does the "peep-hole" optimisations as it goes.

The code involved in `study_chunk()` is extremely cryptic. Be careful. :-)

64.4.2 Execution

Execution of a regex generally involves two phases, the first being finding the start point in the string where we should match from, and the second being running the regop interpreter.

If we can tell that there is no valid start point then we don't bother running the interpreter at all. Likewise, if we know from the analysis phase that we cannot detect a short-cut to the start position, we go straight to the interpreter.

The two entry points are `re_intuit_start()` and `pregexec()`. These routines have a somewhat incestuous relationship with overlap between their functions, and `pregexec()` may even call `re_intuit_start()` on its own. Nevertheless other parts of the perl source code may call into either, or both.

Execution of the interpreter itself used to be recursive, but thanks to the efforts of Dave Mitchell in the 5.9.x development track, that has changed: now an internal stack is maintained on the heap and the routine is fully iterative. This can make it tricky as the code is quite conservative about what state it stores, with the result that two consecutive lines in the code can actually be running in totally different contexts due to the simulated recursion.

64.4.2.1 Start position and no-match optimisations

`re_intuit_start()` is responsible for handling start points and no-match optimisations as determined by the results of the analysis done by `study_chunk()` (and described in Section 64.4.1.6 [Peep-hole Optimisation and Analysis], page 1110).

The basic structure of this routine is to try to find the start- and/or end-points of where the pattern could match, and to ensure that the string is long enough to match the pattern. It tries to use more efficient methods over less efficient methods and may involve considerable cross-checking of constraints to find the place in the string that matches. For instance it may try to determine that a given fixed string must be not only present but a certain number of chars before the end of the string, or whatever.

It calls several other routines, such as `fbm_instr()` which does Fast Boyer Moore matching and `find_byclass()` which is responsible for finding the start using the first mandatory regop in the program.

When the optimisation criteria have been satisfied, `reg_try()` is called to perform the match.

64.4.2.2 Program execution

`pregexec()` is the main entry point for running a regex. It contains support for initialising the regex interpreter's state, running `re_intuit_start()` if needed, and running the interpreter on the string from various start positions as needed. When it is necessary to use the regex interpreter `pregexec()` calls `regtry()`.

`regtry()` is the entry point into the regex interpreter. It expects as arguments a pointer to a `regmatch_info` structure and a pointer to a string. It returns an integer 1 for success and a 0 for failure. It is basically a set-up wrapper around `regmatch()`.

`regmatch` is the main "recursive loop" of the interpreter. It is basically a giant switch statement that implements a state machine, where the possible states are the regops themselves, plus a number of additional intermediate and failure states. A few of the states are implemented as subroutines but the bulk are inline code.

64.5 MISCELLANEOUS

64.5.1 Unicode and Localisation Support

When dealing with strings containing characters that cannot be represented using an eight-bit character set, perl uses an internal representation that is a permissive version of Unicode's UTF-8 encoding[2]. This uses single bytes to represent characters from the ASCII character set, and sequences of two or more bytes for all other characters. (See Section 84.1 [perlunitut NAME], page 1367 for more information about the relationship between UTF-8 and perl's encoding, utf8. The difference isn't important for this discussion.)

No matter how you look at it, Unicode support is going to be a pain in a regex engine. Tricks that might be fine when you have 256 possible characters often won't scale to handle the size of the UTF-8 character set. Things you can take for granted with ASCII may not be true with Unicode. For instance, in ASCII, it is safe to assume that `sizeof(char1) == sizeof(char2)`, but in UTF-8 it isn't. Unicode case folding is vastly more complex than the simple rules of ASCII, and even when not using Unicode but only localised single byte encodings, things can get tricky (for example, **LATIN SMALL LETTER SHARP S** (U+00DF, ß) should match 'SS' in localised case-insensitive matching).

Making things worse is that UTF-8 support was a later addition to the regex engine (as it was to perl) and this necessarily made things a lot more complicated. Obviously it is easier to design a regex engine with Unicode support in mind from the beginning than it is to retrofit it to one that wasn't.

Nearly all regops that involve looking at the input string have two cases, one for UTF-8, and one not. In fact, it's often more complex than that, as the pattern may be UTF-8 as well.

Care must be taken when making changes to make sure that you handle UTF-8 properly, both at compile time and at execution time, including when the string and pattern are mismatched.

64.5.2 Base Structures

The `regexp` structure described in Section 59.1 [perlreapi NAME], page 1032 is common to all regex engines. Two of its fields are intended for the private use of the regex engine that compiled the pattern. These are the `intflags` and pprivate members. The `pprivate` is a void pointer to an arbitrary structure whose use and management is the responsibility of the compiling engine. perl will never modify either of these values. In the case of the stock engine the structure pointed to by `pprivate` is called `regexp_internal`.

Its `pprivate` and `intflags` fields contain data specific to each engine.

There are two structures used to store a compiled regular expression. One, the `regexp` structure described in Section 59.1 [perlreapi NAME], page 1032 is populated by the engine currently being. used and some of its fields read by perl to implement things such as the stringification of `qr//`.

The other structure is pointed to by the `regexp` struct's `pprivate` and is in addition to `intflags` in the same struct considered to be the property of the regex engine which compiled the regular expression;

The regexp structure contains all the data that perl needs to be aware of to properly work with the regular expression. It includes data about optimisations that perl can use to determine if the regex engine should really be used, and various other control info that is needed to properly execute patterns in various contexts such as is the pattern anchored in some way, or what flags were used during the compile, or whether the program contains special constructs that perl needs to be aware of.

In addition it contains two fields that are intended for the private use of the regex engine that compiled the pattern. These are the `intflags` and pprivate members. The `pprivate` is a void pointer to an arbitrary structure whose use and management is the responsibility of the compiling engine. perl will never modify either of these values.

As mentioned earlier, in the case of the default engines, the `pprivate` will be a pointer to a regexp_internal structure which holds the compiled program and any additional data that is private to the regex engine implementation.

64.5.2.1 Perl's `pprivate` structure

The following structure is used as the `pprivate` struct by perl's regex engine. Since it is specific to perl it is only of curiosity value to other engine implementations.

```
 typedef struct regexp_internal {
         U32 *offsets;           /* offset annotations 20001228 MJD
                                  * data about mapping the program to
                                  * the string*/
         regnode *regstclass;    /* Optional startclass as identified or
                                  * constructed by the optimiser */
         struct reg_data *data;  /* Additional miscellaneous data used
                                  * by the program.  Used to make it
                                  * easier to clone and free arbitrary
                                  * data that the regops need. Often the
                                  * ARG field of a regop is an index
                                  * into this structure */
         regnode program[1];     /* Unwarranted chumminess with
                                  * compiler. */
 } regexp_internal;
```

`offsets`

Offsets holds a mapping of offset in the `program` to offset in the `precomp` string. This is only used by ActiveState's visual regex debugger.

`regstclass`

Special regop that is used by `re_intuit_start()` to check if a pattern can match at a certain position. For instance if the regex engine knows that the pattern must start with a 'Z' then it can scan the string until it finds one and then launch the regex engine from there. The routine that handles this is called `find_by_class()`. Sometimes this field points at a regop embedded in the program, and sometimes it points at an independent synthetic regop that has been constructed by the optimiser.

`data`

This field points at a `reg_data` structure, which is defined as follows

```
struct reg_data {
    U32 count;
    U8 *what;
```

```
        void* data[1];
    };
```

This structure is used for handling data structures that the regex engine needs to handle specially during a clone or free operation on the compiled product. Each element in the data array has a corresponding element in the what array. During compilation regops that need special structures stored will add an element to each array using the add_data() routine and then store the index in the regop.

`program`

Compiled program. Inlined into the structure so the entire struct can be treated as a single blob.

64.6 SEE ALSO

Section 59.1 [perlreapi NAME], page 1032

Section 58.1 [perlre NAME], page 989

Section 84.1 [perlunitut NAME], page 1367

64.7 AUTHOR

by Yves Orton, 2006.

With excerpts from Perl, and contributions and suggestions from Ronald J. Kimball, Dave Mitchell, Dominic Dunlop, Mark Jason Dominus, Stephen McCamant, and David Landgren.

64.8 LICENCE

Same terms as Perl.

64.9 REFERENCES

[1] `http://perl.plover.com/Rx/paper/`

[2] `http://www.unicode.org`

65 perlrepository

65.1 NAME

perlrepository - Links to current information on the Perl source repository

65.2 DESCRIPTION

Perl's source code is stored in a Git repository.

See Section 29.1 [perlhack NAME], page 562 for an explanation of Perl development, including the Section 29.3 [Super Quick Patch Guide], page 562 for making and submitting a small patch.

See Section 26.1 [perlgit NAME], page 492 for detailed information about Perl's Git repository.

(The above documents supersede the information that was formerly here in perlrepository.)

66 perlrequick

66.1 NAME

perlrequick - Perl regular expressions quick start

66.2 DESCRIPTION

This page covers the very basics of understanding, creating and using regular expressions ('regexes') in Perl.

66.3 The Guide

66.3.1 Simple word matching

The simplest regex is simply a word, or more generally, a string of characters. A regex consisting of a word matches any string that contains that word:

```
"Hello World" =~ /World/;  # matches
```

In this statement, `World` is a regex and the `//` enclosing `/World/` tells Perl to search a string for a match. The operator `=~` associates the string with the regex match and produces a true value if the regex matched, or false if the regex did not match. In our case, `World` matches the second word in `"Hello World"`, so the expression is true. This idea has several variations.

Expressions like this are useful in conditionals:

```
print "It matches\n" if "Hello World" =~ /World/;
```

The sense of the match can be reversed by using `!~` operator:

```
print "It doesn't match\n" if "Hello World" !~ /World/;
```

The literal string in the regex can be replaced by a variable:

```
$greeting = "World";
print "It matches\n" if "Hello World" =~ /$greeting/;
```

If you're matching against `$_`, the `$_ =~` part can be omitted:

```
$_ = "Hello World";
print "It matches\n" if /World/;
```

Finally, the `//` default delimiters for a match can be changed to arbitrary delimiters by putting an `'m'` out front:

```
"Hello World" =~ m!World!;   # matches, delimited by '!'
"Hello World" =~ m{World};   # matches, note the matching '{}'
"/usr/bin/perl" =~ m"/perl"; # matches after '/usr/bin',
                             # '/' becomes an ordinary char
```

Regexes must match a part of the string *exactly* in order for the statement to be true:

```
"Hello World" =~ /world/;  # doesn't match, case sensitive
"Hello World" =~ /o W/;    # matches, ' ' is an ordinary char
"Hello World" =~ /World /; # doesn't match, no ' ' at end
```

Perl will always match at the earliest possible point in the string:

```
"Hello World" =~ /o/;       # matches 'o' in 'Hello'
"That hat is red" =~ /hat/; # matches 'hat' in 'That'
```

Not all characters can be used 'as is' in a match. Some characters, called **metacharacters**, are reserved for use in regex notation. The metacharacters are

```
{}[]()^$.|*+?\
```

A metacharacter can be matched by putting a backslash before it:

```
"2+2=4" =~ /2+2/;    # doesn't match, + is a metacharacter
"2+2=4" =~ /2\+2/;   # matches, \+ is treated like an ordinary +
'C:\WIN32' =~ /C:\\WIN/;                       # matches
"/usr/bin/perl" =~ /\/usr\/bin\/perl/;  # matches
```

In the last regex, the forward slash `'/'` is also backslashed, because it is used to delimit the regex.

Non-printable ASCII characters are represented by **escape sequences**. Common examples are `\t` for a tab, `\n` for a newline, and `\r` for a carriage return. Arbitrary bytes are represented by octal escape sequences, e.g., `\033`, or hexadecimal escape sequences, e.g., `\x1B`:

```
"1000\t2000" =~ m(0\t2)  # matches
"cat" =~ /\143\x61\x74/  # matches in ASCII, but
                         # a weird way to spell cat
```

Regexes are treated mostly as double-quoted strings, so variable substitution works:

```
$foo = 'house';
'cathouse' =~ /cat$foo/;   # matches
'housecat' =~ /${foo}cat/; # matches
```

With all of the regexes above, if the regex matched anywhere in the string, it was considered a match. To specify *where* it should match, we would use the **anchor** metacharacters `^` and `$`. The anchor `^` means match at the beginning of the string and the anchor `$` means match at the end of the string, or before a newline at the end of the string. Some examples:

```
"housekeeper" =~ /keeper/;         # matches
"housekeeper" =~ /^keeper/;        # doesn't match
"housekeeper" =~ /keeper$/;        # matches
"housekeeper\n" =~ /keeper$/;      # matches
"housekeeper" =~ /^housekeeper$/;  # matches
```

66.3.2 Using character classes

A **character class** allows a set of possible characters, rather than just a single character, to match at a particular point in a regex. Character classes are denoted by brackets `[...]`, with the set of characters to be possibly matched inside. Here are some examples:

```
/cat/;            # matches 'cat'
/[bcr]at/;        # matches 'bat', 'cat', or 'rat'
"abc" =~ /[cab]/; # matches 'a'
```

In the last statement, even though `'c'` is the first character in the class, the earliest point at which the regex can match is `'a'`.

```
/[yY][eE][sS]/; # match 'yes' in a case-insensitive way
                # 'yes', 'Yes', 'YES', etc.
```

```
 /yes/i;         # also match ’yes’ in a case-insensitive way
```

The last example shows a match with an `’i’` **modifier**, which makes the match case-insensitive.

Character classes also have ordinary and special characters, but the sets of ordinary and special characters inside a character class are different than those outside a character class. The special characters for a character class are `-]\^$` and are matched using an escape:

```
/[\]c]def/; # matches ’]def’ or ’cdef’
$x = ’bcr’;
/[$x]at/;   # matches ’bat, ’cat’, or ’rat’
/[\$x]at/;  # matches ’$at’ or ’xat’
/[\\$x]at/; # matches ’\at’, ’bat, ’cat’, or ’rat’
```

The special character `’-’` acts as a range operator within character classes, so that the unwieldy `[0123456789]` and `[abc...xyz]` become the svelte `[0-9]` and `[a-z]`:

```
 /item[0-9]/;  # matches ’item0’ or ... or ’item9’
 /[0-9a-fA-F]/;  # matches a hexadecimal digit
```

If `’-’` is the first or last character in a character class, it is treated as an ordinary character.

The special character `^` in the first position of a character class denotes a **negated character class**, which matches any character but those in the brackets. Both `[...]` and `[^...]` must match a character, or the match fails. Then

```
 /[^a]at/;  # doesn’t match ’aat’ or ’at’, but matches
            # all other ’bat’, ’cat, ’0at’, ’%at’, etc.
 /[^0-9]/;  # matches a non-numeric character
 /[a^]at/;  # matches ’aat’ or ’^at’; here ’^’ is ordinary
```

Perl has several abbreviations for common character classes. (These definitions are those that Perl uses in ASCII-safe mode with the `/a` modifier. Otherwise they could match many more non-ASCII Unicode characters as well. See Section 61.2.2 [perlrecharclass Backslash sequences], page 1059 for details.)

- \d is a digit and represents

  ```
  [0-9]
  ```

- \s is a whitespace character and represents

  ```
  [\ \t\r\n\f]
  ```

- \w is a word character (alphanumeric or _) and represents

  ```
  [0-9a-zA-Z_]
  ```

- \D is a negated \d; it represents any character but a digit

  ```
  [^0-9]
  ```

- \S is a negated \s; it represents any non-whitespace character

  ```
  [^\s]
  ```

- \W is a negated \w; it represents any non-word character

  ```
  [^\w]
  ```

- The period ’.’ matches any character but `"\n"`

The `\d\s\w\D\S\W` abbreviations can be used both inside and outside of character classes. Here are some in use:

```
/\d\d:\d\d:\d\d/; # matches a hh:mm:ss time format
/[\d\s]/;         # matches any digit or whitespace character
/\w\W\w/;         # matches a word char, followed by a
                  # non-word char, followed by a word char
/..rt/;           # matches any two chars, followed by 'rt'
/end\./;          # matches 'end.'
/end[.]/;         # same thing, matches 'end.'
```

The **word anchor** `\b` matches a boundary between a word character and a non-word character `\w\W` or `\W\w`:

```
$x = "Housecat catenates house and cat";
$x =~ /\bcat/;  # matches cat in 'catenates'
$x =~ /cat\b/;  # matches cat in 'housecat'
$x =~ /\bcat\b/;  # matches 'cat' at end of string
```

In the last example, the end of the string is considered a word boundary.

For natural language processing (so that, for example, apostrophes are included in words), use instead `\b{wb}`

```
"don't" =~ / .+? \b{wb} /x;  # matches the whole string
```

66.3.3 Matching this or that

We can match different character strings with the **alternation** metacharacter `'|'`. To match `dog` or `cat`, we form the regex `dog|cat`. As before, Perl will try to match the regex at the earliest possible point in the string. At each character position, Perl will first try to match the first alternative, `dog`. If `dog` doesn't match, Perl will then try the next alternative, `cat`. If `cat` doesn't match either, then the match fails and Perl moves to the next position in the string. Some examples:

```
"cats and dogs" =~ /cat|dog|bird/;  # matches "cat"
"cats and dogs" =~ /dog|cat|bird/;  # matches "cat"
```

Even though `dog` is the first alternative in the second regex, `cat` is able to match earlier in the string.

```
"cats"          =~ /c|ca|cat|cats/; # matches "c"
"cats"          =~ /cats|cat|ca|c/; # matches "cats"
```

At a given character position, the first alternative that allows the regex match to succeed will be the one that matches. Here, all the alternatives match at the first string position, so the first matches.

66.3.4 Grouping things and hierarchical matching

The **grouping** metacharacters `()` allow a part of a regex to be treated as a single unit. Parts of a regex are grouped by enclosing them in parentheses. The regex `house(cat|keeper)` means match `house` followed by either `cat` or `keeper`. Some more examples are

```
/(a|b)b/;    # matches 'ab' or 'bb'
/(^a|b)c/;   # matches 'ac' at start of string or 'bc' anywhere
```

```
/house(cat|)/;  # matches either 'housecat' or 'house'
/house(cat(s|)|)/;  # matches either 'housecats' or 'housecat' or
                    # 'house'.  Note groups can be nested.

"20" =~ /(19|20|)\d\d/;  # matches the null alternative '()\d\d',
                         # because '20\d\d' can't match
```

66.3.5 Extracting matches

The grouping metacharacters `()` also allow the extraction of the parts of a string that matched. For each grouping, the part that matched inside goes into the special variables `$1`, `$2`, etc. They can be used just as ordinary variables:

```
# extract hours, minutes, seconds
$time =~ /(\d\d):(\d\d):(\d\d)/;  # match hh:mm:ss format
$hours = $1;
$minutes = $2;
$seconds = $3;
```

In list context, a match `/regex/` with groupings will return the list of matched values `($1,$2,...)`. So we could rewrite it as

```
($hours, $minutes, $second) = ($time =~ /(\d\d):(\d\d):(\d\d)/);
```

If the groupings in a regex are nested, `$1` gets the group with the leftmost opening parenthesis, `$2` the next opening parenthesis, etc. For example, here is a complex regex and the matching variables indicated below it:

```
/(ab(cd|ef)((gi)|j))/;
 1  2      34
```

Associated with the matching variables `$1`, `$2`, ... are the **backreferences** `\g1`, `\g2`, ... Backreferences are matching variables that can be used *inside* a regex:

```
/(\w\w\w)\s\g1/; # find sequences like 'the the' in string
```

`$1`, `$2`, ... should only be used outside of a regex, and `\g1`, `\g2`, ... only inside a regex.

66.3.6 Matching repetitions

The **quantifier** metacharacters `?`, `*`, `+`, and `{}` allow us to determine the number of repeats of a portion of a regex we consider to be a match. Quantifiers are put immediately after the character, character class, or grouping that we want to specify. They have the following meanings:

- `a?` = match 'a' 1 or 0 times
- `a*` = match 'a' 0 or more times, i.e., any number of times
- `a+` = match 'a' 1 or more times, i.e., at least once
- `a{n,m}` = match at least `n` times, but not more than `m` times.
- `a{n,}` = match at least `n` or more times
- `a{n}` = match exactly `n` times

Here are some examples:

```
/[a-z]+\s+\d*/;  # match a lowercase word, at least some space, and
                 # any number of digits
```

```
/(\w+)\s+\g1/;    # match doubled words of arbitrary length
$year =~ /^\d{2,4}$/;  # make sure year is at least 2 but not more
                       # than 4 digits
$year =~ /^\d{4}$|^\d{2}$/; # better match; throw out 3 digit dates
```

These quantifiers will try to match as much of the string as possible, while still allowing the regex to match. So we have

```
$x = 'the cat in the hat';
$x =~ /^(.*)(at)(.*)$/; # matches,
                        # $1 = 'the cat in the h'
                        # $2 = 'at'
                        # $3 = ''   (0 matches)
```

The first quantifier `.*` grabs as much of the string as possible while still having the regex match. The second quantifier `.*` has no string left to it, so it matches 0 times.

66.3.7 More matching

There are a few more things you might want to know about matching operators. The global modifier `//g` allows the matching operator to match within a string as many times as possible. In scalar context, successive matches against a string will have `//g` jump from match to match, keeping track of position in the string as it goes along. You can get or set the position with the `pos()` function. For example,

```
$x = "cat dog house"; # 3 words
while ($x =~ /(\w+)/g) {
    print "Word is $1, ends at position ", pos $x, "\n";
}
```

prints

```
Word is cat, ends at position 3
Word is dog, ends at position 7
Word is house, ends at position 13
```

A failed match or changing the target string resets the position. If you don't want the position reset after failure to match, add the `//c`, as in `/regex/gc`.

In list context, `//g` returns a list of matched groupings, or if there are no groupings, a list of matches to the whole regex. So

```
@words = ($x =~ /(\w+)/g);  # matches,
                            # $word[0] = 'cat'
                            # $word[1] = 'dog'
                            # $word[2] = 'house'
```

66.3.8 Search and replace

Search and replace is performed using `s/regex/replacement/modifiers`. The `replacement` is a Perl double-quoted string that replaces in the string whatever is matched with the `regex`. The operator `=~` is also used here to associate a string with `s///`. If matching against `$_`, the `$_ =~` can be dropped. If there is a match, `s///` returns the number of substitutions made; otherwise it returns false. Here are a few examples:

```
$x = "Time to feed the cat!";
```

```
$x =~ s/cat/hacker/;   # $x contains "Time to feed the hacker!"
$y = "'quoted words'";
$y =~ s/^'(.*)'$/$1/;  # strip single quotes,
                       # $y contains "quoted words"
```

With the `s///` operator, the matched variables `$1`, `$2`, etc. are immediately available for use in the replacement expression. With the global modifier, `s///g` will search and replace all occurrences of the regex in the string:

```
$x = "I batted 4 for 4";
$x =~ s/4/four/;   # $x contains "I batted four for 4"
$x = "I batted 4 for 4";
$x =~ s/4/four/g;  # $x contains "I batted four for four"
```

The non-destructive modifier `s///r` causes the result of the substitution to be returned instead of modifying `$_` (or whatever variable the substitute was bound to with `=~`):

```
$x = "I like dogs.";
$y = $x =~ s/dogs/cats/r;
print "$x $y\n"; # prints "I like dogs. I like cats."

$x = "Cats are great.";
print $x =~ s/Cats/Dogs/r =~ s/Dogs/Frogs/r =~
    s/Frogs/Hedgehogs/r, "\n";
# prints "Hedgehogs are great."

@foo = map { s/[a-z]/X/r } qw(a b c 1 2 3);
# @foo is now qw(X X X 1 2 3)
```

The evaluation modifier `s///e` wraps an `eval{...}` around the replacement string and the evaluated result is substituted for the matched substring. Some examples:

```
# reverse all the words in a string
$x = "the cat in the hat";
$x =~ s/(\w+)/reverse $1/ge;   # $x contains "eht tac ni eht tah"

# convert percentage to decimal
$x = "A 39% hit rate";
$x =~ s!(\d+)%!$1/100!e;       # $x contains "A 0.39 hit rate"
```

The last example shows that `s///` can use other delimiters, such as `s!!!` and `s{}{}`, and even `s{}//`. If single quotes are used `s'''`, then the regex and replacement are treated as single-quoted strings.

66.3.9 The split operator

`split /regex/, string` splits `string` into a list of substrings and returns that list. The regex determines the character sequence that `string` is split with respect to. For example, to split a string into words, use

```
$x = "Calvin and Hobbes";
@word = split /\s+/, $x;  # $word[0] = 'Calvin'
                          # $word[1] = 'and'
                          # $word[2] = 'Hobbes'
```

To extract a comma-delimited list of numbers, use

```
$x = "1.618,2.718,   3.142";
@const = split /,\s*/, $x;  # $const[0] = '1.618'
                            # $const[1] = '2.718'
                            # $const[2] = '3.142'
```

If the empty regex `//` is used, the string is split into individual characters. If the regex has groupings, then the list produced contains the matched substrings from the groupings as well:

```
$x = "/usr/bin";
@parts = split m!(/)!, $x;  # $parts[0] = ''
                            # $parts[1] = '/'
                            # $parts[2] = 'usr'
                            # $parts[3] = '/'
                            # $parts[4] = 'bin'
```

Since the first character of $x matched the regex, `split` prepended an empty initial element to the list.

66.3.10 `use re 'strict'`

New in v5.22, this applies stricter rules than otherwise when compiling regular expression patterns. It can find things that, while legal, may not be what you intended.

See Section "'strict' mode" in `re`.

66.4 BUGS

None.

66.5 SEE ALSO

This is just a quick start guide. For a more in-depth tutorial on regexes, see Section 68.1 [perlretut NAME], page 1131 and for the reference page, see Section 58.1 [perlre NAME], page 989.

66.6 AUTHOR AND COPYRIGHT

66.6.1 Acknowledgments

The author would like to thank Mark-Jason Dominus, Tom Christiansen, Ilya Zakharevich, Brad Hughes, and Mike Giroux for all their helpful comments.

67 perlreref

67.1 NAME

perlreref - Perl Regular Expressions Reference

67.2 DESCRIPTION

This is a quick reference to Perl's regular expressions. For full information see Section 58.1 [perlre NAME], page 989 and Section 48.1 [perlop NAME], page 798, as well as the Section 67.4 [SEE ALSO], page 1130 section in this document.

67.2.1 OPERATORS

`=~` determines to which variable the regex is applied. In its absence, $_ is used.

```
$var =~ /foo/;
```

`!~` determines to which variable the regex is applied, and negates the result of the match; it returns false if the match succeeds, and true if it fails.

```
$var !~ /foo/;
```

`m/pattern/msixpogcdualn` searches a string for a pattern match, applying the given options.

```
m  Multiline mode - ^ and $ match internal lines
s  match as a Single line - . matches \n
i  case-Insensitive
x  eXtended legibility - free whitespace and comments
p  Preserve a copy of the matched string -
   ${^PREMATCH}, ${^MATCH}, ${^POSTMATCH} will be defined.
o  compile pattern Once
g  Global - all occurrences
c  don't reset pos on failed matches when using /g
a  restrict \d, \s, \w and [:posix:] to match ASCII only
aa (two a's) also /i matches exclude ASCII/non-ASCII
l  match according to current locale
u  match according to Unicode rules
d  match according to native rules unless something indicates
   Unicode
n  Non-capture mode. Don't let () fill in $1, $2, etc...
```

If 'pattern' is an empty string, the last *successfully* matched regex is used. Delimiters other than '/' may be used for both this operator and the following ones. The leading `m` can be omitted if the delimiter is '/'.

`qr/pattern/msixpodualn` lets you store a regex in a variable, or pass one around. Modifiers as for `m//`, and are stored within the regex.

`s/pattern/replacement/msixpogcedual` substitutes matches of 'pattern' with 'replacement'. Modifiers as for `m//`, with two additions:

```
e  Evaluate 'replacement' as an expression
r  Return substitution and leave the original string untouched.
```

'e' may be specified multiple times. 'replacement' is interpreted as a double quoted string unless a single-quote (') is the delimiter.

?pattern? is like m/pattern/ but matches only once. No alternate delimiters can be used. Must be reset with reset().

67.2.2 SYNTAX

```
\       Escapes the character immediately following it
.       Matches any single character except a newline (unless /s is
          used)
^       Matches at the beginning of the string (or line, if /m is used)
$       Matches at the end of the string (or line, if /m is used)
*       Matches the preceding element 0 or more times
+       Matches the preceding element 1 or more times
?       Matches the preceding element 0 or 1 times
{...}   Specifies a range of occurrences for the element preceding it
[...]   Matches any one of the characters contained within the brackets
(...)   Groups subexpressions for capturing to $1, $2...
(?:...) Groups subexpressions without capturing (cluster)
|       Matches either the subexpression preceding or following it
\g1 or \g{1}, \g2 ...    Matches the text from the Nth group
\1, \2, \3 ...           Matches the text from the Nth group
\g-1 or \g{-1}, \g-2 ... Matches the text from the Nth previous group
\g{name}     Named backreference
\k<name>     Named backreference
\k'name'     Named backreference
(?P=name)    Named backreference (python syntax)
```

67.2.3 ESCAPE SEQUENCES

These work as in normal strings.

```
\a       Alarm (beep)
\e       Escape
\f       Formfeed
\n       Newline
\r       Carriage return
\t       Tab
\037     Char whose ordinal is the 3 octal digits, max \777
\o{2307} Char whose ordinal is the octal number, unrestricted
\x7f     Char whose ordinal is the 2 hex digits, max \xFF
\x{263a} Char whose ordinal is the hex number, unrestricted
\cx      Control-x
\N{name} A named Unicode character or character sequence
\N{U+263D} A Unicode character by hex ordinal

\l  Lowercase next character
\u  Titlecase next character
\L  Lowercase until \E
```

```
\U  Uppercase until \E
\F  Foldcase until \E
\Q  Disable pattern metacharacters until \E
\E  End modification
```

For Titlecase, see Section 67.2.10.1 [Titlecase], page 1130.

This one works differently from normal strings:

```
\b  An assertion, not backspace, except in a character class
```

67.2.4 CHARACTER CLASSES

```
[amy]    Match 'a', 'm' or 'y'
[f-j]    Dash specifies "range"
[f-j-]   Dash escaped or at start or end means 'dash'
[^f-j]   Caret indicates "match any character _except_ these"
```

The following sequences (except `\N`) work within or without a character class. The first six are locale aware, all are Unicode aware. See Section 38.1 [perllocale NAME], page 701 and Section 81.1 [perlunicode NAME], page 1317 for details.

```
\d      A digit
\D      A nondigit
\w      A word character
\W      A non-word character
\s      A whitespace character
\S      A non-whitespace character
\h      An horizontal whitespace
\H      A non horizontal whitespace
\N      A non newline (when not followed by '{NAME}';;
        not valid in a character class; equivalent to [^\n]; it's
        like '.' without /s modifier)
\v      A vertical whitespace
\V      A non vertical whitespace
\R      A generic newline           (?>\v|\x0D\x0A)

\C      Match a byte (with Unicode, '.' matches a character)
        (Deprecated.)
\pP     Match P-named (Unicode) property
\p{...} Match Unicode property with name longer than 1 character
\PP     Match non-P
\P{...} Match lack of Unicode property with name longer than 1 char
\X      Match Unicode extended grapheme cluster
```

POSIX character classes and their Unicode and Perl equivalents:

```
           ASCII-         Full-
  POSIX    range          range    backslash
 [[:...:]] \p{...}        \p{...}   sequence    Description

 -----------------------------------------------------------------------
 alnum   PosixAlnum       XPosixAlnum            Alpha plus Digit
```

```
alpha   PosixAlpha       XPosixAlpha              Alphabetic characters
ascii   ASCII                                     Any ASCII character
blank   PosixBlank       XPosixBlank   \h         Horizontal whitespace;
                                                    full-range also
                                                    written as
                                                    \p{HorizSpace} (GNU
                                                    extension)
cntrl   PosixCntrl       XPosixCntrl              Control characters
digit   PosixDigit       XPosixDigit   \d         Decimal digits
graph   PosixGraph       XPosixGraph              Alnum plus Punct
lower   PosixLower       XPosixLower              Lowercase characters
print   PosixPrint       XPosixPrint              Graph plus Print, but
                                                    not any Cntrls
punct   PosixPunct       XPosixPunct              Punctuation and Symbols
                                                    in ASCII-range; just
                                                    punct outside it
space   PosixSpace       XPosixSpace              [\s\cK]
        PerlSpace        XPerlSpace    \s         Perl's whitespace def'n
upper   PosixUpper       XPosixUpper              Uppercase characters
word    PosixWord        XPosixWord    \w         Alnum + Unicode marks +
                                                    connectors, like '_'
                                                    (Perl extension)
xdigit  ASCII_Hex_Digit  XPosixDigit              Hexadecimal digit,
                                                     ASCII-range is
                                                     [0-9A-Fa-f]
```

Also, various synonyms like `\p{Alpha}` for `\p{XPosixAlpha}`; all listed in Section "Properties accessible through \p{} and \P{}" in `perluniprops`

Within a character class:

```
  POSIX      traditional   Unicode
[:digit:]       \d        \p{Digit}
[:^digit:]      \D        \P{Digit}
```

67.2.5 ANCHORS

All are zero-width assertions.

```
^  Match string start (or line, if /m is used)
$  Match string end (or line, if /m is used) or before newline
\b{} Match boundary of type specified within the braces
\B{} Match wherever \b{} doesn't match
\b Match word boundary (between \w and \W)
\B Match except at word boundary (between \w and \w or \W and \W)
\A Match string start (regardless of /m)
\Z Match string end (before optional newline)
\z Match absolute string end
\G Match where previous m//g left off
\K Keep the stuff left of the \K, don't include it in $&
```

67.2.6 QUANTIFIERS

Quantifiers are greedy by default and match the **longest** leftmost.

```
Maximal Minimal Possessive Allowed range
------- ------- ---------- -------------
{n,m}   {n,m}?  {n,m}+     Must occur at least n times
                           but no more than m times
{n,}    {n,}?   {n,}+      Must occur at least n times
{n}     {n}?    {n}+       Must occur exactly n times
*       *?      *+         0 or more times (same as {0,})
+       +?      ++         1 or more times (same as {1,})
?       ??      ?+         0 or 1 time (same as {0,1})
```

The possessive forms (new in Perl 5.10) prevent backtracking: what gets matched by a pattern with a possessive quantifier will not be backtracked into, even if that causes the whole match to fail.

There is no quantifier `{,n}`. That's interpreted as a literal string.

67.2.7 EXTENDED CONSTRUCTS

```
(?#text)          A comment
(?:...)           Groups subexpressions without capturing (cluster)
(?pimsx-imsx:...) Enable/disable option (as per m// modifiers)
(?=...)           Zero-width positive lookahead assertion
(?!...)           Zero-width negative lookahead assertion
(?<=...)          Zero-width positive lookbehind assertion
(?<!...)          Zero-width negative lookbehind assertion
(?>...)           Grab what we can, prohibit backtracking
(?|...)           Branch reset
(?<name>...)      Named capture
(?'name'...)      Named capture
(?P<name>...)     Named capture (python syntax)
(?[...])          Extended bracketed character class
(?{ code })       Embedded code, return value becomes $^R
(??{ code })      Dynamic regex, return value used as regex
(?N)              Recurse into subpattern number N
(?-N), (?+N)      Recurse into Nth previous/next subpattern
(?R), (?0)        Recurse at the beginning of the whole pattern
(?&name)          Recurse into a named subpattern
(?P>name)         Recurse into a named subpattern (python syntax)
(?(cond)yes|no)
(?(cond)yes)      Conditional expression, where "cond" can be:
                  (?=pat)   look-ahead
                  (?!pat)   negative look-ahead
                  (?<=pat)  look-behind
                  (?<!pat)  negative look-behind
                  (N)       subpattern N has matched something
                  (<name>)  named subpattern has matched something
                  ('name')  named subpattern has matched something
```

```
(?{code}) code condition
(R)       true if recursing
(RN)      true if recursing into Nth subpattern
(R&name)  true if recursing into named subpattern
(DEFINE)  always false, no no-pattern allowed
```

67.2.8 VARIABLES

```
$_    Default variable for operators to use

$`    Everything prior to matched string
$&    Entire matched string
$'    Everything after to matched string

${^PREMATCH}   Everything prior to matched string
${^MATCH}      Entire matched string
${^POSTMATCH}  Everything after to matched string
```

Note to those still using Perl 5.18 or earlier: The use of `` $` ``, `$&` or `$'` will slow down **all** regex use within your program. Consult Section 86.1 [perlvar NAME], page 1375 for `@-` to see equivalent expressions that won't cause slow down. See also `Devel-SawAmpersand`. Starting with Perl 5.10, you can also use the equivalent variables `${^PREMATCH}`, `${^MATCH}` and `${^POSTMATCH}`, but for them to be defined, you have to specify the `/p` (preserve) modifier on your regular expression. In Perl 5.20, the use of `` $` ``, `$&` and `$'` makes no speed difference.

```
$1, $2 ...  hold the Xth captured expr
$+    Last parenthesized pattern match
$^N   Holds the most recently closed capture
$^R   Holds the result of the last (?{...}) expr
@-    Offsets of starts of groups. $-[0] holds start of whole match
@+    Offsets of ends of groups. $+[0] holds end of whole match
%+    Named capture groups
%-    Named capture groups, as array refs
```

Captured groups are numbered according to their *opening* paren.

67.2.9 FUNCTIONS

```
lc          Lowercase a string
lcfirst     Lowercase first char of a string
uc          Uppercase a string
ucfirst     Titlecase first char of a string
fc          Foldcase a string

pos         Return or set current match position
quotemeta   Quote metacharacters
reset       Reset ?pattern? status
study       Analyze string for optimizing matching

split       Use a regex to split a string into parts
```

The first five of these are like the escape sequences `\L`, `\l`, `\U`, `\u`, and `\F`. For Titlecase, see Section 67.2.10.1 [Titlecase], page 1130; For Foldcase, see Section 67.2.10.2 [Foldcase], page 1130.

67.2.10 TERMINOLOGY

67.2.10.1 Titlecase

Unicode concept which most often is equal to uppercase, but for certain characters like the German "sharp s" there is a difference.

67.2.10.2 Foldcase

Unicode form that is useful when comparing strings regardless of case, as certain characters have complex one-to-many case mappings. Primarily a variant of lowercase.

67.3 AUTHOR

Iain Truskett. Updated by the Perl 5 Porters.

This document may be distributed under the same terms as Perl itself.

67.4 SEE ALSO

- Section 68.1 [perlretut NAME], page 1131 for a tutorial on regular expressions.
- Section 66.1 [perlrequick NAME], page 1116 for a rapid tutorial.
- Section 58.1 [perlre NAME], page 989 for more details.
- Section 86.1 [perlvar NAME], page 1375 for details on the variables.
- Section 48.1 [perlop NAME], page 798 for details on the operators.
- Section 25.1 [perlfunc NAME], page 351 for details on the functions.
- `perlfaq6` for FAQs on regular expressions.
- Section 60.1 [perlrebackslash NAME], page 1046 for a reference on backslash sequences.
- Section 61.1 [perlrecharclass NAME], page 1059 for a reference on character classes.
- The `re` module to alter behaviour and aid debugging.
- Section 15.4 [perldebug Debugging Regular Expressions], page 135
- Section 83.1 [perluniintro NAME], page 1352, Section 81.1 [perlunicode NAME], page 1317, `charnames` and Section 38.1 [perllocale NAME], page 701 for details on regexes and internationalisation.
- *Mastering Regular Expressions* by Jeffrey Friedl (`http://oreilly.com/catalog/9780596528126/`) for a thorough grounding and reference on the topic.

67.5 THANKS

David P.C. Wollmann, Richard Soderberg, Sean M. Burke, Tom Christiansen, Jim Cromie, and Jeffrey Goff for useful advice.

68 perlretut

68.1 NAME

perlretut - Perl regular expressions tutorial

68.2 DESCRIPTION

This page provides a basic tutorial on understanding, creating and using regular expressions in Perl. It serves as a complement to the reference page on regular expressions Section 58.1 [perlre NAME], page 989. Regular expressions are an integral part of the `m//`, `s///`, `qr//` and `split` operators and so this tutorial also overlaps with Section 48.2.30 [perlop Regexp Quote-Like Operators], page 823 and [perlfunc split], page 453.

Perl is widely renowned for excellence in text processing, and regular expressions are one of the big factors behind this fame. Perl regular expressions display an efficiency and flexibility unknown in most other computer languages. Mastering even the basics of regular expressions will allow you to manipulate text with surprising ease.

What is a regular expression? A regular expression is simply a string that describes a pattern. Patterns are in common use these days; examples are the patterns typed into a search engine to find web pages and the patterns used to list files in a directory, e.g., `ls *.txt` or `dir *.*`. In Perl, the patterns described by regular expressions are used to search strings, extract desired parts of strings, and to do search and replace operations.

Regular expressions have the undeserved reputation of being abstract and difficult to understand. Regular expressions are constructed using simple concepts like conditionals and loops and are no more difficult to understand than the corresponding `if` conditionals and `while` loops in the Perl language itself. In fact, the main challenge in learning regular expressions is just getting used to the terse notation used to express these concepts.

This tutorial flattens the learning curve by discussing regular expression concepts, along with their notation, one at a time and with many examples. The first part of the tutorial will progress from the simplest word searches to the basic regular expression concepts. If you master the first part, you will have all the tools needed to solve about 98% of your needs. The second part of the tutorial is for those comfortable with the basics and hungry for more power tools. It discusses the more advanced regular expression operators and introduces the latest cutting-edge innovations.

A note: to save time, 'regular expression' is often abbreviated as regexp or regex. Regexp is a more natural abbreviation than regex, but is harder to pronounce. The Perl pod documentation is evenly split on regexp vs regex; in Perl, there is more than one way to abbreviate it. We'll use regexp in this tutorial.

New in v5.22, Section “'strict' mode” in `re` applies stricter rules than otherwise when compiling regular expression patterns. It can find things that, while legal, may not be what you intended.

68.3 Part 1: The basics

68.3.1 Simple word matching

The simplest regexp is simply a word, or more generally, a string of characters. A regexp consisting of a word matches any string that contains that word:

```
"Hello World" =~ /World/;  # matches
```

What is this Perl statement all about? "Hello World" is a simple double-quoted string. World is the regular expression and the // enclosing /World/ tells Perl to search a string for a match. The operator =~ associates the string with the regexp match and produces a true value if the regexp matched, or false if the regexp did not match. In our case, World matches the second word in "Hello World", so the expression is true. Expressions like this are useful in conditionals:

```
if ("Hello World" =~ /World/) {
    print "It matches\n";
}
else {
    print "It doesn't match\n";
}
```

There are useful variations on this theme. The sense of the match can be reversed by using the !~ operator:

```
if ("Hello World" !~ /World/) {
    print "It doesn't match\n";
}
else {
    print "It matches\n";
}
```

The literal string in the regexp can be replaced by a variable:

```
$greeting = "World";
if ("Hello World" =~ /$greeting/) {
    print "It matches\n";
}
else {
    print "It doesn't match\n";
}
```

If you're matching against the special default variable $_, the $_ =~ part can be omitted:

```
$_ = "Hello World";
if (/World/) {
    print "It matches\n";
}
else {
    print "It doesn't match\n";
}
```

And finally, the // default delimiters for a match can be changed to arbitrary delimiters by putting an 'm' out front:

```
"Hello World" =~ m!World!;   # matches, delimited by '!'
"Hello World" =~ m{World};   # matches, note the matching '{}'
```

```
"/usr/bin/perl" =~ m"/perl"; # matches after ’/usr/bin’,
                             # ’/’ becomes an ordinary char
```

`/World/`, `m!World!`, and `m{World}` all represent the same thing. When, e.g., the quote (`"`) is used as a delimiter, the forward slash `’/’` becomes an ordinary character and can be used in this regexp without trouble.

Let’s consider how different regexps would match `"Hello World"`:

```
"Hello World" =~ /world/;  # doesn’t match
"Hello World" =~ /o W/;    # matches
"Hello World" =~ /oW/;     # doesn’t match
"Hello World" =~ /World /; # doesn’t match
```

The first regexp `world` doesn’t match because regexps are case-sensitive. The second regexp matches because the substring `’o W’` occurs in the string `"Hello World"`. The space character ’ ’ is treated like any other character in a regexp and is needed to match in this case. The lack of a space character is the reason the third regexp `’oW’` doesn’t match. The fourth regexp `’World ’` doesn’t match because there is a space at the end of the regexp, but not at the end of the string. The lesson here is that regexps must match a part of the string *exactly* in order for the statement to be true.

If a regexp matches in more than one place in the string, Perl will always match at the earliest possible point in the string:

```
"Hello World" =~ /o/;       # matches ’o’ in ’Hello’
"That hat is red" =~ /hat/; # matches ’hat’ in ’That’
```

With respect to character matching, there are a few more points you need to know about. First of all, not all characters can be used ’as is’ in a match. Some characters, called *metacharacters*, are reserved for use in regexp notation. The metacharacters are

```
{}[]()^$.|*+?\
```

The significance of each of these will be explained in the rest of the tutorial, but for now, it is important only to know that a metacharacter can be matched by putting a backslash before it:

```
"2+2=4" =~ /2+2/;    # doesn’t match, + is a metacharacter
"2+2=4" =~ /2\+2/;   # matches, \+ is treated like an ordinary +
"The interval is [0,1)." =~ /[0,1)./     # is a syntax error!
"The interval is [0,1)." =~ /\[0,1\)\./  # matches
"#!/usr/bin/perl" =~ /#!\/usr\/bin\/perl/;  # matches
```

In the last regexp, the forward slash `’/’` is also backslashed, because it is used to delimit the regexp. This can lead to LTS (leaning toothpick syndrome), however, and it is often more readable to change delimiters.

```
"#!/usr/bin/perl" =~ m!#\!/usr/bin/perl!;  # easier to read
```

The backslash character `’\’` is a metacharacter itself and needs to be backslashed:

```
’C:\WIN32’ =~ /C:\\WIN/;   # matches
```

In addition to the metacharacters, there are some ASCII characters which don’t have printable character equivalents and are instead represented by *escape sequences*. Common examples are `\t` for a tab, `\n` for a newline, `\r` for a carriage return and `\a` for a bell (or alert). If your string is better thought of as a sequence of arbitrary bytes, the octal escape sequence, e.g., \033, or hexadecimal escape sequence, e.g., \x1B may be a more natural representation for your bytes. Here are some examples of escapes:

```
"1000\t2000" =~ m(0\t2)   # matches
"1000\n2000" =~ /0\n20/   # matches
"1000\t2000" =~ /\000\t2/ # doesn't match, "0" ne "\000"
"cat"   =~ /\o{143}\x61\x74/ # matches in ASCII, but a weird way
                             # to spell cat
```

If you've been around Perl a while, all this talk of escape sequences may seem familiar. Similar escape sequences are used in double-quoted strings and in fact the regexps in Perl are mostly treated as double-quoted strings. This means that variables can be used in regexps as well. Just like double-quoted strings, the values of the variables in the regexp will be substituted in before the regexp is evaluated for matching purposes. So we have:

```
$foo = 'house';
'housecat' =~ /$foo/;      # matches
'cathouse' =~ /cat$foo/;   # matches
'housecat' =~ /${foo}cat/; # matches
```

So far, so good. With the knowledge above you can already perform searches with just about any literal string regexp you can dream up. Here is a *very simple* emulation of the Unix grep program:

```
% cat > simple_grep
#!/usr/bin/perl
$regexp = shift;
while (<>) {
    print if /$regexp/;
}
^D

% chmod +x simple_grep

% simple_grep abba /usr/dict/words
Babbage
cabbage
cabbages
sabbath
Sabbathize
Sabbathizes
sabbatical
scabbard
scabbards
```

This program is easy to understand. #!/usr/bin/perl is the standard way to invoke a perl program from the shell. $regexp = shift; saves the first command line argument as the regexp to be used, leaving the rest of the command line arguments to be treated as files. while (<>) loops over all the lines in all the files. For each line, print if /$regexp/; prints the line if the regexp matches the line. In this line, both print and /$regexp/ use the default variable $_ implicitly.

With all of the regexps above, if the regexp matched anywhere in the string, it was considered a match. Sometimes, however, we'd like to specify *where* in the string the regexp should try to match. To do this, we would use the *anchor* metacharacters ^ and $. The anchor ^ means match at the beginning of the string and the anchor $ means match at the end of the string, or before a newline at the end of the string. Here is how they are used:

```
"housekeeper" =~ /keeper/;    # matches
"housekeeper" =~ /^keeper/;   # doesn't match
"housekeeper" =~ /keeper$/;   # matches
"housekeeper\n" =~ /keeper$/; # matches
```

The second regexp doesn't match because ^ constrains `keeper` to match only at the beginning of the string, but `"housekeeper"` has keeper starting in the middle. The third regexp does match, since the $ constrains `keeper` to match only at the end of the string.

When both ^ and $ are used at the same time, the regexp has to match both the beginning and the end of the string, i.e., the regexp matches the whole string. Consider

```
"keeper" =~ /^keep$/;      # doesn't match
"keeper" =~ /^keeper$/;    # matches
""       =~ /^$/;          # ^$ matches an empty string
```

The first regexp doesn't match because the string has more to it than `keep`. Since the second regexp is exactly the string, it matches. Using both ^ and $ in a regexp forces the complete string to match, so it gives you complete control over which strings match and which don't. Suppose you are looking for a fellow named bert, off in a string by himself:

```
"dogbert" =~ /bert/;   # matches, but not what you want

"dilbert" =~ /^bert/;  # doesn't match, but ..
"bertram" =~ /^bert/;  # matches, so still not good enough

"bertram" =~ /^bert$/; # doesn't match, good
"dilbert" =~ /^bert$/; # doesn't match, good
"bert"    =~ /^bert$/; # matches, perfect
```

Of course, in the case of a literal string, one could just as easily use the string comparison `$string eq 'bert'` and it would be more efficient. The `^...$` regexp really becomes useful when we add in the more powerful regexp tools below.

68.3.2 Using character classes

Although one can already do quite a lot with the literal string regexps above, we've only scratched the surface of regular expression technology. In this and subsequent sections we will introduce regexp concepts (and associated metacharacter notations) that will allow a regexp to represent not just a single character sequence, but a *whole class* of them.

One such concept is that of a *character class*. A character class allows a set of possible characters, rather than just a single character, to match at a particular point in a regexp. You can define your own custom character classes. These are denoted by brackets `[...]`, with the set of characters to be possibly matched inside. Here are some examples:

```
/cat/;       # matches 'cat'
/[bcr]at/;   # matches 'bat, 'cat', or 'rat'
```

```
/item[0123456789]/;  # matches 'item0' or ... or 'item9'
"abc" =~ /[cab]/;    # matches 'a'
```

In the last statement, even though 'c' is the first character in the class, 'a' matches because the first character position in the string is the earliest point at which the regexp can match.

```
/[yY][eE][sS]/;      # match 'yes' in a case-insensitive way
                     # 'yes', 'Yes', 'YES', etc.
```

This regexp displays a common task: perform a case-insensitive match. Perl provides a way of avoiding all those brackets by simply appending an 'i' to the end of the match. Then /[yY][eE][sS]/; can be rewritten as /yes/i;. The 'i' stands for case-insensitive and is an example of a *modifier* of the matching operation. We will meet other modifiers later in the tutorial.

We saw in the section above that there were ordinary characters, which represented themselves, and special characters, which needed a backslash \ to represent themselves. The same is true in a character class, but the sets of ordinary and special characters inside a character class are different than those outside a character class. The special characters for a character class are -]\^$ (and the pattern delimiter, whatever it is).] is special because it denotes the end of a character class. $ is special because it denotes a scalar variable. \ is special because it is used in escape sequences, just like above. Here is how the special characters]$\ are handled:

```
/[\]c]def/; # matches ']def' or 'cdef'
$x = 'bcr';
/[$x]at/;   # matches 'bat', 'cat', or 'rat'
/[\$x]at/;  # matches '$at' or 'xat'
/[\\$x]at/; # matches '\at', 'bat, 'cat', or 'rat'
```

The last two are a little tricky. In [\$x], the backslash protects the dollar sign, so the character class has two members $ and x. In [\\$x], the backslash is protected, so $x is treated as a variable and substituted in double quote fashion.

The special character '-' acts as a range operator within character classes, so that a contiguous set of characters can be written as a range. With ranges, the unwieldy [0123456789] and [abc...xyz] become the svelte [0-9] and [a-z]. Some examples are

```
/item[0-9]/;  # matches 'item0' or ... or 'item9'
/[0-9bx-z]aa/;  # matches '0aa', ..., '9aa',
                # 'baa', 'xaa', 'yaa', or 'zaa'
/[0-9a-fA-F]/;  # matches a hexadecimal digit
/[0-9a-zA-Z_]/; # matches a "word" character,
                # like those in a Perl variable name
```

If '-' is the first or last character in a character class, it is treated as an ordinary character; [-ab], [ab-] and [a\-b] are all equivalent.

The special character ^ in the first position of a character class denotes a *negated character class*, which matches any character but those in the brackets. Both [...] and [^...] must match a character, or the match fails. Then

```
/[^a]at/;  # doesn't match 'aat' or 'at', but matches
```

```
                # all other 'bat', 'cat, '0at', '%at', etc.
    /[^0-9]/;   # matches a non-numeric character
    /[a^]at/;   # matches 'aat' or '^at'; here '^' is ordinary
```

Now, even [0-9] can be a bother to write multiple times, so in the interest of saving keystrokes and making regexps more readable, Perl has several abbreviations for common character classes, as shown below. Since the introduction of Unicode, unless the //a modifier is in effect, these character classes match more than just a few characters in the ASCII range.

- \d matches a digit, not just [0-9] but also digits from non-roman scripts
- \s matches a whitespace character, the set [\ \t\r\n\f] and others
- \w matches a word character (alphanumeric or _), not just [0-9a-zA-Z_] but also digits and characters from non-roman scripts
- \D is a negated \d; it represents any other character than a digit, or [^\d]
- \S is a negated \s; it represents any non-whitespace character [^\s]
- \W is a negated \w; it represents any non-word character [^\w]
- The period '.' matches any character but "\n" (unless the modifier //s is in effect, as explained below).
- \N, like the period, matches any character but "\n", but it does so regardless of whether the modifier //s is in effect.

The //a modifier, available starting in Perl 5.14, is used to restrict the matches of \d, \s, and \w to just those in the ASCII range. It is useful to keep your program from being needlessly exposed to full Unicode (and its accompanying security considerations) when all you want is to process English-like text. (The "a" may be doubled, //aa, to provide even more restrictions, preventing case-insensitive matching of ASCII with non-ASCII characters; otherwise a Unicode "Kelvin Sign" would caselessly match a "k" or "K".)

The \d\s\w\D\S\W abbreviations can be used both inside and outside of bracketed character classes. Here are some in use:

```
    /\d\d:\d\d:\d\d/; # matches a hh:mm:ss time format
    /[\d\s]/;         # matches any digit or whitespace character
    /\w\W\w/;         # matches a word char, followed by a
                      # non-word char, followed by a word char
    /..rt/;           # matches any two chars, followed by 'rt'
    /end\./;          # matches 'end.'
    /end[.]/;         # same thing, matches 'end.'
```

Because a period is a metacharacter, it needs to be escaped to match as an ordinary period. Because, for example, \d and \w are sets of characters, it is incorrect to think of [^\d\w] as [\D\W]; in fact [^\d\w] is the same as [^\w], which is the same as [\W]. Think DeMorgan's laws.

In actuality, the period and \d\s\w\D\S\W abbreviations are themselves types of character classes, so the ones surrounded by brackets are just one type of character class. When we need to make a distinction, we refer to them as "bracketed character classes."

An anchor useful in basic regexps is the *word anchor* \b. This matches a boundary between a word character and a non-word character \w\W or \W\w:

```
$x = "Housecat catenates house and cat";
$x =~ /cat/;    # matches cat in 'housecat'
$x =~ /\bcat/;  # matches cat in 'catenates'
$x =~ /cat\b/;  # matches cat in 'housecat'
$x =~ /\bcat\b/;  # matches 'cat' at end of string
```

Note in the last example, the end of the string is considered a word boundary.

For natural language processing (so that, for example, apostrophes are included in words), use instead `\b{wb}`

```
"don't" =~ / .+? \b{wb} /x;  # matches the whole string
```

You might wonder why `'.'` matches everything but `"\n"` - why not every character? The reason is that often one is matching against lines and would like to ignore the newline characters. For instance, while the string `"\n"` represents one line, we would like to think of it as empty. Then

```
""   =~ /^$/;    # matches
"\n" =~ /^$/;    # matches, $ anchors before "\n"

""   =~ /./;      # doesn't match; it needs a char
""   =~ /^.$/;    # doesn't match; it needs a char
"\n" =~ /^.$/;    # doesn't match; it needs a char other than "\n"
"a"  =~ /^.$/;    # matches
"a\n"  =~ /^.$/;  # matches, $ anchors before "\n"
```

This behavior is convenient, because we usually want to ignore newlines when we count and match characters in a line. Sometimes, however, we want to keep track of newlines. We might even want `^` and `$` to anchor at the beginning and end of lines within the string, rather than just the beginning and end of the string. Perl allows us to choose between ignoring and paying attention to newlines by using the `//s` and `//m` modifiers. `//s` and `//m` stand for single line and multi-line and they determine whether a string is to be treated as one continuous string, or as a set of lines. The two modifiers affect two aspects of how the regexp is interpreted: 1) how the `'.'` character class is defined, and 2) where the anchors `^` and `$` are able to match. Here are the four possible combinations:

- no modifiers (//): Default behavior. `'.'` matches any character except `"\n"`. `^` matches only at the beginning of the string and `$` matches only at the end or before a newline at the end.
- s modifier (//s): Treat string as a single long line. `'.'` matches any character, even `"\n"`. `^` matches only at the beginning of the string and `$` matches only at the end or before a newline at the end.
- m modifier (//m): Treat string as a set of multiple lines. `'.'` matches any character except `"\n"`. `^` and `$` are able to match at the start or end of *any* line within the string.
- both s and m modifiers (//sm): Treat string as a single long line, but detect multiple lines. `'.'` matches any character, even `"\n"`. `^` and `$`, however, are able to match at the start or end of *any* line within the string.

Here are examples of `//s` and `//m` in action:

```
$x = "There once was a girl\nWho programmed in Perl\n";
```

```
$x =~ /^Who/;   # doesn't match, "Who" not at start of string
$x =~ /^Who/s;  # doesn't match, "Who" not at start of string
$x =~ /^Who/m;  # matches, "Who" at start of second line
$x =~ /^Who/sm; # matches, "Who" at start of second line

$x =~ /girl.Who/;   # doesn't match, "." doesn't match "\n"
$x =~ /girl.Who/s;  # matches, "." matches "\n"
$x =~ /girl.Who/m;  # doesn't match, "." doesn't match "\n"
$x =~ /girl.Who/sm; # matches, "." matches "\n"
```

Most of the time, the default behavior is what is wanted, but `//s` and `//m` are occasionally very useful. If `//m` is being used, the start of the string can still be matched with `\A` and the end of the string can still be matched with the anchors `\Z` (matches both the end and the newline before, like `$`), and `\z` (matches only the end):

```
$x =~ /^Who/m;   # matches, "Who" at start of second line
$x =~ /\AWho/m;  # doesn't match, "Who" is not at start of string

$x =~ /girl$/m;  # matches, "girl" at end of first line
$x =~ /girl\Z/m; # doesn't match, "girl" is not at end of string

$x =~ /Perl\Z/m; # matches, "Perl" is at newline before end
$x =~ /Perl\z/m; # doesn't match, "Perl" is not at end of string
```

We now know how to create choices among classes of characters in a regexp. What about choices among words or character strings? Such choices are described in the next section.

68.3.3 Matching this or that

Sometimes we would like our regexp to be able to match different possible words or character strings. This is accomplished by using the *alternation* metacharacter `|`. To match `dog` or `cat`, we form the regexp `dog|cat`. As before, Perl will try to match the regexp at the earliest possible point in the string. At each character position, Perl will first try to match the first alternative, `dog`. If `dog` doesn't match, Perl will then try the next alternative, `cat`. If `cat` doesn't match either, then the match fails and Perl moves to the next position in the string. Some examples:

```
"cats and dogs" =~ /cat|dog|bird/;  # matches "cat"
"cats and dogs" =~ /dog|cat|bird/;  # matches "cat"
```

Even though `dog` is the first alternative in the second regexp, `cat` is able to match earlier in the string.

```
"cats"          =~ /c|ca|cat|cats/; # matches "c"
"cats"          =~ /cats|cat|ca|c/; # matches "cats"
```

Here, all the alternatives match at the first string position, so the first alternative is the one that matches. If some of the alternatives are truncations of the others, put the longest ones first to give them a chance to match.

```
"cab" =~ /a|b|c/ # matches "c"
                 # /a|b|c/ == /[abc]/
```

The last example points out that character classes are like alternations of characters. At a given character position, the first alternative that allows the regexp match to succeed will be the one that matches.

68.3.4 Grouping things and hierarchical matching

Alternation allows a regexp to choose among alternatives, but by itself it is unsatisfying. The reason is that each alternative is a whole regexp, but sometime we want alternatives for just part of a regexp. For instance, suppose we want to search for housecats or housekeepers. The regexp `housecat|housekeeper` fits the bill, but is inefficient because we had to type `house` twice. It would be nice to have parts of the regexp be constant, like `house`, and some parts have alternatives, like `cat|keeper`.

The *grouping* metacharacters `()` solve this problem. Grouping allows parts of a regexp to be treated as a single unit. Parts of a regexp are grouped by enclosing them in parentheses. Thus we could solve the `housecat|housekeeper` by forming the regexp as `house(cat|keeper)`. The regexp `house(cat|keeper)` means match `house` followed by either `cat` or `keeper`. Some more examples are

```
/(a|b)b/;    # matches 'ab' or 'bb'
/(ac|b)b/;   # matches 'acb' or 'bb'
/(^a|b)c/;   # matches 'ac' at start of string or 'bc' anywhere
/(a|[bc])d/; # matches 'ad', 'bd', or 'cd'

/house(cat|)/;  # matches either 'housecat' or 'house'
/house(cat(s|)|)/;  # matches either 'housecats' or 'housecat' or
                    # 'house'.  Note groups can be nested.

/(19|20|)\d\d/;  # match years 19xx, 20xx, or the Y2K problem, xx
"20" =~ /(19|20|)\d\d/;  # matches the null alternative '()\d\d',
                         # because '20\d\d' can't match
```

Alternations behave the same way in groups as out of them: at a given string position, the leftmost alternative that allows the regexp to match is taken. So in the last example at the first string position, `"20"` matches the second alternative, but there is nothing left over to match the next two digits `\d\d`. So Perl moves on to the next alternative, which is the null alternative and that works, since `"20"` is two digits.

The process of trying one alternative, seeing if it matches, and moving on to the next alternative, while going back in the string from where the previous alternative was tried, if it doesn't, is called *backtracking*. The term 'backtracking' comes from the idea that matching a regexp is like a walk in the woods. Successfully matching a regexp is like arriving at a destination. There are many possible trailheads, one for each string position, and each one is tried in order, left to right. From each trailhead there may be many paths, some of which get you there, and some which are dead ends. When you walk along a trail and hit a dead end, you have to backtrack along the trail to an earlier point to try another trail. If you hit your destination, you stop immediately and forget about trying all the other trails. You are persistent, and only if you have tried all the trails from all the trailheads and not arrived at your destination, do you declare failure. To be concrete, here is a step-by-step analysis of what Perl does when it tries to match the regexp

```
"abcde" =~ /(abd|abc)(df|d|de)/;
```

0

Start with the first letter in the string 'a'.

1

Try the first alternative in the first group 'abd'.

2

Match 'a' followed by 'b'. So far so good.

3

'd' in the regexp doesn't match 'c' in the string - a dead end. So backtrack two characters and pick the second alternative in the first group 'abc'.

4

Match 'a' followed by 'b' followed by 'c'. We are on a roll and have satisfied the first group. Set $1 to 'abc'.

5

Move on to the second group and pick the first alternative 'df'.

6

Match the 'd'.

7

'f' in the regexp doesn't match 'e' in the string, so a dead end. Backtrack one character and pick the second alternative in the second group 'd'.

8

'd' matches. The second grouping is satisfied, so set $2 to 'd'.

9

We are at the end of the regexp, so we are done! We have matched 'abcd' out of the string "abcde".

There are a couple of things to note about this analysis. First, the third alternative in the second group 'de' also allows a match, but we stopped before we got to it - at a given character position, leftmost wins. Second, we were able to get a match at the first character position of the string 'a'. If there were no matches at the first position, Perl would move to the second character position 'b' and attempt the match all over again. Only when all possible paths at all possible character positions have been exhausted does Perl give up and declare `$string =~ /(abd|abc)(df|d|de)/;` to be false.

Even with all this work, regexp matching happens remarkably fast. To speed things up, Perl compiles the regexp into a compact sequence of opcodes that can often fit inside a processor cache. When the code is executed, these opcodes can then run at full throttle and search very quickly.

68.3.5 Extracting matches

The grouping metacharacters `()` also serve another completely different function: they allow the extraction of the parts of a string that matched. This is very useful to find out what matched and for text processing in general. For each grouping, the part that matched inside goes into the special variables `$1`, `$2`, etc. They can be used just as ordinary variables:

```
# extract hours, minutes, seconds
if ($time =~ /(\d\d):(\d\d):(\d\d)/) {    # match hh:mm:ss format
    $hours = $1;
    $minutes = $2;
    $seconds = $3;
}
```

Now, we know that in scalar context, `$time =~ /(\d\d):(\d\d):(\d\d)/` returns a true or false value. In list context, however, it returns the list of matched values `($1,$2,$3)`. So we could write the code more compactly as

```
# extract hours, minutes, seconds
($hours, $minutes, $second) = ($time =~ /(\d\d):(\d\d):(\d\d)/);
```

If the groupings in a regexp are nested, `$1` gets the group with the leftmost opening parenthesis, `$2` the next opening parenthesis, etc. Here is a regexp with nested groups:

```
/(ab(cd|ef)((gi)|j))/;
 1  2      34
```

If this regexp matches, `$1` contains a string starting with `'ab'`, `$2` is either set to `'cd'` or `'ef'`, `$3` equals either `'gi'` or `'j'`, and `$4` is either set to `'gi'`, just like `$3`, or it remains undefined.

For convenience, Perl sets `$+` to the string held by the highest numbered `$1`, `$2`,... that got assigned (and, somewhat related, `$^N` to the value of the `$1`, `$2`,... most-recently assigned; i.e. the `$1`, `$2`,... associated with the rightmost closing parenthesis used in the match).

68.3.6 Backreferences

Closely associated with the matching variables `$1`, `$2`, ... are the *backreferences* `\g1`, `\g2`,... Backreferences are simply matching variables that can be used *inside* a regexp. This is a really nice feature; what matches later in a regexp is made to depend on what matched earlier in the regexp. Suppose we wanted to look for doubled words in a text, like 'the the'. The following regexp finds all 3-letter doubles with a space in between:

```
/\b(\w\w\w)\s\g1\b/;
```

The grouping assigns a value to \g1, so that the same 3-letter sequence is used for both parts.

A similar task is to find words consisting of two identical parts:

```
% simple_grep '^(\w\w\w\w|\w\w\w|\w\w|\w)\g1$' /usr/dict/words
beriberi
booboo
coco
mama
murmur
papa
```

The regexp has a single grouping which considers 4-letter combinations, then 3-letter combinations, etc., and uses `\g1` to look for a repeat. Although `$1` and `\g1` represent the same thing, care should be taken to use matched variables `$1`, `$2`,... only *outside* a regexp and backreferences `\g1`, `\g2`,... only *inside* a regexp; not doing so may lead to surprising and unsatisfactory results.

68.3.7 Relative backreferences

Counting the opening parentheses to get the correct number for a backreference is error-prone as soon as there is more than one capturing group. A more convenient technique became available with Perl 5.10: relative backreferences. To refer to the immediately preceding capture group one now may write `\g{-1}`, the next but last is available via `\g{-2}`, and so on.

Another good reason in addition to readability and maintainability for using relative backreferences is illustrated by the following example, where a simple pattern for matching peculiar strings is used:

```
$a99a = '([a-z])(\d)\g2\g1';   # matches a11a, g22g, x33x, etc.
```

Now that we have this pattern stored as a handy string, we might feel tempted to use it as a part of some other pattern:

```
$line = "code=e99e";
if ($line =~ /^(\w+)=$a99a$/){   # unexpected behavior!
    print "$1 is valid\n";
} else {
    print "bad line: '$line'\n";
}
```

But this doesn't match, at least not the way one might expect. Only after inserting the interpolated `$a99a` and looking at the resulting full text of the regexp is it obvious that the backreferences have backfired. The subexpression `(\w+)` has snatched number 1 and demoted the groups in `$a99a` by one rank. This can be avoided by using relative backreferences:

```
$a99a = '([a-z])(\d)\g{-1}\g{-2}';  # safe for being interpolated
```

68.3.8 Named backreferences

Perl 5.10 also introduced named capture groups and named backreferences. To attach a name to a capturing group, you write either `(?<name>...)` or `(?'name'...)`. The backreference may then be written as `\g{name}`. It is permissible to attach the same name to more than one group, but then only the leftmost one of the eponymous set can be referenced. Outside of the pattern a named capture group is accessible through the `%+` hash.

Assuming that we have to match calendar dates which may be given in one of the three formats yyyy-mm-dd, mm/dd/yyyy or dd.mm.yyyy, we can write three suitable patterns where we use 'd', 'm' and 'y' respectively as the names of the groups capturing the pertaining components of a date. The matching operation combines the three patterns as alternatives:

```
$fmt1 = '(?<y>\d\d\d\d)-(?<m>\d\d)-(?<d>\d\d)';
$fmt2 = '(?<m>\d\d)/(?<d>\d\d)/(?<y>\d\d\d\d)';
$fmt3 = '(?<d>\d\d)\.(?<m>\d\d)\.(?<y>\d\d\d\d)';
for my $d qw( 2006-10-21 15.01.2007 10/31/2005 ){
    if ( $d =~ m{$fmt1|$fmt2|$fmt3} ){
        print "day=$+{d} month=$+{m} year=$+{y}\n";
    }
}
```

If any of the alternatives matches, the hash `%+` is bound to contain the three key-value pairs.

68.3.9 Alternative capture group numbering

Yet another capturing group numbering technique (also as from Perl 5.10) deals with the problem of referring to groups within a set of alternatives. Consider a pattern for matching a time of the day, civil or military style:

```
if ( $time =~ /(\d\d|\d):(\d\d)|(\d\d)(\d\d)/ ){
    # process hour and minute
}
```

Processing the results requires an additional if statement to determine whether `$1` and `$2` or `$3` and `$4` contain the goodies. It would be easier if we could use group numbers 1 and 2 in second alternative as well, and this is exactly what the parenthesized construct `(?|...)`, set around an alternative achieves. Here is an extended version of the previous pattern:

```
if($time =~ /(?|(\d\d|\d):(\d\d)|(\d\d)(\d\d))\s+([A-Z][A-Z][A-Z])/){
    print "hour=$1 minute=$2 zone=$3\n";
}
```

Within the alternative numbering group, group numbers start at the same position for each alternative. After the group, numbering continues with one higher than the maximum reached across all the alternatives.

68.3.10 Position information

In addition to what was matched, Perl also provides the positions of what was matched as contents of the `@-` and `@+` arrays. `$-[0]` is the position of the start of the entire match and `$+[0]` is the position of the end. Similarly, `$-[n]` is the position of the start of the `$n` match and `$+[n]` is the position of the end. If `$n` is undefined, so are `$-[n]` and `$+[n]`. Then this code

```
$x = "Mmm...donut, thought Homer";
$x =~ /^(Mmm|Yech)\.\.\.(donut|peas)/; # matches
foreach $exp (1..$#-) {
    print "Match $exp: '${$exp}' at position ($-[$exp],$+[$exp])\n";
}
```

prints

```
Match 1: 'Mmm' at position (0,3)
Match 2: 'donut' at position (6,11)
```

Even if there are no groupings in a regexp, it is still possible to find out what exactly matched in a string. If you use them, Perl will set `` $` `` to the part of the string before the match, will set `$&` to the part of the string that matched, and will set `$'` to the part of the string after the match. An example:

```
$x = "the cat caught the mouse";
$x =~ /cat/;  # $` = 'the ', $& = 'cat', $' = ' caught the mouse'
$x =~ /the/;  # $` = '', $& = 'the', $' = ' cat caught the mouse'
```

In the second match, `` $` `` equals `''` because the regexp matched at the first character position in the string and stopped; it never saw the second 'the'.

If your code is to run on Perl versions earlier than 5.20, it is worthwhile to note that using `` $` `` and `$'` slows down regexp matching quite a bit, while `$&` slows it down to a lesser extent, because if they are used in one regexp in a program, they are generated for *all* regexps in the program. So if raw performance is a goal of your application, they should be avoided. If you need to extract the corresponding substrings, use `@-` and `@+` instead:

```
$‘ is the same as substr( $x, 0, $-[0] )
$& is the same as substr( $x, $-[0], $+[0]-$-[0] )
$’ is the same as substr( $x, $+[0] )
```

As of Perl 5.10, the `${^PREMATCH}`, `${^MATCH}` and `${^POSTMATCH}` variables may be used. These are only set if the `/p` modifier is present. Consequently they do not penalize the rest of the program. In Perl 5.20, `${^PREMATCH}`, `${^MATCH}` and `${^POSTMATCH}` are available whether the `/p` has been used or not (the modifier is ignored), and `$‘`, `$’` and `$&` do not cause any speed difference.

68.3.11 Non-capturing groupings

A group that is required to bundle a set of alternatives may or may not be useful as a capturing group. If it isn’t, it just creates a superfluous addition to the set of available capture group values, inside as well as outside the regexp. Non-capturing groupings, denoted by `(?:regexp)`, still allow the regexp to be treated as a single unit, but don’t establish a capturing group at the same time. Both capturing and non-capturing groupings are allowed to co-exist in the same regexp. Because there is no extraction, non-capturing groupings are faster than capturing groupings. Non-capturing groupings are also handy for choosing exactly which parts of a regexp are to be extracted to matching variables:

```
# match a number, $1-$4 are set, but we only want $1
/([+-]?\ *(\d+(\.\d*)?|\.\d+)([eE][+-]?\d+)?)/;

# match a number faster , only $1 is set
/([+-]?\ *(?:\d+(?:\.\d*)?|\.\d+)(?:[eE][+-]?\d+)?)/;

# match a number, get $1 = whole number, $2 = exponent
/([+-]?\ *(?:\d+(?:\.\d*)?|\.\d+)(?:[eE]([+-]?\d+))?)/;
```

Non-capturing groupings are also useful for removing nuisance elements gathered from a split operation where parentheses are required for some reason:

```
$x = ’12aba34ba5’;
@num = split /(a|b)+/, $x;    # @num = (’12’,’a’,’34’,’a’,’5’)
@num = split /(?:a|b)+/, $x;  # @num = (’12’,’34’,’5’)
```

In Perl 5.22 and later, all groups within a regexp can be set to non-capturing by using the new `/n` flag:

```
"hello" =~ /(hi|hello)/n; # $1 is not set!
```

See [perlre n], page 990 for more information.

68.3.12 Matching repetitions

The examples in the previous section display an annoying weakness. We were only matching 3-letter words, or chunks of words of 4 letters or less. We’d like to be able to match words or, more generally, strings of any length, without writing out tedious alternatives like `\w\w\w\w|\w\w\w|\w\w|\w`.

This is exactly the problem the *quantifier* metacharacters ?, *, +, and {} were created for. They allow us to delimit the number of repeats for a portion of a regexp we consider to be a match. Quantifiers are put immediately after the character, character class, or grouping that we want to specify. They have the following meanings:

- a? means: match 'a' 1 or 0 times
- a* means: match 'a' 0 or more times, i.e., any number of times
- a+ means: match 'a' 1 or more times, i.e., at least once
- a{n,m} means: match at least n times, but not more than m times.
- a{n,} means: match at least n or more times
- a{n} means: match exactly n times

Here are some examples:

```
/[a-z]+\s+\d*/;  # match a lowercase word, at least one space, and
                 # any number of digits
/(\w+)\s+\g1/;    # match doubled words of arbitrary length
/y(es)?/i;       # matches 'y', 'Y', or a case-insensitive 'yes'
$year =~ /^\d{2,4}$/;  # make sure year is at least 2 but not more
                       # than 4 digits
$year =~ /^\d{4}$|^\d{2}$/; # better match; throw out 3-digit dates
$year =~ /^\d{2}(\d{2})?$/; # same thing written differently.
                            # However, this captures the last two
                            # digits in $1 and the other does not.

% simple_grep '^(\w+)\g1$' /usr/dict/words   # isn't this easier?
beriberi
booboo
coco
mama
murmur
papa
```

For all of these quantifiers, Perl will try to match as much of the string as possible, while still allowing the regexp to succeed. Thus with /a?.../, Perl will first try to match the regexp with the a present; if that fails, Perl will try to match the regexp without the a present. For the quantifier *, we get the following:

```
$x = "the cat in the hat";
$x =~ /^(.*)(cat)(.*)$/; # matches,
                         # $1 = 'the '
                         # $2 = 'cat'
                         # $3 = ' in the hat'
```

Which is what we might expect, the match finds the only cat in the string and locks onto it. Consider, however, this regexp:

```
$x =~ /^(.*)(at)(.*)$/; # matches,
                        # $1 = 'the cat in the h'
                        # $2 = 'at'
                        # $3 = ''   (0 characters match)
```

One might initially guess that Perl would find the at in cat and stop there, but that wouldn't give the longest possible string to the first quantifier .*. Instead, the first quantifier .* grabs as much of the string as possible while still having the regexp match. In this example, that means having the at sequence with the final at in the string. The other important principle illustrated here is that, when there are two or more elements in a regexp, the *leftmost* quantifier, if there is one, gets to grab as much of the string as possible, leaving the rest of the regexp to fight over scraps. Thus in our example, the first quantifier .* grabs most of the string, while the second quantifier .* gets the empty string. Quantifiers that grab as much of the string as possible are called *maximal match* or *greedy* quantifiers.

When a regexp can match a string in several different ways, we can use the principles above to predict which way the regexp will match:

- Principle 0: Taken as a whole, any regexp will be matched at the earliest possible position in the string.
- Principle 1: In an alternation a|b|c..., the leftmost alternative that allows a match for the whole regexp will be the one used.
- Principle 2: The maximal matching quantifiers ?, *, + and {n,m} will in general match as much of the string as possible while still allowing the whole regexp to match.
- Principle 3: If there are two or more elements in a regexp, the leftmost greedy quantifier, if any, will match as much of the string as possible while still allowing the whole regexp to match. The next leftmost greedy quantifier, if any, will try to match as much of the string remaining available to it as possible, while still allowing the whole regexp to match. And so on, until all the regexp elements are satisfied.

As we have seen above, Principle 0 overrides the others. The regexp will be matched as early as possible, with the other principles determining how the regexp matches at that earliest character position.

Here is an example of these principles in action:

```
$x = "The programming republic of Perl";
$x =~ /^(.+)(e|r)(.*)$/;  # matches,
                          # $1 = 'The programming republic of Pe'
                          # $2 = 'r'
                          # $3 = 'l'
```

This regexp matches at the earliest string position, 'T'. One might think that e, being leftmost in the alternation, would be matched, but r produces the longest string in the first quantifier.

```
$x =~ /(m{1,2})(.*)$/;  # matches,
                        # $1 = 'mm'
                        # $2 = 'ing republic of Perl'
```

Here, The earliest possible match is at the first 'm' in programming. m{1,2} is the first quantifier, so it gets to match a maximal mm.

```
$x =~ /.*(m{1,2})(.*)$/;  # matches,
                          # $1 = 'm'
                          # $2 = 'ing republic of Perl'
```

Here, the regexp matches at the start of the string. The first quantifier .* grabs as much as possible, leaving just a single 'm' for the second quantifier m{1,2}.

```
$x =~ /(.?)(m{1,2})(.*)$/;  # matches,
                            # $1 = 'a'
                            # $2 = 'mm'
                            # $3 = 'ing republic of Perl'
```

Here, `.?` eats its maximal one character at the earliest possible position in the string, `'a'` in `programming`, leaving `m{1,2}` the opportunity to match both `m`'s. Finally,

```
"aXXXb" =~ /(X*)/; # matches with $1 = ''
```

because it can match zero copies of `'X'` at the beginning of the string. If you definitely want to match at least one `'X'`, use `X+`, not `X*`.

Sometimes greed is not good. At times, we would like quantifiers to match a *minimal* piece of string, rather than a maximal piece. For this purpose, Larry Wall created the *minimal match* or *non-greedy* quantifiers `??`, `*?`, `+?`, and `{}?`. These are the usual quantifiers with a `?` appended to them. They have the following meanings:

- `a??` means: match 'a' 0 or 1 times. Try 0 first, then 1.
- `a*?` means: match 'a' 0 or more times, i.e., any number of times, but as few times as possible
- `a+?` means: match 'a' 1 or more times, i.e., at least once, but as few times as possible
- `a{n,m}?` means: match at least `n` times, not more than `m` times, as few times as possible
- `a{n,}?` means: match at least `n` times, but as few times as possible
- `a{n}?` means: match exactly `n` times. Because we match exactly `n` times, `a{n}?` is equivalent to `a{n}` and is just there for notational consistency.

Let's look at the example above, but with minimal quantifiers:

```
$x = "The programming republic of Perl";
$x =~ /^(.+?)(e|r)(.*)$/; # matches,
                          # $1 = 'Th'
                          # $2 = 'e'
                          # $3 = ' programming republic of Perl'
```

The minimal string that will allow both the start of the string `^` and the alternation to match is `Th`, with the alternation `e|r` matching `e`. The second quantifier `.*` is free to gobble up the rest of the string.

```
$x =~ /(m{1,2}?)(.*?)$/;  # matches,
                          # $1 = 'm'
                          # $2 = 'ming republic of Perl'
```

The first string position that this regexp can match is at the first `'m'` in `programming`. At this position, the minimal `m{1,2}?` matches just one `'m'`. Although the second quantifier `.*?` would prefer to match no characters, it is constrained by the end-of-string anchor `$` to match the rest of the string.

```
$x =~ /(.*?)(m{1,2}?)(.*)$/;  # matches,
                              # $1 = 'The progra'
                              # $2 = 'm'
                              # $3 = 'ming republic of Perl'
```

In this regexp, you might expect the first minimal quantifier `.*?` to match the empty string, because it is not constrained by a `^` anchor to match the beginning of the word.

Principle 0 applies here, however. Because it is possible for the whole regexp to match at the start of the string, it *will* match at the start of the string. Thus the first quantifier has to match everything up to the first m. The second minimal quantifier matches just one m and the third quantifier matches the rest of the string.

```
$x =~ /(.??)(m{1,2})(.*)$/;  # matches,
                             # $1 = 'a'
                             # $2 = 'mm'
                             # $3 = 'ing republic of Perl'
```

Just as in the previous regexp, the first quantifier .?? can match earliest at position 'a', so it does. The second quantifier is greedy, so it matches mm, and the third matches the rest of the string.

We can modify principle 3 above to take into account non-greedy quantifiers:

- Principle 3: If there are two or more elements in a regexp, the leftmost greedy (non-greedy) quantifier, if any, will match as much (little) of the string as possible while still allowing the whole regexp to match. The next leftmost greedy (non-greedy) quantifier, if any, will try to match as much (little) of the string remaining available to it as possible, while still allowing the whole regexp to match. And so on, until all the regexp elements are satisfied.

Just like alternation, quantifiers are also susceptible to backtracking. Here is a step-by-step analysis of the example

```
$x = "the cat in the hat";
$x =~ /^(.*)(at)(.*)$/; # matches,
                        # $1 = 'the cat in the h'
                        # $2 = 'at'
                        # $3 = ''   (0 matches)
```

0

Start with the first letter in the string 't'.

1

The first quantifier '.*' starts out by matching the whole string 'the cat in the hat'.

2

'a' in the regexp element 'at' doesn't match the end of the string. Backtrack one character.

3

'a' in the regexp element 'at' still doesn't match the last letter of the string 't', so backtrack one more character.

4

Now we can match the 'a' and the 't'.

5

Move on to the third element '.*'. Since we are at the end of the string and '.*' can match 0 times, assign it the empty string.

We are done!

Most of the time, all this moving forward and backtracking happens quickly and searching is fast. There are some pathological regexps, however, whose execution time exponentially grows with the size of the string. A typical structure that blows up in your face is of the form

```
/(a|b+)*/;
```

The problem is the nested indeterminate quantifiers. There are many different ways of partitioning a string of length n between the `+` and `*`: one repetition with `b+` of length n, two repetitions with the first `b+` length k and the second with length n-k, m repetitions whose bits add up to length n, etc. In fact there are an exponential number of ways to partition a string as a function of its length. A regexp may get lucky and match early in the process, but if there is no match, Perl will try *every* possibility before giving up. So be careful with nested `*`'s, `{n,m}`'s, and `+`'s. The book *Mastering Regular Expressions* by Jeffrey Friedl gives a wonderful discussion of this and other efficiency issues.

68.3.13 Possessive quantifiers

Backtracking during the relentless search for a match may be a waste of time, particularly when the match is bound to fail. Consider the simple pattern

```
/^\w+\s+\w+$/; # a word, spaces, a word
```

Whenever this is applied to a string which doesn't quite meet the pattern's expectations such as `"abc "` or `"abc def "`, the regex engine will backtrack, approximately once for each character in the string. But we know that there is no way around taking *all* of the initial word characters to match the first repetition, that *all* spaces must be eaten by the middle part, and the same goes for the second word.

With the introduction of the *possessive quantifiers* in Perl 5.10, we have a way of instructing the regex engine not to backtrack, with the usual quantifiers with a `+` appended to them. This makes them greedy as well as stingy; once they succeed they won't give anything back to permit another solution. They have the following meanings:

- `a{n,m}+` means: match at least `n` times, not more than `m` times, as many times as possible, and don't give anything up. `a?+` is short for `a{0,1}+`
- `a{n,}+` means: match at least `n` times, but as many times as possible, and don't give anything up. `a*+` is short for `a{0,}+` and `a++` is short for `a{1,}+`.
- `a{n}+` means: match exactly `n` times. It is just there for notational consistency.

These possessive quantifiers represent a special case of a more general concept, the *independent subexpression*, see below.

As an example where a possessive quantifier is suitable we consider matching a quoted string, as it appears in several programming languages. The backslash is used as an escape character that indicates that the next character is to be taken literally, as another character for the string. Therefore, after the opening quote, we expect a (possibly empty) sequence of alternatives: either some character except an unescaped quote or backslash or an escaped character.

```
/"(?:[^"\\]++|\\.)*+"/;
```

68.3.14 Building a regexp

At this point, we have all the basic regexp concepts covered, so let's give a more involved example of a regular expression. We will build a regexp that matches numbers.

The first task in building a regexp is to decide what we want to match and what we want to exclude. In our case, we want to match both integers and floating point numbers and we want to reject any string that isn't a number.

The next task is to break the problem down into smaller problems that are easily converted into a regexp.

The simplest case is integers. These consist of a sequence of digits, with an optional sign in front. The digits we can represent with `\d+` and the sign can be matched with `[+-]`. Thus the integer regexp is

```
 /[+-]?\d+/;  # matches integers
```

A floating point number potentially has a sign, an integral part, a decimal point, a fractional part, and an exponent. One or more of these parts is optional, so we need to check out the different possibilities. Floating point numbers which are in proper form include 123., 0.345, .34, -1e6, and 25.4E-72. As with integers, the sign out front is completely optional and can be matched by `[+-]?`. We can see that if there is no exponent, floating point numbers must have a decimal point, otherwise they are integers. We might be tempted to model these with `\d*\.\d*`, but this would also match just a single decimal point, which is not a number. So the three cases of floating point number without exponent are

```
/[+-]?\d+\./;  # 1., 321., etc.
/[+-]?\.\d+/;  # .1, .234, etc.
/[+-]?\d+\.\d+/;  # 1.0, 30.56, etc.
```

These can be combined into a single regexp with a three-way alternation:

```
/[+-]?(\d+\.\d+|\d+\.|\.\d+)/;  # floating point, no exponent
```

In this alternation, it is important to put `'\d+\.\d+'` before `'\d+\.'`. If `'\d+\.'` were first, the regexp would happily match that and ignore the fractional part of the number.

Now consider floating point numbers with exponents. The key observation here is that *both* integers and numbers with decimal points are allowed in front of an exponent. Then exponents, like the overall sign, are independent of whether we are matching numbers with or without decimal points, and can be 'decoupled' from the mantissa. The overall form of the regexp now becomes clear:

```
 /^(optional sign)(integer | f.p. mantissa)(optional exponent)$/;
```

The exponent is an `e` or `E`, followed by an integer. So the exponent regexp is

```
/[eE][+-]?\d+/;  # exponent
```

Putting all the parts together, we get a regexp that matches numbers:

```
/^[+-]?(\d+\.\d+|\d+\.|\.\d+|\d+)([eE][+-]?\d+)?$/;  # Ta da!
```

Long regexps like this may impress your friends, but can be hard to decipher. In complex situations like this, the `//x` modifier for a match is invaluable. It allows one to put nearly arbitrary whitespace and comments into a regexp without affecting their meaning. Using it, we can rewrite our 'extended' regexp in the more pleasing form

```
/^
   [+-]?         # first, match an optional sign
```

```
    (             # then match integers or f.p. mantissas:
        \d+\.\d+  # mantissa of the form a.b
       |\d+\.     # mantissa of the form a.
       |\.\d+     # mantissa of the form .b
       |\d+       # integer of the form a
    )
    ([eE][+-]?\d+)?  # finally, optionally match an exponent
$/x;
```

If whitespace is mostly irrelevant, how does one include space characters in an extended regexp? The answer is to backslash it '\ ' or put it in a character class [ ]. The same thing goes for pound signs: use \# or [#]. For instance, Perl allows a space between the sign and the mantissa or integer, and we could add this to our regexp as follows:

```
/^
    [+-]?\ *      # first, match an optional sign *and space*
    (             # then match integers or f.p. mantissas:
        \d+\.\d+  # mantissa of the form a.b
       |\d+\.     # mantissa of the form a.
       |\.\d+     # mantissa of the form .b
       |\d+       # integer of the form a
    )
    ([eE][+-]?\d+)?  # finally, optionally match an exponent
$/x;
```

In this form, it is easier to see a way to simplify the alternation. Alternatives 1, 2, and 4 all start with \d+, so it could be factored out:

```
/^
    [+-]?\ *      # first, match an optional sign
    (             # then match integers or f.p. mantissas:
        \d+       # start out with a ...
        (
            \.\d* # mantissa of the form a.b or a.
        )?        # ? takes care of integers of the form a
       |\.\d+     # mantissa of the form .b
    )
    ([eE][+-]?\d+)?  # finally, optionally match an exponent
$/x;
```

or written in the compact form,

```
 /^[+-]?\ *(\d+(\.\d*)?|\.\d+)([eE][+-]?\d+)?$/;
```

This is our final regexp. To recap, we built a regexp by

- specifying the task in detail,
- breaking down the problem into smaller parts,
- translating the small parts into regexps,
- combining the regexps,
- and optimizing the final combined regexp.

These are also the typical steps involved in writing a computer program. This makes perfect sense, because regular expressions are essentially programs written in a little computer language that specifies patterns.

68.3.15 Using regular expressions in Perl

The last topic of Part 1 briefly covers how regexps are used in Perl programs. Where do they fit into Perl syntax?

We have already introduced the matching operator in its default `/regexp/` and arbitrary delimiter `m!regexp!` forms. We have used the binding operator `=~` and its negation `!~` to test for string matches. Associated with the matching operator, we have discussed the single line `//s`, multi-line `//m`, case-insensitive `//i` and extended `//x` modifiers. There are a few more things you might want to know about matching operators.

68.3.15.1 Prohibiting substitution

If you change `$pattern` after the first substitution happens, Perl will ignore it. If you don't want any substitutions at all, use the special delimiter `m''`:

```
@pattern = ('Seuss');
while (<>) {
    print if m'@pattern';  # matches literal '@pattern', not 'Seuss'
}
```

Similar to strings, `m''` acts like apostrophes on a regexp; all other `m` delimiters act like quotes. If the regexp evaluates to the empty string, the regexp in the *last successful match* is used instead. So we have

```
"dog" =~ /d/;  # 'd' matches
"dogbert =~ //;  # this matches the 'd' regexp used before
```

68.3.15.2 Global matching

The final two modifiers we will discuss here, `//g` and `//c`, concern multiple matches. The modifier `//g` stands for global matching and allows the matching operator to match within a string as many times as possible. In scalar context, successive invocations against a string will have `//g` jump from match to match, keeping track of position in the string as it goes along. You can get or set the position with the `pos()` function.

The use of `//g` is shown in the following example. Suppose we have a string that consists of words separated by spaces. If we know how many words there are in advance, we could extract the words using groupings:

```
$x = "cat dog house"; # 3 words
$x =~ /^\s*(\w+)\s+(\w+)\s+(\w+)\s*$/; # matches,
                                       # $1 = 'cat'
                                       # $2 = 'dog'
                                       # $3 = 'house'
```

But what if we had an indeterminate number of words? This is the sort of task `//g` was made for. To extract all words, form the simple regexp `(\w+)` and loop over all matches with `/(\w+)/g`:

```
while ($x =~ /(\w+)/g) {
    print "Word is $1, ends at position ", pos $x, "\n";
}
```

prints

```
Word is cat, ends at position 3
Word is dog, ends at position 7
Word is house, ends at position 13
```

A failed match or changing the target string resets the position. If you don’t want the position reset after failure to match, add the `//c`, as in `/regexp/gc`. The current position in the string is associated with the string, not the regexp. This means that different strings have different positions and their respective positions can be set or read independently.

In list context, `//g` returns a list of matched groupings, or if there are no groupings, a list of matches to the whole regexp. So if we wanted just the words, we could use

```
@words = ($x =~ /(\w+)/g);  # matches,
                            # $words[0] = ’cat’
                            # $words[1] = ’dog’
                            # $words[2] = ’house’
```

Closely associated with the `//g` modifier is the `\G` anchor. The `\G` anchor matches at the point where the previous `//g` match left off. `\G` allows us to easily do context-sensitive matching:

```
$metric = 1;  # use metric units
...
$x = <FILE>;  # read in measurement
$x =~ /^([+-]?\d+)\s*/g;  # get magnitude
$weight = $1;
if ($metric) { # error checking
    print "Units error!" unless $x =~ /\Gkg\./g;
}
else {
    print "Units error!" unless $x =~ /\Glbs\./g;
}
$x =~ /\G\s+(widget|sprocket)/g;  # continue processing
```

The combination of `//g` and `\G` allows us to process the string a bit at a time and use arbitrary Perl logic to decide what to do next. Currently, the `\G` anchor is only fully supported when used to anchor to the start of the pattern.

`\G` is also invaluable in processing fixed-length records with regexps. Suppose we have a snippet of coding region DNA, encoded as base pair letters `ATCGTTGAAT...` and we want to find all the stop codons `TGA`. In a coding region, codons are 3-letter sequences, so we can think of the DNA snippet as a sequence of 3-letter records. The naive regexp

```
# expanded, this is "ATC GTT GAA TGC AAA TGA CAT GAC"
$dna = "ATCGTTGAATGCAAATGACATGAC";
$dna =~ /TGA/;
```

doesn’t work; it may match a `TGA`, but there is no guarantee that the match is aligned with codon boundaries, e.g., the substring `GTT GAA` gives a match. A better solution is

```
while ($dna =~ /(\w\w\w)*?TGA/g) {  # note the minimal *?
    print "Got a TGA stop codon at position ", pos $dna, "\n";
}
```

which prints

```
Got a TGA stop codon at position 18
Got a TGA stop codon at position 23
```

Position 18 is good, but position 23 is bogus. What happened?

The answer is that our regexp works well until we get past the last real match. Then the regexp will fail to match a synchronized TGA and start stepping ahead one character position at a time, not what we want. The solution is to use \G to anchor the match to the codon alignment:

```
while ($dna =~ /\G(\w\w\w)*?TGA/g) {
    print "Got a TGA stop codon at position ", pos $dna, "\n";
}
```

This prints

```
Got a TGA stop codon at position 18
```

which is the correct answer. This example illustrates that it is important not only to match what is desired, but to reject what is not desired.

(There are other regexp modifiers that are available, such as //o, but their specialized uses are beyond the scope of this introduction.)

68.3.15.3 Search and replace

Regular expressions also play a big role in *search and replace* operations in Perl. Search and replace is accomplished with the s/// operator. The general form is s/regexp/replacement/modifiers, with everything we know about regexps and modifiers applying in this case as well. The replacement is a Perl double-quoted string that replaces in the string whatever is matched with the regexp. The operator =~ is also used here to associate a string with s///. If matching against $_, the $_ =~ can be dropped. If there is a match, s/// returns the number of substitutions made; otherwise it returns false. Here are a few examples:

```
$x = "Time to feed the cat!";
$x =~ s/cat/hacker/;   # $x contains "Time to feed the hacker!"
if ($x =~ s/^(Time.*hacker)!$/$1 now!/) {
    $more_insistent = 1;
}
$y = "'quoted words'";
$y =~ s/^'(.*)'$/$1/;  # strip single quotes,
                       # $y contains "quoted words"
```

In the last example, the whole string was matched, but only the part inside the single quotes was grouped. With the s/// operator, the matched variables $1, $2, etc. are immediately available for use in the replacement expression, so we use $1 to replace the quoted string with just what was quoted. With the global modifier, s///g will search and replace all occurrences of the regexp in the string:

```
$x = "I batted 4 for 4";
$x =~ s/4/four/;   # doesn't do it all:
                   # $x contains "I batted four for 4"
```

```
$x = "I batted 4 for 4";
$x =~ s/4/four/g;  # does it all:
                   # $x contains "I batted four for four"
```

If you prefer 'regex' over 'regexp' in this tutorial, you could use the following program to replace it:

```
% cat > simple_replace
#!/usr/bin/perl
$regexp = shift;
$replacement = shift;
while (<>) {
    s/$regexp/$replacement/g;
    print;
}
^D

% simple_replace regexp regex perlretut.pod
```

In `simple_replace` we used the `s///g` modifier to replace all occurrences of the regexp on each line. (Even though the regular expression appears in a loop, Perl is smart enough to compile it only once.) As with `simple_grep`, both the `print` and the `s/$regexp/$replacement/g` use `$_` implicitly.

If you don't want `s///` to change your original variable you can use the non-destructive substitute modifier, `s///r`. This changes the behavior so that `s///r` returns the final substituted string (instead of the number of substitutions):

```
$x = "I like dogs.";
$y = $x =~ s/dogs/cats/r;
print "$x $y\n";
```

That example will print "I like dogs. I like cats". Notice the original `$x` variable has not been affected. The overall result of the substitution is instead stored in `$y`. If the substitution doesn't affect anything then the original string is returned:

```
$x = "I like dogs.";
$y = $x =~ s/elephants/cougars/r;
print "$x $y\n"; # prints "I like dogs. I like dogs."
```

One other interesting thing that the `s///r` flag allows is chaining substitutions:

```
$x = "Cats are great.";
print $x =~ s/Cats/Dogs/r =~ s/Dogs/Frogs/r =~
    s/Frogs/Hedgehogs/r, "\n";
# prints "Hedgehogs are great."
```

A modifier available specifically to search and replace is the `s///e` evaluation modifier. `s///e` treats the replacement text as Perl code, rather than a double-quoted string. The value that the code returns is substituted for the matched substring. `s///e` is useful if you need to do a bit of computation in the process of replacing text. This example counts character frequencies in a line:

```
$x = "Bill the cat";
$x =~ s/(.)/$chars{$1}++;$1/eg; # final $1 replaces char with itself
```

```
print "frequency of '$_' is $chars{$_}\n"
    foreach (sort {$chars{$b} <=> $chars{$a}} keys %chars);
```

This prints

```
frequency of ' ' is 2
frequency of 't' is 2
frequency of 'l' is 2
frequency of 'B' is 1
frequency of 'c' is 1
frequency of 'e' is 1
frequency of 'h' is 1
frequency of 'i' is 1
frequency of 'a' is 1
```

As with the match `m//` operator, `s///` can use other delimiters, such as `s!!!` and `s{}{}`, and even `s{}//`. If single quotes are used `s'''`, then the regexp and replacement are treated as single-quoted strings and there are no variable substitutions. `s///` in list context returns the same thing as in scalar context, i.e., the number of matches.

68.3.15.4 The split function

The `split()` function is another place where a regexp is used. `split /regexp/, string, limit` separates the `string` operand into a list of substrings and returns that list. The regexp must be designed to match whatever constitutes the separators for the desired substrings. The `limit`, if present, constrains splitting into no more than `limit` number of strings. For example, to split a string into words, use

```
$x = "Calvin and Hobbes";
@words = split /\s+/, $x;  # $word[0] = 'Calvin'
                           # $word[1] = 'and'
                           # $word[2] = 'Hobbes'
```

If the empty regexp `//` is used, the regexp always matches and the string is split into individual characters. If the regexp has groupings, then the resulting list contains the matched substrings from the groupings as well. For instance,

```
$x = "/usr/bin/perl";
@dirs = split m!/!, $x;  # $dirs[0] = ''
                         # $dirs[1] = 'usr'
                         # $dirs[2] = 'bin'
                         # $dirs[3] = 'perl'
@parts = split m!(/)!, $x;  # $parts[0] = ''
                            # $parts[1] = '/'
                            # $parts[2] = 'usr'
                            # $parts[3] = '/'
                            # $parts[4] = 'bin'
                            # $parts[5] = '/'
                            # $parts[6] = 'perl'
```

Since the first character of $x matched the regexp, `split` prepended an empty initial element to the list.

If you have read this far, congratulations! You now have all the basic tools needed to use regular expressions to solve a wide range of text processing problems. If this is your first time through the tutorial, why not stop here and play around with regexps a while.... Part 2 concerns the more esoteric aspects of regular expressions and those concepts certainly aren't needed right at the start.

68.4 Part 2: Power tools

OK, you know the basics of regexps and you want to know more. If matching regular expressions is analogous to a walk in the woods, then the tools discussed in Part 1 are analogous to topo maps and a compass, basic tools we use all the time. Most of the tools in part 2 are analogous to flare guns and satellite phones. They aren't used too often on a hike, but when we are stuck, they can be invaluable.

What follows are the more advanced, less used, or sometimes esoteric capabilities of Perl regexps. In Part 2, we will assume you are comfortable with the basics and concentrate on the advanced features.

68.4.1 More on characters, strings, and character classes

There are a number of escape sequences and character classes that we haven't covered yet.

There are several escape sequences that convert characters or strings between upper and lower case, and they are also available within patterns. `\l` and `\u` convert the next character to lower or upper case, respectively:

```
$x = "perl";
$string =~ /\u$x/;  # matches 'Perl' in $string
$x = "M(rs?|s)\\."; # note the double backslash
$string =~ /\l$x/;  # matches 'mr.', 'mrs.', and 'ms.',
```

A `\L` or `\U` indicates a lasting conversion of case, until terminated by `\E` or thrown over by another `\U` or `\L`:

```
$x = "This word is in lower case:\L SHOUT\E";
$x =~ /shout/;       # matches
$x = "I STILL KEYPUNCH CARDS FOR MY 360"
$x =~ /\Ukeypunch/;  # matches punch card string
```

If there is no `\E`, case is converted until the end of the string. The regexps `\L\u$word` or `\u\L$word` convert the first character of `$word` to uppercase and the rest of the characters to lowercase.

Control characters can be escaped with `\c`, so that a control-Z character would be matched with `\cZ`. The escape sequence `\Q`...`\E` quotes, or protects most non-alphabetic characters. For instance,

```
$x = "\QThat !^*&%~& cat!";
$x =~ /\Q!^*&%~&\E/;  # check for rough language
```

It does not protect `$` or `@`, so that variables can still be substituted.

`\Q`, `\L`, `\l`, `\U`, `\u` and `\E` are actually part of double-quotish syntax, and not part of regexp syntax proper. They will work if they appear in a regular expression embedded directly in a program, but not when contained in a string that is interpolated in a pattern.

Perl regexps can handle more than just the standard ASCII character set. Perl supports *Unicode*, a standard for representing the alphabets from virtually all of the world's written languages, and a host of symbols. Perl's text strings are Unicode strings, so they can contain characters with a value (codepoint or character number) higher than 255.

What does this mean for regexps? Well, regexp users don't need to know much about Perl's internal representation of strings. But they do need to know 1) how to represent Unicode characters in a regexp and 2) that a matching operation will treat the string to be searched as a sequence of characters, not bytes. The answer to 1) is that Unicode characters greater than `chr(255)` are represented using the `\x{hex}` notation, because \x hex (without curly braces) doesn't go further than 255. (Starting in Perl 5.14, if you're an octal fan, you can also use `\o{oct}`.)

```
/\x{263a}/;  # match a Unicode smiley face :)
```

NOTE: In Perl 5.6.0 it used to be that one needed to say `use utf8` to use any Unicode features. This is no more the case: for almost all Unicode processing, the explicit `utf8` pragma is not needed. (The only case where it matters is if your Perl script is in Unicode and encoded in UTF-8, then an explicit `use utf8` is needed.)

Figuring out the hexadecimal sequence of a Unicode character you want or deciphering someone else's hexadecimal Unicode regexp is about as much fun as programming in machine code. So another way to specify Unicode characters is to use the *named character* escape sequence `\N{`*`name`*`}`. *name* is a name for the Unicode character, as specified in the Unicode standard. For instance, if we wanted to represent or match the astrological sign for the planet Mercury, we could use

```
$x = "abc\N{MERCURY}def";
$x =~ /\N{MERCURY}/;   # matches
```

One can also use "short" names:

```
print "\N{GREEK SMALL LETTER SIGMA} is called sigma.\n";
print "\N{greek:Sigma} is an upper-case sigma.\n";
```

You can also restrict names to a certain alphabet by specifying the `charnames` pragma:

```
use charnames qw(greek);
print "\N{sigma} is Greek sigma\n";
```

An index of character names is available on-line from the Unicode Consortium, `http://www.unicode.org/charts/charindex.html`; explanatory material with links to other resources at `http://www.unicode.org/standard/where`.

The answer to requirement 2) is that a regexp (mostly) uses Unicode characters. The "mostly" is for messy backward compatibility reasons, but starting in Perl 5.14, any regex compiled in the scope of a `use feature 'unicode_strings'` (which is automatically turned on within the scope of a `use 5.012` or higher) will turn that "mostly" into "always". If you want to handle Unicode properly, you should ensure that `'unicode_strings'` is turned on. Internally, this is encoded to bytes using either UTF-8 or a native 8 bit encoding, depending on the history of the string, but conceptually it is a sequence of characters, not bytes. See Section 84.1 [perlunitut NAME], page 1367 for a tutorial about that.

Let us now discuss Unicode character classes, most usually called "character properties". These are represented by the `\p{name}` escape sequence. Closely associated is the `\P{name}` property, which is the negation of the `\p{name}` one. For example, to match lower and uppercase characters,

```
$x = "BOB";
$x =~ /^\p{IsUpper}/;   # matches, uppercase char class
$x =~ /^\P{IsUpper}/;   # doesn't match, char class sans uppercase
$x =~ /^\p{IsLower}/;   # doesn't match, lowercase char class
$x =~ /^\P{IsLower}/;   # matches, char class sans lowercase
```

(The "Is" is optional.)

There are many, many Unicode character properties. For the full list see `perluniprops`. Most of them have synonyms with shorter names, also listed there. Some synonyms are a single character. For these, you can drop the braces. For instance, `\pM` is the same thing as `\p{Mark}`, meaning things like accent marks.

The Unicode `\p{Script}` property is used to categorize every Unicode character into the language script it is written in. For example, English, French, and a bunch of other European languages are written in the Latin script. But there is also the Greek script, the Thai script, the Katakana script, etc. You can test whether a character is in a particular script with, for example `\p{Latin}`, `\p{Greek}`, or `\p{Katakana}`. To test if it isn't in the Balinese script, you would use `\P{Balinese}`.

What we have described so far is the single form of the `\p{...}` character classes. There is also a compound form which you may run into. These look like `\p{name=value}` or `\p{name:value}` (the equals sign and colon can be used interchangeably). These are more general than the single form, and in fact most of the single forms are just Perl-defined shortcuts for common compound forms. For example, the script examples in the previous paragraph could be written equivalently as `\p{Script=Latin}`, `\p{Script:Greek}`, `\p{script=katakana}`, and `\P{script=balinese}` (case is irrelevant between the `{}` braces). You may never have to use the compound forms, but sometimes it is necessary, and their use can make your code easier to understand.

`\X` is an abbreviation for a character class that comprises a Unicode *extended grapheme cluster*. This represents a "logical character": what appears to be a single character, but may be represented internally by more than one. As an example, using the Unicode full names, e.g., `A + COMBINING RING` is a grapheme cluster with base character `A` and combining character `COMBINING RING`, which translates in Danish to A with the circle atop it, as in the word Ångstrom.

For the full and latest information about Unicode see the latest Unicode standard, or the Unicode Consortium's website `http://www.unicode.org`

As if all those classes weren't enough, Perl also defines POSIX-style character classes. These have the form `[:name:]`, with `name` the name of the POSIX class. The POSIX classes are `alpha`, `alnum`, `ascii`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, and `xdigit`, and two extensions, `word` (a Perl extension to match `\w`), and `blank` (a GNU extension). The `//a` modifier restricts these to matching just in the ASCII range; otherwise they can match the same as their corresponding Perl Unicode classes: `[:upper:]` is the same as `\p{IsUpper}`, etc. (There are some exceptions and gotchas with this; see Section 61.1 [perlrecharclass NAME], page 1059 for a full discussion.) The `[:digit:]`, `[:word:]`, and `[:space:]` correspond to the familiar `\d`, `\w`, and `\s` character classes. To negate a POSIX class, put a `^` in front of the name, so that, e.g., `[:^digit:]` corresponds to `\D` and, under Unicode, `\P{IsDigit}`. The Unicode and POSIX character classes can be used just like

\d, with the exception that POSIX character classes can only be used inside of a character class:

```
/\s+[abc[:digit:]xyz]\s*/;  # match a,b,c,x,y,z, or a digit
/^=item\s[[:digit:]]/;      # match '=item',
                            # followed by a space and a digit
/\s+[abc\p{IsDigit}xyz]\s+/;  # match a,b,c,x,y,z, or a digit
/^=item\s\p{IsDigit}/;        # match '=item',
                              # followed by a space and a digit
```

Whew! That is all the rest of the characters and character classes.

68.4.2 Compiling and saving regular expressions

In Part 1 we mentioned that Perl compiles a regexp into a compact sequence of opcodes. Thus, a compiled regexp is a data structure that can be stored once and used again and again. The regexp quote qr// does exactly that: qr/string/ compiles the string as a regexp and transforms the result into a form that can be assigned to a variable:

```
$reg = qr/foo+bar?/;  # reg contains a compiled regexp
```

Then $reg can be used as a regexp:

```
$x = "fooooba";
$x =~ $reg;     # matches, just like /foo+bar?/
$x =~ /$reg/;   # same thing, alternate form
```

$reg can also be interpolated into a larger regexp:

```
$x =~ /(abc)?$reg/;  # still matches
```

As with the matching operator, the regexp quote can use different delimiters, e.g., qr!!, qr{} or qr~~. Apostrophes as delimiters (qr'') inhibit any interpolation.

Pre-compiled regexps are useful for creating dynamic matches that don't need to be recompiled each time they are encountered. Using pre-compiled regexps, we write a grep_step program which greps for a sequence of patterns, advancing to the next pattern as soon as one has been satisfied.

```
% cat > grep_step
#!/usr/bin/perl
# grep_step - match <number> regexps, one after the other
# usage: multi_grep <number> regexp1 regexp2 ... file1 file2 ...

$number = shift;
$regexp[$_] = shift foreach (0..$number-1);
@compiled = map qr/$_/, @regexp;
while ($line = <>) {
    if ($line =~ /$compiled[0]/) {
        print $line;
        shift @compiled;
        last unless @compiled;
    }
}
^D
```

```
% grep_step 3 shift print last grep_step
$number = shift;
        print $line;
        last unless @compiled;
```

Storing pre-compiled regexps in an array `@compiled` allows us to simply loop through the regexps without any recompilation, thus gaining flexibility without sacrificing speed.

68.4.3 Composing regular expressions at runtime

Backtracking is more efficient than repeated tries with different regular expressions. If there are several regular expressions and a match with any of them is acceptable, then it is possible to combine them into a set of alternatives. If the individual expressions are input data, this can be done by programming a join operation. We’ll exploit this idea in an improved version of the `simple_grep` program: a program that matches multiple patterns:

```
% cat > multi_grep
#!/usr/bin/perl
# multi_grep - match any of <number> regexps
# usage: multi_grep <number> regexp1 regexp2 ... file1 file2 ...

$number = shift;
$regexp[$_] = shift foreach (0..$number-1);
$pattern = join ’|’, @regexp;

while ($line = <>) {
    print $line if $line =~ /$pattern/;
}
^D

% multi_grep 2 shift for multi_grep
$number = shift;
$regexp[$_] = shift foreach (0..$number-1);
```

Sometimes it is advantageous to construct a pattern from the *input* that is to be analyzed and use the permissible values on the left hand side of the matching operations. As an example for this somewhat paradoxical situation, let’s assume that our input contains a command verb which should match one out of a set of available command verbs, with the additional twist that commands may be abbreviated as long as the given string is unique. The program below demonstrates the basic algorithm.

```
% cat > keymatch
#!/usr/bin/perl
$kwds = ’copy compare list print’;
while( $cmd = <> ){
    $cmd =~ s/^\s+|\s+$//g;  # trim leading and trailing spaces
    if( ( @matches = $kwds =~ /\b$cmd\w*/g ) == 1 ){
        print "command: ’@matches’\n";
    } elsif( @matches == 0 ){
        print "no such command: ’$cmd’\n";
    } else {
        print "not unique: '$cmd' (could be one of: @matches)\n";
    }
}
^D

% keymatch
li
command: 'list'
co
not unique: 'co' (could be one of: copy compare)
printer
no such command: 'printer'
```

Rather than trying to match the input against the keywords, we match the combined set of keywords against the input. The pattern matching operation `$kwds =~ /\b($cmd\w*)/g` does several things at the same time. It makes sure that the given command begins where a keyword begins (`\b`). It tolerates abbreviations due to the added `\w*`. It tells us the number of matches (`scalar @matches`) and all the keywords that were actually matched. You could hardly ask for more.

68.4.4 Embedding comments and modifiers in a regular expression

Starting with this section, we will be discussing Perl's set of *extended patterns*. These are extensions to the traditional regular expression syntax that provide powerful new tools for pattern matching. We have already seen extensions in the form of the minimal matching constructs `??`, `*?`, `+?`, `{n,m}?`, and `{n,}?`. Most of the extensions below have the form `(?char...)`, where the `char` is a character that determines the type of extension.

The first extension is an embedded comment `(?#text)`. This embeds a comment into the regular expression without affecting its meaning. The comment should not have any closing parentheses in the text. An example is

```
    /(?# Match an integer:)[+-]?\d+/;
```

This style of commenting has been largely superseded by the raw, freeform commenting that is allowed with the `//x` modifier.

Most modifiers, such as `//i`, `//m`, `//s` and `//x` (or any combination thereof) can also be embedded in a regexp using `(?i)`, `(?m)`, `(?s)`, and `(?x)`. For instance,

```
    /(?i)yes/;  # match 'yes' case insensitively
    /yes/i;     # same thing
    /(?x)(          # freeform version of an integer regexp
             [+-]?  # match an optional sign
             \d+    # match a sequence of digits
         )
    /x;
```

Embedded modifiers can have two important advantages over the usual modifiers. Embedded modifiers allow a custom set of modifiers to *each* regexp pattern. This is great for matching an array of regexps that must have different modifiers:

```
    $pattern[0] = '(?i)doctor';
```

```
$pattern[1] = 'Johnson';
...
while (<>) {
    foreach $patt (@pattern) {
        print if /$patt/;
    }
}
```

The second advantage is that embedded modifiers (except `//p`, which modifies the entire regexp) only affect the regexp inside the group the embedded modifier is contained in. So grouping can be used to localize the modifier's effects:

```
/Answer: ((?i)yes)/;  # matches 'Answer: yes', 'Answer: YES', etc.
```

Embedded modifiers can also turn off any modifiers already present by using, e.g., `(?-i)`. Modifiers can also be combined into a single expression, e.g., `(?s-i)` turns on single line mode and turns off case insensitivity.

Embedded modifiers may also be added to a non-capturing grouping. `(?i-m:regexp)` is a non-capturing grouping that matches `regexp` case insensitively and turns off multi-line mode.

68.4.5 Looking ahead and looking behind

This section concerns the lookahead and lookbehind assertions. First, a little background.

In Perl regular expressions, most regexp elements 'eat up' a certain amount of string when they match. For instance, the regexp element `[abc}]` eats up one character of the string when it matches, in the sense that Perl moves to the next character position in the string after the match. There are some elements, however, that don't eat up characters (advance the character position) if they match. The examples we have seen so far are the anchors. The anchor `^` matches the beginning of the line, but doesn't eat any characters. Similarly, the word boundary anchor `\b` matches wherever a character matching `\w` is next to a character that doesn't, but it doesn't eat up any characters itself. Anchors are examples of *zero-width assertions*: zero-width, because they consume no characters, and assertions, because they test some property of the string. In the context of our walk in the woods analogy to regexp matching, most regexp elements move us along a trail, but anchors have us stop a moment and check our surroundings. If the local environment checks out, we can proceed forward. But if the local environment doesn't satisfy us, we must backtrack.

Checking the environment entails either looking ahead on the trail, looking behind, or both. `^` looks behind, to see that there are no characters before. `$` looks ahead, to see that there are no characters after. `\b` looks both ahead and behind, to see if the characters on either side differ in their "word-ness".

The lookahead and lookbehind assertions are generalizations of the anchor concept. Lookahead and lookbehind are zero-width assertions that let us specify which characters we want to test for. The lookahead assertion is denoted by `(?=regexp)` and the lookbehind assertion is denoted by `(?<=fixed-regexp)`. Some examples are

```
$x = "I catch the housecat 'Tom-cat' with catnip";
$x =~ /cat(?=\s)/;   # matches 'cat' in 'housecat'
@catwords = ($x =~ /(?<=\s)cat\w+/g);  # matches,
                                        # $catwords[0] = 'catch'
```

```
                                            # $catwords[1] = ’catnip’
    $x =~ /\bcat\b/;  # matches ’cat’ in ’Tom-cat’
    $x =~ /(?<=\s)cat(?=\s)/; # doesn’t match; no isolated ’cat’ in
                              # middle of $x
```

Note that the parentheses in `(?=regexp)` and `(?<=regexp)` are non-capturing, since these are zero-width assertions. Thus in the second regexp, the substrings captured are those of the whole regexp itself. Lookahead `(?=regexp)` can match arbitrary regexps, but lookbehind `(?<=fixed-regexp)` only works for regexps of fixed width, i.e., a fixed number of characters long. Thus `(?<=(ab|bc))` is fine, but `(?<=(ab)*)` is not. The negated versions of the lookahead and lookbehind assertions are denoted by `(?!regexp)` and `(?<!fixed-regexp)` respectively. They evaluate true if the regexps do *not* match:

```
    $x = "foobar";
    $x =~ /foo(?!bar)/;  # doesn’t match, ’bar’ follows ’foo’
    $x =~ /foo(?!baz)/;  # matches, ’baz’ doesn’t follow ’foo’
    $x =~ /(?<!\s)foo/;  # matches, there is no \s before ’foo’
```

The `\C` is unsupported in lookbehind, because the already treacherous definition of `\C` would become even more so when going backwards.

Here is an example where a string containing blank-separated words, numbers and single dashes is to be split into its components. Using `/\s+/` alone won’t work, because spaces are not required between dashes, or a word or a dash. Additional places for a split are established by looking ahead and behind:

```
    $str = "one two - --6-8";
    @toks = split / \s+              # a run of spaces
                  | (?<=\S) (?=-)    # any non-space followed by ’-’
                  | (?<=-)  (?=\S)   # a ’-’ followed by any non-space
                  /x, $str;          # @toks = qw(one two - - - 6 - 8)
```

68.4.6 Using independent subexpressions to prevent backtracking

Independent subexpressions are regular expressions, in the context of a larger regular expression, that function independently of the larger regular expression. That is, they consume as much or as little of the string as they wish without regard for the ability of the larger regexp to match. Independent subexpressions are represented by `(?>regexp)`. We can illustrate their behavior by first considering an ordinary regexp:

```
    $x = "ab";
    $x =~ /a*ab/;  # matches
```

This obviously matches, but in the process of matching, the subexpression `a*` first grabbed the `a`. Doing so, however, wouldn’t allow the whole regexp to match, so after backtracking, `a*` eventually gave back the `a` and matched the empty string. Here, what `a*` matched was *dependent* on what the rest of the regexp matched.

Contrast that with an independent subexpression:

```
    $x =~ /(?>a*)ab/;  # doesn’t match!
```

The independent subexpression `(?>a*)` doesn’t care about the rest of the regexp, so it sees an `a` and grabs it. Then the rest of the regexp `ab` cannot match. Because `(?>a*)` is independent, there is no backtracking and the independent subexpression does not give up its a. Thus the match of the regexp as a whole fails. A similar behavior occurs with completely independent regexps:

```
$x = "ab";
$x =~ /a*/g;   # matches, eats an 'a'
$x =~ /\Gab/g; # doesn't match, no 'a' available
```

Here //g and \G create a 'tag team' handoff of the string from one regexp to the other. Regexps with an independent subexpression are much like this, with a handoff of the string to the independent subexpression, and a handoff of the string back to the enclosing regexp.

The ability of an independent subexpression to prevent backtracking can be quite useful. Suppose we want to match a non-empty string enclosed in parentheses up to two levels deep. Then the following regexp matches:

```
$x = "abc(de(fg)h";  # unbalanced parentheses
$x =~ /\( ( [^()]+ | \([^()]*\) )+ \)/x;
```

The regexp matches an open parenthesis, one or more copies of an alternation, and a close parenthesis. The alternation is two-way, with the first alternative `[^()]+` matching a substring with no parentheses and the second alternative `\([^()]*\)` matching a substring delimited by parentheses. The problem with this regexp is that it is pathological: it has nested indeterminate quantifiers of the form `(a+|b)+`. We discussed in Part 1 how nested quantifiers like this could take an exponentially long time to execute if there was no match possible. To prevent the exponential blowup, we need to prevent useless backtracking at some point. This can be done by enclosing the inner quantifier as an independent subexpression:

```
$x =~ /\( ( (?>[^()]+) | \([^()]*\) )+ \)/x;
```

Here, `(?>[^()]+)` breaks the degeneracy of string partitioning by gobbling up as much of the string as possible and keeping it. Then match failures fail much more quickly.

68.4.7 Conditional expressions

A *conditional expression* is a form of if-then-else statement that allows one to choose which patterns are to be matched, based on some condition. There are two types of conditional expression: `(?(condition)yes-regexp)` and `(?(condition)yes-regexp|no-regexp)`. `(?(condition)yes-regexp)` is like an `'if () {}'` statement in Perl. If the `condition` is true, the `yes-regexp` will be matched. If the `condition` is false, the `yes-regexp` will be skipped and Perl will move onto the next regexp element. The second form is like an `'if () {} else {}'` statement in Perl. If the `condition` is true, the `yes-regexp` will be matched, otherwise the `no-regexp` will be matched.

The `condition` can have several forms. The first form is simply an integer in parentheses `(integer)`. It is true if the corresponding backreference `\integer` matched earlier in the regexp. The same thing can be done with a name associated with a capture group, written as `(<name>)` or `('name')`. The second form is a bare zero-width assertion `(?...)`, either a lookahead, a lookbehind, or a code assertion (discussed in the next section). The third set of forms provides tests that return true if the expression is executed within a recursion (`(R)`) or is being called from some capturing group, referenced either by number (`(R1)`, `(R2)`,...) or by name (`(R&name)`).

The integer or name form of the condition allows us to choose, with more flexibility, what to match based on what matched earlier in the regexp. This searches for words of the form "$x$x" or "$x$y$y$x":

```
% simple_grep '^(\w+)(\w+)?(?(2)\g2\g1|\g1)$' /usr/dict/words
beriberi
coco
couscous
deed
...
toot
toto
tutu
```

The lookbehind condition allows, along with backreferences, an earlier part of the match to influence a later part of the match. For instance,

```
/[ATGC]+(?(?<=AA)G|C)$/;
```

matches a DNA sequence such that it either ends in AAG, or some other base pair combination and C. Note that the form is (?(?<=AA)G|C) and not (?((?<=AA))G|C); for the lookahead, lookbehind or code assertions, the parentheses around the conditional are not needed.

68.4.8 Defining named patterns

Some regular expressions use identical subpatterns in several places. Starting with Perl 5.10, it is possible to define named subpatterns in a section of the pattern so that they can be called up by name anywhere in the pattern. This syntactic pattern for this definition group is (?(DEFINE)(?<name>pattern)...). An insertion of a named pattern is written as (?&name).

The example below illustrates this feature using the pattern for floating point numbers that was presented earlier on. The three subpatterns that are used more than once are the optional sign, the digit sequence for an integer and the decimal fraction. The DEFINE group at the end of the pattern contains their definition. Notice that the decimal fraction pattern is the first place where we can reuse the integer pattern.

```
/^ (?&osg)\ * ( (?&int)(?&dec)? | (?&dec) )
   (?: [eE](?&osg)(?&int) )?
 $
 (?(DEFINE)
   (?<osg>[-+]?)         # optional sign
   (?<int>\d++)          # integer
   (?<dec>\.(?&int))     # decimal fraction
 )/x
```

68.4.9 Recursive patterns

This feature (introduced in Perl 5.10) significantly extends the power of Perl's pattern matching. By referring to some other capture group anywhere in the pattern with the construct (?group-ref), the *pattern* within the referenced group is used as an independent subpattern in place of the group reference itself. Because the group reference may be contained *within* the group it refers to, it is now possible to apply pattern matching to tasks that hitherto required a recursive parser.

To illustrate this feature, we'll design a pattern that matches if a string contains a palindrome. (This is a word or a sentence that, while ignoring spaces, interpunctuation and case, reads the same backwards as forwards. We begin by observing that the empty string or a string containing just one word character is a palindrome. Otherwise it must have a word character up front and the same at its end, with another palindrome in between.

```
/(?: (\w) (?...Here be a palindrome...) \g{-1} | \w? )/x
```

Adding `\W*` at either end to eliminate what is to be ignored, we already have the full pattern:

```
my $pp = qr/^(\W* (?: (\w) (?1) \g{-1} | \w? ) \W*)$/ix;
for $s ( "saippuakauppias", "A man, a plan, a canal: Panama!" ){
    print "'$s' is a palindrome\n" if $s =~ /$pp/;
}
```

In `(?...)` both absolute and relative backreferences may be used. The entire pattern can be reinserted with `(?R)` or `(?0)`. If you prefer to name your groups, you can use `(?&name)` to recurse into that group.

68.4.10 A bit of magic: executing Perl code in a regular expression

Normally, regexps are a part of Perl expressions. *Code evaluation* expressions turn that around by allowing arbitrary Perl code to be a part of a regexp. A code evaluation expression is denoted `(?{code})`, with *code* a string of Perl statements.

Be warned that this feature is considered experimental, and may be changed without notice.

Code expressions are zero-width assertions, and the value they return depends on their environment. There are two possibilities: either the code expression is used as a conditional in a conditional expression `(?(condition)...)`, or it is not. If the code expression is a conditional, the code is evaluated and the result (i.e., the result of the last statement) is used to determine truth or falsehood. If the code expression is not used as a conditional, the assertion always evaluates true and the result is put into the special variable `$^R`. The variable `$^R` can then be used in code expressions later in the regexp. Here are some silly examples:

```
$x = "abcdef";
$x =~ /abc(?{print "Hi Mom!";})def/; # matches,
                                     # prints 'Hi Mom!'
$x =~ /aaa(?{print "Hi Mom!";})def/; # doesn't match,
                                     # no 'Hi Mom!'
```

Pay careful attention to the next example:

```
$x =~ /abc(?{print "Hi Mom!";})ddd/; # doesn't match,
                                     # no 'Hi Mom!'
                                     # but why not?
```

At first glance, you'd think that it shouldn't print, because obviously the `ddd` isn't going to match the target string. But look at this example:

```
$x =~ /abc(?{print "Hi Mom!";})[dD]dd/; # doesn't match,
                                        # but _does_ print
```

Hmm. What happened here? If you've been following along, you know that the above pattern should be effectively (almost) the same as the last one; enclosing the `d` in a character class isn't going to change what it matches. So why does the first not print while the second one does?

The answer lies in the optimizations the regex engine makes. In the first case, all the engine sees are plain old characters (aside from the `?{}` construct). It's smart enough to realize that the string 'ddd' doesn't occur in our target string before actually running the pattern through. But in the second case, we've tricked it into thinking that our pattern is more complicated. It takes a look, sees our character class, and decides that it will have to actually run the pattern to determine whether or not it matches, and in the process of running it hits the print statement before it discovers that we don't have a match.

To take a closer look at how the engine does optimizations, see the section Section 68.4.12 [Pragmas and debugging], page 1172 below.

More fun with `?{}`:

```
$x =~ /(?{print "Hi Mom!";})/;       # matches,
                                     # prints 'Hi Mom!'
$x =~ /(?{$c = 1;})(?{print "$c";})/;  # matches,
                                       # prints '1'
$x =~ /(?{$c = 1;})(?{print "$^R";})/; # matches,
                                       # prints '1'
```

The bit of magic mentioned in the section title occurs when the regexp backtracks in the process of searching for a match. If the regexp backtracks over a code expression and if the variables used within are localized using `local`, the changes in the variables produced by the code expression are undone! Thus, if we wanted to count how many times a character got matched inside a group, we could use, e.g.,

```
$x = "aaaa";
$count = 0;  # initialize 'a' count
$c = "bob";  # test if $c gets clobbered
$x =~ /(?{local $c = 0;})         # initialize count
       ( a                        # match 'a'
         (?{local $c = $c + 1;})  # increment count
       )*                         # do this any number of times,
       aa                         # but match 'aa' at the end
       (?{$count = $c;})          # copy local $c var into $count
      /x;
print "'a' count is $count, \$c variable is '$c'\n";
```

This prints

```
'a' count is 2, $c variable is 'bob'
```

If we replace the `(?{local $c = $c + 1;})` with `(?{$c = $c + 1;})`, the variable changes are *not* undone during backtracking, and we get

```
'a' count is 4, $c variable is 'bob'
```

Note that only localized variable changes are undone. Other side effects of code expression execution are permanent. Thus

```
$x = "aaaa";
$x =~ /(a(?{print "Yow\n";}))*aa/;
```

produces

```
Yow
Yow
Yow
Yow
```

The result `$^R` is automatically localized, so that it will behave properly in the presence of backtracking.

This example uses a code expression in a conditional to match a definite article, either 'the' in English or 'der|die|das' in German:

```
$lang = 'DE';  # use German
...
$text = "das";
print "matched\n"
    if $text =~ /(?(?{
                      $lang eq 'EN'; # is the language English?
                     })
                   the |             # if so, then match 'the'
                   (der|die|das)     # else, match 'der|die|das'
                 )
                /xi;
```

Note that the syntax here is `(?(?{...})yes-regexp|no-regexp)`, not `(?((?{...}))yes-regexp|no-regexp)`. In other words, in the case of a code expression, we don't need the extra parentheses around the conditional.

If you try to use code expressions where the code text is contained within an interpolated variable, rather than appearing literally in the pattern, Perl may surprise you:

```
$bar = 5;
$pat = '(?{ 1 })';
/foo(?{ $bar })bar/; # compiles ok, $bar not interpolated
/foo(?{ 1 })$bar/;   # compiles ok, $bar interpolated
/foo${pat}bar/;      # compile error!

$pat = qr/(?{ $foo = 1 })/;  # precompile code regexp
/foo${pat}bar/;      # compiles ok
```

If a regexp has a variable that interpolates a code expression, Perl treats the regexp as an error. If the code expression is precompiled into a variable, however, interpolating is ok. The question is, why is this an error?

The reason is that variable interpolation and code expressions together pose a security risk. The combination is dangerous because many programmers who write search engines often take user input and plug it directly into a regexp:

```
$regexp = <>;       # read user-supplied regexp
$chomp $regexp;     # get rid of possible newline
$text =~ /$regexp/; # search $text for the $regexp
```

If the `$regexp` variable contains a code expression, the user could then execute arbitrary Perl code. For instance, some joker could search for `system(’rm -rf *’);` to erase your files. In this sense, the combination of interpolation and code expressions *taints* your regexp. So by default, using both interpolation and code expressions in the same regexp is not allowed. If you’re not concerned about malicious users, it is possible to bypass this security check by invoking `use re ’eval’`:

```
use re ’eval’;       # throw caution out the door
$bar = 5;
$pat = ’(?{ 1 })’;
/foo${pat}bar/;      # compiles ok
```

Another form of code expression is the *pattern code expression*. The pattern code expression is like a regular code expression, except that the result of the code evaluation is treated as a regular expression and matched immediately. A simple example is

```
$length = 5;
$char = ’a’;
$x = ’aaaaabb’;
$x =˜ /(??{$char x $length})/x; # matches, there are 5 of ’a’
```

This final example contains both ordinary and pattern code expressions. It detects whether a binary string `1101010010001...` has a Fibonacci spacing 0,1,1,2,3,5,... of the 1’s:

```
$x = "1101010010001000001";
$z0 = ’’; $z1 = ’0’;   # initial conditions
print "It is a Fibonacci sequence\n"
    if $x =˜ /ˆ1         # match an initial ’1’
                (?:
                   ((??{ $z0 })) # match some ’0’
                   1             # and then a ’1’
                   (?{ $z0 = $z1; $z1 .= $ˆN; })
                )+   # repeat as needed
              $      # that is all there is
             /x;
printf "Largest sequence matched was %d\n", length($z1)-length($z0);
```

Remember that `$ˆN` is set to whatever was matched by the last completed capture group. This prints

```
It is a Fibonacci sequence
Largest sequence matched was 5
```

Ha! Try that with your garden variety regexp package...

Note that the variables `$z0` and `$z1` are not substituted when the regexp is compiled, as happens for ordinary variables outside a code expression. Rather, the whole code block is parsed as perl code at the same time as perl is compiling the code containing the literal regexp pattern.

The regexp without the `//x` modifier is

```
/ˆ1(?:((??{ $z0 }))1(?{ $z0 = $z1; $z1 .= $ˆN; }))+$/
```

which shows that spaces are still possible in the code parts. Nevertheless, when working with code and conditional expressions, the extended form of regexps is almost necessary in creating and debugging regexps.

68.4.11 Backtracking control verbs

Perl 5.10 introduced a number of control verbs intended to provide detailed control over the backtracking process, by directly influencing the regexp engine and by providing monitoring techniques. As all the features in this group are experimental and subject to change or removal in a future version of Perl, the interested reader is referred to Section 58.2.5 [perlre Special Backtracking Control Verbs], page 1017 for a detailed description.

Below is just one example, illustrating the control verb `(*FAIL)`, which may be abbreviated as `(*F)`. If this is inserted in a regexp it will cause it to fail, just as it would at some mismatch between the pattern and the string. Processing of the regexp continues as it would after any "normal" failure, so that, for instance, the next position in the string or another alternative will be tried. As failing to match doesn't preserve capture groups or produce results, it may be necessary to use this in combination with embedded code.

```
%count = ();
"supercalifragilisticexpialidocious" =~
    /([aeiou])(?{ $count{$1}++; })(*FAIL)/i;
printf "%3d '%s'\n", $count{$_}, $_ for (sort keys %count);
```

The pattern begins with a class matching a subset of letters. Whenever this matches, a statement like `$count{'a'}++;` is executed, incrementing the letter's counter. Then `(*FAIL)` does what it says, and the regexp engine proceeds according to the book: as long as the end of the string hasn't been reached, the position is advanced before looking for another vowel. Thus, match or no match makes no difference, and the regexp engine proceeds until the entire string has been inspected. (It's remarkable that an alternative solution using something like

```
$count{lc($_)}++ for split('', "supercalifragilisticexpialidocious");
printf "%3d '%s'\n", $count2{$_}, $_ for ( qw{ a e i o u } );
```

is considerably slower.)

68.4.12 Pragmas and debugging

Speaking of debugging, there are several pragmas available to control and debug regexps in Perl. We have already encountered one pragma in the previous section, `use re 'eval';`, that allows variable interpolation and code expressions to coexist in a regexp. The other pragmas are

```
use re 'taint';
$tainted = <>;
@parts = ($tainted =~ /(\w+)\s+(\w+)/; # @parts is now tainted
```

The `taint` pragma causes any substrings from a match with a tainted variable to be tainted as well. This is not normally the case, as regexps are often used to extract the safe bits from a tainted variable. Use `taint` when you are not extracting safe bits, but are performing some other processing. Both `taint` and `eval` pragmas are lexically scoped, which means they are in effect only until the end of the block enclosing the pragmas.

```
use re '/m';  # or any other flags
```

```
$multiline_string =~ /^foo/; # /m is implied
```

The `re '/flags'` pragma (introduced in Perl 5.14) turns on the given regular expression flags until the end of the lexical scope. See Section "'/flags' mode" in `re` for more detail.

```
use re 'debug';
/^(.*)$/s;       # output debugging info

use re 'debugcolor';
/^(.*)$/s;       # output debugging info in living color
```

The global `debug` and `debugcolor` pragmas allow one to get detailed debugging info about regexp compilation and execution. `debugcolor` is the same as debug, except the debugging information is displayed in color on terminals that can display termcap color sequences. Here is example output:

```
% perl -e 'use re "debug"; "abc" =~ /a*b+c/;'
Compiling REx 'a*b+c'
size 9 first at 1
   1: STAR(4)
   2:   EXACT <a>(0)
   4: PLUS(7)
   5:   EXACT <b>(0)
   7: EXACT <c>(9)
   9: END(0)
floating 'bc' at 0..2147483647 (checking floating) minlen 2
Guessing start of match, REx 'a*b+c' against 'abc'...
Found floating substr 'bc' at offset 1...
Guessed: match at offset 0
Matching REx 'a*b+c' against 'abc'
  Setting an EVAL scope, savestack=3
   0 <> <abc>           |  1:  STAR
                         EXACT <a> can match 1 times out of 32767...
  Setting an EVAL scope, savestack=3
   1 <a> <bc>           |  4:    PLUS
                         EXACT <b> can match 1 times out of 32767...
  Setting an EVAL scope, savestack=3
   2 <ab> <c>           |  7:      EXACT <c>
   3 <abc> <>           |  9:      END
Match successful!
Freeing REx: 'a*b+c'
```

If you have gotten this far into the tutorial, you can probably guess what the different parts of the debugging output tell you. The first part

```
Compiling REx 'a*b+c'
size 9 first at 1
   1: STAR(4)
   2:   EXACT <a>(0)
   4: PLUS(7)
   5:   EXACT <b>(0)
   7: EXACT <c>(9)
   9: END(0)
```

describes the compilation stage. `STAR(4)` means that there is a starred object, in this case `'a'`, and if it matches, goto line 4, i.e., `PLUS(7)`. The middle lines describe some heuristics and optimizations performed before a match:

```
floating 'bc' at 0..2147483647 (checking floating) minlen 2
Guessing start of match, REx 'a*b+c' against 'abc'...
Found floating substr 'bc' at offset 1...
Guessed: match at offset 0
```

Then the match is executed and the remaining lines describe the process:

```
Matching REx 'a*b+c' against 'abc'
  Setting an EVAL scope, savestack=3
   0 <> <abc>           |  1:  STAR
                         EXACT <a> can match 1 times out of 32767...
  Setting an EVAL scope, savestack=3
   1 <a> <bc>           |  4:    PLUS
                         EXACT <b> can match 1 times out of 32767...
  Setting an EVAL scope, savestack=3
   2 <ab> <c>           |  7:      EXACT <c>
   3 <abc> <>           |  9:      END
Match successful!
Freeing REx: 'a*b+c'
```

Each step is of the form `n <x> <y>`, with `<x>` the part of the string matched and `<y>` the part not yet matched. The `| 1: STAR` says that Perl is at line number 1 in the compilation list above. See Section 13.5 [perldebguts Debugging Regular Expressions], page 96 for much more detail.

An alternative method of debugging regexps is to embed `print` statements within the regexp. This provides a blow-by-blow account of the backtracking in an alternation:

```
"that this" =~ m@(?{print "Start at position ", pos, "\n";})
                 t(?{print "t1\n";})
                 h(?{print "h1\n";})
                 i(?{print "i1\n";})
                 s(?{print "s1\n";})
                     |
                 t(?{print "t2\n";})
                 h(?{print "h2\n";})
                 a(?{print "a2\n";})
                 t(?{print "t2\n";})
                 (?{print "Done at position ", pos, "\n";})
                @x;
```

prints

```
Start at position 0
t1
h1
t2
```

```
h2
a2
t2
Done at position 4
```

68.5 BUGS

Code expressions, conditional expressions, and independent expressions are *experimental*. Don't use them in production code. Yet.

68.6 SEE ALSO

This is just a tutorial. For the full story on Perl regular expressions, see the Section 58.1 [perlre NAME], page 989 regular expressions reference page.

For more information on the matching `m//` and substitution `s///` operators, see Section 48.2.30 [perlop Regexp Quote-Like Operators], page 823. For information on the `split` operation, see [perlfunc split], page 453.

For an excellent all-around resource on the care and feeding of regular expressions, see the book *Mastering Regular Expressions* by Jeffrey Friedl (published by O'Reilly, ISBN 1556592-257-3).

68.7 AUTHOR AND COPYRIGHT

68.7.1 Acknowledgments

The inspiration for the stop codon DNA example came from the ZIP code example in chapter 7 of *Mastering Regular Expressions*.

The author would like to thank Jeff Pinyan, Andrew Johnson, Peter Haworth, Ronald J Kimball, and Joe Smith for all their helpful comments.

69 perlrun

69.1 NAME

perlrun - how to execute the Perl interpreter

69.2 SYNOPSIS

perl [**-sTtuUWX**] [**-hv**] [**-V**[:*configvar*]] [**-cw**] [**-d**[**t**][:*debugger*]] [**-D**[*number/list*]]
[**-pna**] [**-F***pattern*] [**-l**[*octal*]] [**-0**[*octal/hexadecimal*]] [**-I***dir*] [**-m**[**-**]*module*] [**-M**[**-**]*'module...'*] [**-f**]
[**-C** [*number/list*]] [**-S**] [**-x**[*dir*]] [**-i**[*extension*]] [[**-e**|**-E**] *'command'*] [–] [*programfile*] [*argument*]...

69.3 DESCRIPTION

The normal way to run a Perl program is by making it directly executable, or else by passing the name of the source file as an argument on the command line. (An interactive Perl environment is also possible–see Section 15.1 [perldebug NAME], page 120 for details on how to do that.) Upon startup, Perl looks for your program in one of the following places:

1. Specified line by line via **-e** or **-E** switches on the command line.
2. Contained in the file specified by the first filename on the command line. (Note that systems supporting the `#!` notation invoke interpreters this way. See Section 69.3.2 [Location of Perl], page 1178.)
3. Passed in implicitly via standard input. This works only if there are no filename arguments–to pass arguments to a STDIN-read program you must explicitly specify a `"-"` for the program name.

With methods 2 and 3, Perl starts parsing the input file from the beginning, unless you've specified a **-x** switch, in which case it scans for the first line starting with `#!` and containing the word "perl", and starts there instead. This is useful for running a program embedded in a larger message. (In this case you would indicate the end of the program using the `__END__` token.)

The `#!` line is always examined for switches as the line is being parsed. Thus, if you're on a machine that allows only one argument with the `#!` line, or worse, doesn't even recognize the `#!` line, you still can get consistent switch behaviour regardless of how Perl was invoked, even if **-x** was used to find the beginning of the program.

Because historically some operating systems silently chopped off kernel interpretation of the `#!` line after 32 characters, some switches may be passed in on the command line, and some may not; you could even get a `"-"` without its letter, if you're not careful. You probably want to make sure that all your switches fall either before or after that 32-character boundary. Most switches don't actually care if they're processed redundantly, but getting a `"-"` instead of a complete switch could cause Perl to try to execute standard input instead of your program. And a partial **-I** switch could also cause odd results.

Some switches do care if they are processed twice, for instance combinations of **-l** and **-0**. Either put all the switches after the 32-character boundary (if applicable), or replace the use of **-0***digits* by `BEGIN{ $/ = "\0digits"; }`.

Parsing of the `#!` switches starts wherever "perl" is mentioned in the line. The sequences "-*" and "- " are specifically ignored so that you could, if you were so inclined, say

```
#!/bin/sh
#! -*-perl-*-
eval 'exec perl -x -wS $0 ${1+"$@"}'
    if 0;
```

to let Perl see the **-p** switch.

A similar trick involves the *env* program, if you have it.

```
#!/usr/bin/env perl
```

The examples above use a relative path to the perl interpreter, getting whatever version is first in the user's path. If you want a specific version of Perl, say, perl5.14.1, you should place that directly in the `#!` line's path.

If the `#!` line does not contain the word "perl" nor the word "indir" the program named after the `#!` is executed instead of the Perl interpreter. This is slightly bizarre, but it helps people on machines that don't do `#!`, because they can tell a program that their SHELL is `/usr/bin/perl`, and Perl will then dispatch the program to the correct interpreter for them.

After locating your program, Perl compiles the entire program to an internal form. If there are any compilation errors, execution of the program is not attempted. (This is unlike the typical shell script, which might run part-way through before finding a syntax error.)

If the program is syntactically correct, it is executed. If the program runs off the end without hitting an exit() or die() operator, an implicit `exit(0)` is provided to indicate successful completion.

69.3.1 #! and quoting on non-Unix systems

Unix's `#!` technique can be simulated on other systems:

OS/2

Put

```
extproc perl -S -your_switches
```

as the first line in `*.cmd` file (**-S** due to a bug in cmd.exe's 'extproc' handling).

MS-DOS

Create a batch file to run your program, and codify it in `ALTERNATE_SHEBANG` (see the `dosish.h` file in the source distribution for more information).

Win95/NT

The Win95/NT installation, when using the ActiveState installer for Perl, will modify the Registry to associate the `.pl` extension with the perl interpreter. If you install Perl by other means (including building from the sources), you may have to modify the Registry yourself. Note that this means you can no longer tell the difference between an executable Perl program and a Perl library file.

VMS

Put

```
$ perl -mysw 'f$env("procedure")' 'p1' 'p2' 'p3' 'p4' 'p5' 'p6' 'p7' 'p8' !
$ exit++ + ++$status != 0 and $exit = $status = undef;
```

at the top of your program, where **-mysw** are any command line switches you want to pass to Perl. You can now invoke the program directly, by saying `perl program`, or as a DCL procedure, by saying `@program` (or implicitly via `DCL$PATH` by just using the name of the program).

This incantation is a bit much to remember, but Perl will display it for you if you say `perl "-V:startperl"`.

Command-interpreters on non-Unix systems have rather different ideas on quoting than Unix shells. You'll need to learn the special characters in your command-interpreter (`*`, `\` and `"` are common) and how to protect whitespace and these characters to run one-liners (see [-e], page 1182 below).

On some systems, you may have to change single-quotes to double ones, which you must *not* do on Unix or Plan 9 systems. You might also have to change a single % to a %%.

For example:

```
# Unix
perl -e 'print "Hello world\n"'

# MS-DOS, etc.
perl -e "print \"Hello world\n\""

# VMS
perl -e "print ""Hello world\n"""
```

The problem is that none of this is reliable: it depends on the command and it is entirely possible neither works. If *4DOS* were the command shell, this would probably work better:

```
perl -e "print <Ctrl-x>"Hello world\n<Ctrl-x>""
```

CMD.EXE in Windows NT slipped a lot of standard Unix functionality in when nobody was looking, but just try to find documentation for its quoting rules.

There is no general solution to all of this. It's just a mess.

69.3.2 Location of Perl

It may seem obvious to say, but Perl is useful only when users can easily find it. When possible, it's good for both `/usr/bin/perl` and `/usr/local/bin/perl` to be symlinks to the actual binary. If that can't be done, system administrators are strongly encouraged to put (symlinks to) perl and its accompanying utilities into a directory typically found along a user's PATH, or in some other obvious and convenient place.

In this documentation, `#!/usr/bin/perl` on the first line of the program will stand in for whatever method works on your system. You are advised to use a specific path if you care about a specific version.

```
#!/usr/local/bin/perl5.14
```

or if you just want to be running at least version, place a statement like this at the top of your program:

```
use 5.014;
```

69.3.3 Command Switches

As with all standard commands, a single-character switch may be clustered with the following switch, if any.

```
#!/usr/bin/perl -spi.orig   # same as -s -p -i.orig
```

A `--` signals the end of options and disables further option processing. Any arguments after the `--` are treated as filenames and arguments.

Switches include:

-0[*octal/hexadecimal*]

specifies the input record separator (`$/`) as an octal or hexadecimal number. If there are no digits, the null character is the separator. Other switches may precede or follow the digits. For example, if you have a version of *find* which can print filenames terminated by the null character, you can say this:

```
find . -name '*.orig' -print0 | perl -n0e unlink
```

The special value 00 will cause Perl to slurp files in paragraph mode. Any value 0400 or above will cause Perl to slurp files whole, but by convention the value 0777 is the one normally used for this purpose.

You can also specify the separator character using hexadecimal notation: **-0x***HHH...*, where the *H* are valid hexadecimal digits. Unlike the octal form, this one may be used to specify any Unicode character, even those beyond 0xFF. So if you *really* want a record separator of 0777, specify it as **-0x1FF**. (This means that you cannot use the **-x** option with a directory name that consists of hexadecimal digits, or else Perl will think you have specified a hex number to **-0**.)

-a

turns on autosplit mode when used with a **-n** or **-p**. An implicit split command to the @F array is done as the first thing inside the implicit while loop produced by the **-n** or **-p**.

```
perl -ane 'print pop(@F), "\n";'
```

is equivalent to

```
while (<>) {
    @F = split(' ');
    print pop(@F), "\n";
}
```

An alternate delimiter may be specified using **-F**.

-a implicitly sets **-n**.

-C [*number/list*]

The **-C** flag controls some of the Perl Unicode features.

As of 5.8.1, the **-C** can be followed either by a number or a list of option letters. The letters, their numeric values, and effects are as follows; listing the letters is equal to summing the numbers.

```
I     1   STDIN is assumed to be in UTF-8
O     2   STDOUT will be in UTF-8
```

```
E     4   STDERR will be in UTF-8
S     7   I + O + E
i     8   UTF-8 is the default PerlIO layer for input streams
o    16   UTF-8 is the default PerlIO layer for output streams
D    24   i + o
A    32   the @ARGV elements are expected to be strings encoded
          in UTF-8
L    64   normally the "IOEioA" are unconditional, the L makes
          them conditional on the locale environment variables
          (the LC_ALL, LC_CTYPE, and LANG, in the order of
          decreasing precedence) -- if the variables indicate
          UTF-8, then the selected "IOEioA" are in effect
a   256   Set ${^UTF8CACHE} to -1, to run the UTF-8 caching
          code in debugging mode.
```

For example, **-COE** and **-C6** will both turn on UTF-8-ness on both STDOUT and STDERR. Repeating letters is just redundant, not cumulative nor toggling.

The `io` options mean that any subsequent open() (or similar I/O operations) in the current file scope will have the `:utf8` PerlIO layer implicitly applied to them, in other words, UTF-8 is expected from any input stream, and UTF-8 is produced to any output stream. This is just the default, with explicit layers in open() and with binmode() one can manipulate streams as usual.

-C on its own (not followed by any number or option list), or the empty string `""` for the `PERL_UNICODE` environment variable, has the same effect as **-CSDL**. In other words, the standard I/O handles and the default `open()` layer are UTF-8-fied *but* only if the locale environment variables indicate a UTF-8 locale. This behaviour follows the *implicit* (and problematic) UTF-8 behaviour of Perl 5.8.0. (See Section “UTF-8 no longer default under UTF-8 locales” in `perl581delta`.)

You can use **-C0** (or `"0"` for `PERL_UNICODE`) to explicitly disable all the above Unicode features.

The read-only magic variable `${^UNICODE}` reflects the numeric value of this setting. This variable is set during Perl startup and is thereafter read-only. If you want runtime effects, use the three-arg open() (see ⟨undefined⟩ [perlfunc open], page ⟨undefined⟩), the two-arg binmode() (see ⟨undefined⟩ [perlfunc binmode], page ⟨undefined⟩), and the `open` pragma (see `open`).

(In Perls earlier than 5.8.1 the **-C** switch was a Win32-only switch that enabled the use of Unicode-aware "wide system call" Win32 APIs. This feature was practically unused, however, and the command line switch was therefore "recycled".)

Note: Since perl 5.10.1, if the **-C** option is used on the `#!` line, it must be specified on the command line as well, since the standard streams are already set up at this point in the execution of the perl interpreter. You can also use binmode() to set the encoding of an I/O stream.

-c

causes Perl to check the syntax of the program and then exit without executing it. Actually, it *will* execute and `BEGIN`, `UNITCHECK`, or `CHECK` blocks and any `use` statements: these are considered as occurring outside the execution of your program. `INIT` and `END` blocks, however, will be skipped.

-d

-dt

runs the program under the Perl debugger. See Section 15.1 [perldebug NAME], page 120. If **t** is specified, it indicates to the debugger that threads will be used in the code being debugged.

-d:*MOD[=bar,baz]*
-dt:*MOD[=bar,baz]*

runs the program under the control of a debugging, profiling, or tracing module installed as `Devel::`*`MOD`*. E.g., **-d:DProf** executes the program using the `Devel::DProf` profiler. As with the **-M** flag, options may be passed to the `Devel::`*`MOD`* package where they will be received and interpreted by the `Devel::`*`MOD`*`::import` routine. Again, like **-M**, use **--d:-***MOD* to call `Devel::`*`MOD`*`::unimport` instead of import. The comma-separated list of options must follow a `=` character. If **t** is specified, it indicates to the debugger that threads will be used in the code being debugged. See Section 15.1 [perldebug NAME], page 120.

-D*letters*

-D*number*

sets debugging flags. To watch how it executes your program, use **-Dtls**. (This works only if debugging is compiled into your Perl.) Another nice value is **-Dx**, which lists your compiled syntax tree. And **-Dr** displays compiled regular expressions; the format of the output is explained in Section 13.1 [perldebguts NAME], page 90.

As an alternative, specify a number instead of list of letters (e.g., **-D14** is equivalent to **-Dtls**):

```
        1  p  Tokenizing and parsing (with v, displays parse stack)
        2  s  Stack snapshots (with v, displays all stacks)
        4  l  Context (loop) stack processing
        8  t  Trace execution
       16  o  Method and overloading resolution
       32  c  String/numeric conversions
       64  P  Print profiling info, source file input state
      128  m  Memory and SV allocation
      256  f  Format processing
      512  r  Regular expression parsing and execution
     1024  x  Syntax tree dump
     2048  u  Tainting checks
     4096  U  Unofficial, User hacking (reserved for private,
              unreleased use)
     8192  H  Hash dump -- usurps values()
    16384  X  Scratchpad allocation
    32768  D  Cleaning up
```

65536	S	Op slab allocation
131072	T	Tokenizing
262144	R	Include reference counts of dumped variables (eg when using -Ds)
524288	J	show s,t,P-debug (don't Jump over) on opcodes within package DB
1048576	v	Verbose: use in conjunction with other flags
2097152	C	Copy On Write
4194304	A	Consistency checks on internal structures
8388608	q	quiet - currently only suppresses the "EXECUTING" message
16777216	M	trace smart match resolution
33554432	B	dump suBroutine definitions, including special Blocks like BEGIN
67108864	L	trace Locale-related info; what gets output is very subject to change

All these flags require **-DDEBUGGING** when you compile the Perl executable (but see `:opd` in `Devel-Peek` or Section "'debug' mode" in `re` which may change this). See the `INSTALL` file in the Perl source distribution for how to do this. This flag is automatically set if you include **-g** option when `Configure` asks you about optimizer/debugger flags.

If you're just trying to get a print out of each line of Perl code as it executes, the way that `sh -x` provides for shell scripts, you can't use Perl's **-D** switch. Instead do this

```
# If you have "env" utility
env PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

# Bourne shell syntax
$ PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

# csh syntax
% (setenv PERLDB_OPTS "NonStop=1 AutoTrace=1 frame=2"; perl -dS program)
```

See Section 15.1 [perldebug NAME], page 120 for details and variations.

-e *commandline*

may be used to enter one line of program. If **-e** is given, Perl will not look for a filename in the argument list. Multiple **-e** commands may be given to build up a multi-line script. Make sure to use semicolons where you would in a normal program.

-E *commandline*

behaves just like **-e**, except that it implicitly enables all optional features (in the main compilation unit). See `feature`.

-f

Disable executing `$Config{sitelib}/sitecustomize.pl` at startup.

Perl can be built so that it by default will try to execute `$Config{sitelib}/sitecustomize.pl` at startup (in a BEGIN block). This is a hook that allows the sysadmin to customize how Perl behaves. It can for instance be used to add entries to the @INC array to make Perl find modules in non-standard locations.

Perl actually inserts the following code:

```
BEGIN {
    do { local $!; -f "$Config{sitelib}/sitecustomize.pl"; }
        && do "$Config{sitelib}/sitecustomize.pl";
}
```

Since it is an actual `do` (not a `require`), `sitecustomize.pl` doesn't need to return a true value. The code is run in package `main`, in its own lexical scope. However, if the script dies, `$@` will not be set.

The value of `$Config{sitelib}` is also determined in C code and not read from `Config.pm`, which is not loaded.

The code is executed *very* early. For example, any changes made to `@INC` will show up in the output of 'perl -V'. Of course, `END` blocks will be likewise executed very late.

To determine at runtime if this capability has been compiled in your perl, you can check the value of `$Config{usesitecustomize}`.

-F*pattern*

specifies the pattern to split on for **-a**. The pattern may be surrounded by `//`, `""`, or `''`, otherwise it will be put in single quotes. You can't use literal whitespace in the pattern.

-F implicitly sets both **-a** and **-n**.

-h

prints a summary of the options.

-i[*extension*]

specifies that files processed by the `<>` construct are to be edited in-place. It does this by renaming the input file, opening the output file by the original name, and selecting that output file as the default for print() statements. The extension, if supplied, is used to modify the name of the old file to make a backup copy, following these rules:

If no extension is supplied, and your system supports it, the original *file* is kept open without a name while the output is redirected to a new file with the original *filename*. When perl exits, cleanly or not, the original *file* is unlinked.

If the extension doesn't contain a `*`, then it is appended to the end of the current filename as a suffix. If the extension does contain one or more `*` characters, then each `*` is replaced with the current filename. In Perl terms, you could think of this as:

```
($backup = $extension) =~ s/\*/$file_name/g;
```

This allows you to add a prefix to the backup file, instead of (or in addition to) a suffix:

```
 $ perl -pi'orig_*' -e 's/bar/baz/' fileA  # backup to
```

```
                                                    # 'orig_fileA'
```

Or even to place backup copies of the original files into another directory (provided the directory already exists):

```
 $ perl -pi'old/*.orig' -e 's/bar/baz/' fileA  # backup to
                                                # 'old/fileA.orig'
```

These sets of one-liners are equivalent:

```
 $ perl -pi -e 's/bar/baz/' fileA          # overwrite current file
 $ perl -pi'*' -e 's/bar/baz/' fileA       # overwrite current file

 $ perl -pi'.orig' -e 's/bar/baz/' fileA   # backup to 'fileA.orig'
 $ perl -pi'*.orig' -e 's/bar/baz/' fileA  # backup to 'fileA.orig'
```

From the shell, saying

```
    $ perl -p -i.orig -e "s/foo/bar/; ... "
```

is the same as using the program:

```
    #!/usr/bin/perl -pi.orig
    s/foo/bar/;
```

which is equivalent to

```
    #!/usr/bin/perl
    $extension = '.orig';
    LINE: while (<>) {
        if ($ARGV ne $oldargv) {
            if ($extension !~ /\*/) {
                $backup = $ARGV . $extension;
            }
            else {
                ($backup = $extension) =~ s/\*/$ARGV/g;
            }
            rename($ARGV, $backup);
            open(ARGVOUT, ">$ARGV");
            select(ARGVOUT);
            $oldargv = $ARGV;
        }
        s/foo/bar/;
    }
    continue {
        print;  # this prints to original filename
    }
    select(STDOUT);
```

except that the **-i** form doesn't need to compare $ARGV to $oldargv to know when the filename has changed. It does, however, use ARGVOUT for the selected filehandle. Note that STDOUT is restored as the default output filehandle after the loop.

As shown above, Perl creates the backup file whether or not any output is actually changed. So this is just a fancy way to copy files:

```
    $ perl -p -i'/some/file/path/*' -e 1 file1 file2 file3...
```

or

```
    $ perl -p -i'.orig' -e 1 file1 file2 file3...
```

You can use `eof` without parentheses to locate the end of each input file, in case you want to append to each file, or reset line numbering (see example in [perlfunc eof], page 376).

If, for a given file, Perl is unable to create the backup file as specified in the extension then it will skip that file and continue on with the next one (if it exists).

For a discussion of issues surrounding file permissions and **-i**, see Section "Why does Perl let me delete read-only files? Why does -i clobber protected files? Isn't this a bug in Perl?" in `perlfaq5`.

You cannot use **-i** to create directories or to strip extensions from files.

Perl does not expand ~ in filenames, which is good, since some folks use it for their backup files:

```
    $ perl -pi~ -e 's/foo/bar/' file1 file2 file3...
```

Note that because **-i** renames or deletes the original file before creating a new file of the same name, Unix-style soft and hard links will not be preserved.

Finally, the **-i** switch does not impede execution when no files are given on the command line. In this case, no backup is made (the original file cannot, of course, be determined) and processing proceeds from STDIN to STDOUT as might be expected.

-I*directory*

Directories specified by **-I** are prepended to the search path for modules (`@INC`).

-l[*octnum*]

enables automatic line-ending processing. It has two separate effects. First, it automatically chomps `$/` (the input record separator) when used with **-n** or **-p**. Second, it assigns `$\` (the output record separator) to have the value of *octnum* so that any print statements will have that separator added back on. If *octnum* is omitted, sets `$\` to the current value of `$/`. For instance, to trim lines to 80 columns:

```
    perl -lpe 'substr($_, 80) = ""'
```

Note that the assignment `$\ = $/` is done when the switch is processed, so the input record separator can be different than the output record separator if the **-l** switch is followed by a **-0** switch:

```
    gnufind / -print0 | perl -ln0e 'print "found $_" if -p'
```

This sets `$\` to newline and then sets `$/` to the null character.

-m[**-**]*module*
-M[**-**]*module*
-M[**-**]*'module ...'*
-[mM][**-**]*module=arg[,arg]...*

-m*module* executes `use` *module* `();` before executing your program.

-M*module* executes `use` *module* `;` before executing your program. You can use quotes to add extra code after the module name, e.g., `'-M`*`MODULE`*` qw(foo bar)'`.

If the first character after the **-M** or **-m** is a dash (**-**) then the 'use' is replaced with 'no'.

A little builtin syntactic sugar means you can also say **-m***MODULE*=**foo,bar** or **-M***MODULE*=**foo,bar** as a shortcut for **'-M***MODULE* **qw(foo bar)'**. This avoids the need to use quotes when importing symbols. The actual code generated by **-M***MODULE*=**foo,bar** is `use module split(/,/,q{foo,bar})`. Note that the `=` form removes the distinction between **-m** and **-M**; that is, **-m***MODULE*=**foo,bar** is the same as **-M***MODULE*=**foo,bar**.

A consequence of this is that **-M***MODULE*=**number** never does a version check, unless *`MODULE`*`::import()` itself is set up to do a version check, which could happen for example if *MODULE* inherits from `Exporter`.

-n

causes Perl to assume the following loop around your program, which makes it iterate over filename arguments somewhat like *sed -n* or *awk*:

```
LINE:
  while (<>) {
      ...             # your program goes here
  }
```

Note that the lines are not printed by default. See [-p], page 1186 to have lines printed. If a file named by an argument cannot be opened for some reason, Perl warns you about it and moves on to the next file.

Also note that `<>` passes command line arguments to ⟨undefined⟩ [perlfunc open], page ⟨undefined⟩, which doesn't necessarily interpret them as file names. See Section 48.1 [perlop NAME], page 798 for possible security implications.

Here is an efficient way to delete all files that haven't been modified for at least a week:

```
    find . -mtime +7 -print | perl -nle unlink
```

This is faster than using the **-exec** switch of *find* because you don't have to start a process on every filename found. It does suffer from the bug of mishandling newlines in pathnames, which you can fix if you follow the example under **-0**.

`BEGIN` and `END` blocks may be used to capture control before or after the implicit program loop, just as in *awk*.

-p

causes Perl to assume the following loop around your program, which makes it iterate over filename arguments somewhat like *sed*:

```
LINE:
  while (<>) {
      ...             # your program goes here
  } continue {
      print or die "-p destination: $!\n";
```

```
    }
```

If a file named by an argument cannot be opened for some reason, Perl warns you about it, and moves on to the next file. Note that the lines are printed automatically. An error occurring during printing is treated as fatal. To suppress printing use the **-n** switch. A **-p** overrides a **-n** switch.

`BEGIN` and `END` blocks may be used to capture control before or after the implicit loop, just as in *awk*.

-s

enables rudimentary switch parsing for switches on the command line after the program name but before any filename arguments (or before an argument of –). Any switch found there is removed from @ARGV and sets the corresponding variable in the Perl program. The following program prints "1" if the program is invoked with a **-xyz** switch, and "abc" if it is invoked with **-xyz=abc**.

```
    #!/usr/bin/perl -s
    if ($xyz) { print "$xyz\n" }
```

Do note that a switch like **–help** creates the variable `${-help}`, which is not compliant with `use strict "refs"`. Also, when using this option on a script with warnings enabled you may get a lot of spurious "used only once" warnings.

-S

makes Perl use the PATH environment variable to search for the program unless the name of the program contains path separators.

On some platforms, this also makes Perl append suffixes to the filename while searching for it. For example, on Win32 platforms, the ".bat" and ".cmd" suffixes are appended if a lookup for the original name fails, and if the name does not already end in one of those suffixes. If your Perl was compiled with `DEBUGGING` turned on, using the **-Dp** switch to Perl shows how the search progresses.

Typically this is used to emulate `#!` startup on platforms that don't support `#!`. It's also convenient when debugging a script that uses `#!`, and is thus normally found by the shell's $PATH search mechanism.

This example works on many platforms that have a shell compatible with Bourne shell:

```
    #!/usr/bin/perl
    eval 'exec /usr/bin/perl -wS $0 ${1+"$@"}'
            if $running_under_some_shell;
```

The system ignores the first line and feeds the program to `/bin/sh`, which proceeds to try to execute the Perl program as a shell script. The shell executes the second line as a normal shell command, and thus starts up the Perl interpreter. On some systems $0 doesn't always contain the full pathname, so the **-S** tells Perl to search for the program if necessary. After Perl locates the program, it parses the lines and ignores them because the variable $running_under_some_shell is never true. If the program will be interpreted by csh, you will need to replace `${1+"$@"}` with `$*`, even though that doesn't understand embedded spaces (and such) in the argument list. To start up *sh* rather than *csh*, some systems may have to replace the `#!` line with a line containing just a colon, which will be politely ignored by Perl. Other systems can't control that, and need a totally devious construct that will work under any of *csh*, *sh*, or Perl, such as the following:

```
eval '(exit $?0)' && eval 'exec perl -wS $0 ${1+"$@"}'
& eval 'exec /usr/bin/perl -wS $0 $argv:q'
        if $running_under_some_shell;
```

If the filename supplied contains directory separators (and so is an absolute or relative pathname), and if that file is not found, platforms that append file extensions will do so and try to look for the file with those extensions added, one by one.

On DOS-like platforms, if the program does not contain directory separators, it will first be searched for in the current directory before being searched for on the PATH. On Unix platforms, the program will be searched for strictly on the PATH.

-t

Like **-T**, but taint checks will issue warnings rather than fatal errors. These warnings can now be controlled normally with `no warnings qw(taint)`.

Note: This is not a substitute for `-T`! This is meant to be used *only* as a temporary development aid while securing legacy code: for real production code and for new secure code written from scratch, always use the real **-T**.

-T

turns on "taint" so you can test them. Ordinarily these checks are done only when running setuid or setgid. It's a good idea to turn them on explicitly for programs that run on behalf of someone else whom you might not necessarily trust, such as CGI programs or any internet servers you might write in Perl. See Section 70.1 [perlsec NAME], page 1198 for details. For security reasons, this option must be seen by Perl quite early; usually this means it must appear early on the command line or in the `#!` line for systems which support that construct.

-u

This switch causes Perl to dump core after compiling your program. You can then in theory take this core dump and turn it into an executable file by using the *undump* program (not supplied). This speeds startup at the expense of some disk space (which you can minimize by stripping the executable). (Still, a "hello world" executable comes out to about 200K on my machine.) If you want to execute a portion of your program before dumping, use the dump() operator instead. Note: availability of *undump* is platform specific and may not be available for a specific port of Perl.

-U

allows Perl to do unsafe operations. Currently the only "unsafe" operations are attempting to unlink directories while running as superuser and running setuid programs with fatal taint checks turned into warnings. Note that warnings must be enabled along with this option to actually *generate* the taint-check warnings.

-v

prints the version and patchlevel of your perl executable.

-V

prints summary of the major perl configuration values and the current values of @INC.

-V:*configvar*

Prints to STDOUT the value of the named configuration variable(s), with multiples when your *configvar* argument looks like a regex (has non-letters). For example:

```
$ perl -V:libc
    libc=’/lib/libc-2.2.4.so’;
$ perl -V:lib.
    libs=’-lnsl -lgdbm -ldb -ldl -lm -lcrypt -lutil -lc’;
    libc=’/lib/libc-2.2.4.so’;
$ perl -V:lib.*
    libpth=’/usr/local/lib /lib /usr/lib’;
    libs=’-lnsl -lgdbm -ldb -ldl -lm -lcrypt -lutil -lc’;
    lib_ext=’.a’;
    libc=’/lib/libc-2.2.4.so’;
    libperl=’libperl.a’;
    ....
```

Additionally, extra colons can be used to control formatting. A trailing colon suppresses the linefeed and terminator ";", allowing you to embed queries into shell commands. (mnemonic: PATH separator ":".)

```
$ echo "compression-vars: " ‘perl -V:z.*: ‘ " are here !"
compression-vars:  zcat=’’ zip=’zip’  are here !
```

A leading colon removes the "name=" part of the response, this allows you to map to the name you need. (mnemonic: empty label)

```
$ echo "goodvfork="‘./perl -Ilib -V::usevfork‘
goodvfork=false;
```

Leading and trailing colons can be used together if you need positional parameter values without the names. Note that in the case below, the `PERL_API` params are returned in alphabetical order.

```
$ echo building_on ‘perl -V::osname: -V::PERL_API_.*:‘ now
building_on ’linux’ ’5’ ’1’ ’9’ now
```

-w

prints warnings about dubious constructs, such as variable names mentioned only once and scalar variables used before being set; redefined subroutines; references to undefined filehandles; filehandles opened read-only that you are attempting to write on; values used as a number that don’t *look* like numbers;

using an array as though it were a scalar; if your subroutines recurse more than 100 deep; and innumerable other things.

This switch really just enables the global `$^W` variable; normally, the lexically scoped `use warnings` pragma is preferred. You can disable or promote into fatal errors specific warnings using `__WARN__` hooks, as described in Section 86.1 [perlvar NAME], page 1375 and ⟨undefined⟩ [perlfunc warn], page ⟨undefined⟩. See also Section 16.1 [perldiag NAME], page 137 and Section 80.1 [perltrap NAME], page 1312. A fine-grained warning facility is also available if you want to manipulate entire classes of warnings; see `warnings`.

-W

Enables all warnings regardless of `no warnings` or `$^W`. See `warnings`.

-X

Disables all warnings regardless of `use warnings` or `$^W`. See `warnings`.

-x

-x*directory*

tells Perl that the program is embedded in a larger chunk of unrelated text, such as in a mail message. Leading garbage will be discarded until the first line that starts with `#!` and contains the string `"perl"`. Any meaningful switches on that line will be applied.

All references to line numbers by the program (warnings, errors, ...) will treat the `#!` line as the first line. Thus a warning on the 2nd line of the program, which is on the 100th line in the file will be reported as line 2, not as line 100. This can be overridden by using the `#line` directive. (See Section 74.2.15 [perlsyn Plain Old Comments (Not!)], page 1262)

If a directory name is specified, Perl will switch to that directory before running the program. The **-x** switch controls only the disposal of leading garbage. The program must be terminated with `__END__` if there is trailing garbage to be ignored; the program can process any or all of the trailing garbage via the `DATA` filehandle if desired.

The directory, if specified, must appear immediately following the **-x** with no intervening whitespace.

69.4 ENVIRONMENT

HOME

Used if `chdir` has no argument.

LOGDIR

Used if `chdir` has no argument and HOME is not set.

PATH

Used in executing subprocesses, and in finding the program if **-S** is used.

PERL5LIB

A list of directories in which to look for Perl library files before looking in the standard library and the current directory. Any architecture-specific and version-specific directories, such as `version/archname/`, `version/`, or `archname/` under the specified locations are automatically included if they exist, with this lookup done at interpreter startup time. In addition, any directories matching the entries in `$Config{inc_version_list}` are added. (These typically would be for older compatible perl versions installed in the same directory tree.)

If PERL5LIB is not defined, PERLLIB is used. Directories are separated (like in PATH) by a colon on Unixish platforms and by a semicolon on Windows (the proper path separator being given by the command `perl -V:`*`path_sep`*).

When running taint checks, either because the program was running setuid or setgid, or the **-T** or **-t** switch was specified, neither PERL5LIB nor PERLLIB is consulted. The program should instead say:

```
use lib "/my/directory";
```

PERL5OPT

Command-line options (switches). Switches in this variable are treated as if they were on every Perl command line. Only the **-[CDIMUdmtwW]** switches are allowed. When running taint checks (either because the program was running setuid or setgid, or because the **-T** or **-t** switch was used), this variable is ignored. If PERL5OPT begins with **-T**, tainting will be enabled and subsequent options ignored. If PERL5OPT begins with **-t**, tainting will be enabled, a writable dot removed from @INC, and subsequent options honored.

PERLIO

A space (or colon) separated list of PerlIO layers. If perl is built to use PerlIO system for IO (the default) these layers affect Perl's IO.

It is conventional to start layer names with a colon (for example, `:perlio`) to emphasize their similarity to variable "attributes". But the code that parses layer specification strings, which is also used to decode the PERLIO environment variable, treats the colon as a separator.

An unset or empty PERLIO is equivalent to the default set of layers for your platform; for example, `:unix:perlio` on Unix-like systems and `:unix:crlf` on Windows and other DOS-like systems.

The list becomes the default for *all* Perl's IO. Consequently only built-in layers can appear in this list, as external layers (such as `:encoding()`) need IO in order to load them! See `open` for how to add external encodings as defaults.

Layers it makes sense to include in the PERLIO environment variable are briefly summarized below. For more details see `PerlIO`.

:bytes

A pseudolayer that turns the `:utf8` flag *off* for the layer below; unlikely to be useful on its own in the global PERLIO environment variable. You perhaps were thinking of `:crlf:bytes` or `:perlio:bytes`.

:crlf

A layer which does CRLF to `"\n"` translation distinguishing `"text"` and `"binary"` files in the manner of MS-DOS and similar operating systems. (It currently does *not* mimic MS-DOS as far as treating of Control-Z as being an end-of-file marker.)

:mmap

A layer that implements "reading" of files by using *mmap*(2) to make an entire file appear in the process's address space, and then using that as PerlIO's "buffer".

:perlio

This is a re-implementation of stdio-like buffering written as a PerlIO layer. As such it will call whatever layer is below it for its operations, typically `:unix`.

:pop

An experimental pseudolayer that removes the topmost layer. Use with the same care as is reserved for nitroglycerine.

:raw

A pseudolayer that manipulates other layers. Applying the `:raw` layer is equivalent to calling `binmode($fh)`. It makes the stream pass each byte as-is without translation. In particular, both CRLF translation and intuiting `:utf8` from the locale are disabled.

Unlike in earlier versions of Perl, `:raw` is *not* just the inverse of `:crlf`: other layers which would affect the binary nature of the stream are also removed or disabled.

:stdio

This layer provides a PerlIO interface by wrapping system's ANSI C "stdio" library calls. The layer provides both buffering and IO. Note that the `:stdio` layer does *not* do CRLF translation even if that is the platform's normal behaviour. You will need a `:crlf` layer above it to do that.

:unix

Low-level layer that calls `read`, `write`, `lseek`, etc.

:utf8

A pseudolayer that enables a flag in the layer below to tell Perl that output should be in utf8 and that input should be regarded as already in valid utf8 form. **WARNING: It does not check for validity and as such should be handled with extreme caution for input, because security violations can occur with non-shortest UTF-8 encodings, etc.** Generally `:encoding(utf8)` is the best option when reading UTF-8 encoded data.

:win32

On Win32 platforms this *experimental* layer uses native "handle" IO rather than a Unix-like numeric file descriptor layer. Known to be buggy in this release (5.14).

The default set of layers should give acceptable results on all platforms

For Unix platforms that will be the equivalent of "unix perlio" or "stdio". Configure is set up to prefer the "stdio" implementation if the system's library provides for fast access to the buffer; otherwise, it uses the "unix perlio" implementation.

On Win32 the default in this release (5.14) is "unix crlf". Win32's "stdio" has a number of bugs/mis-features for Perl IO which are somewhat depending on the version and vendor of the C compiler. Using our own `crlf` layer as the buffer avoids those issues and makes things more uniform. The `crlf` layer provides CRLF conversion as well as buffering.

This release (5.14) uses `unix` as the bottom layer on Win32, and so still uses the C compiler's numeric file descriptor routines. There is an experimental native `win32` layer, which is expected to be enhanced and should eventually become the default under Win32.

The PERLIO environment variable is completely ignored when Perl is run in taint mode.

PERLIO_DEBUG

If set to the name of a file or device, certain operations of PerlIO subsystem will be logged to that file, which is opened in append mode. Typical uses are in Unix:

```
% env PERLIO_DEBUG=/dev/tty perl script ...
```

and under Win32, the approximately equivalent:

```
> set PERLIO_DEBUG=CON
perl script ...
```

This functionality is disabled for setuid scripts and for scripts run with **-T**.

PERLLIB

A list of directories in which to look for Perl library files before looking in the standard library and the current directory. If PERL5LIB is defined, PERLLIB is not used.

The PERLLIB environment variable is completely ignored when Perl is run in taint mode.

PERL5DB

The command used to load the debugger code. The default is:

```
        BEGIN { require "perl5db.pl" }
```

The PERL5DB environment variable is only used when Perl is started with a bare **-d** switch.

PERL5DB_THREADED

If set to a true value, indicates to the debugger that the code being debugged uses threads.

PERL5SHELL (specific to the Win32 port)

On Win32 ports only, may be set to an alternative shell that Perl must use internally for executing "backtick" commands or system(). Default is `cmd.exe`

`/x/d/c` on WindowsNT and `command.com /c` on Windows95. The value is considered space-separated. Precede any character that needs to be protected, like a space or backslash, with another backslash.

Note that Perl doesn't use COMSPEC for this purpose because COMSPEC has a high degree of variability among users, leading to portability concerns. Besides, Perl can use a shell that may not be fit for interactive use, and setting COMSPEC to such a shell may interfere with the proper functioning of other programs (which usually look in COMSPEC to find a shell fit for interactive use).

Before Perl 5.10.0 and 5.8.8, PERL5SHELL was not taint checked when running external commands. It is recommended that you explicitly set (or delete) `$ENV{PERL5SHELL}` when running in taint mode under Windows.

PERL_ALLOW_NON_IFS_LSP (specific to the Win32 port)

Set to 1 to allow the use of non-IFS compatible LSPs (Layered Service Providers). Perl normally searches for an IFS-compatible LSP because this is required for its emulation of Windows sockets as real filehandles. However, this may cause problems if you have a firewall such as *McAfee Guardian*, which requires that all applications use its LSP but which is not IFS-compatible, because clearly Perl will normally avoid using such an LSP.

Setting this environment variable to 1 means that Perl will simply use the first suitable LSP enumerated in the catalog, which keeps *McAfee Guardian* happy–and in that particular case Perl still works too because *McAfee Guardian*'s LSP actually plays other games which allow applications requiring IFS compatibility to work.

PERL_DEBUG_MSTATS

Relevant only if Perl is compiled with the `malloc` included with the Perl distribution; that is, if `perl -V:d_mymalloc` is "define".

If set, this dumps out memory statistics after execution. If set to an integer greater than one, also dumps out memory statistics after compilation.

PERL_DESTRUCT_LEVEL

Relevant only if your Perl executable was built with **-DDEBUGGING**, this controls the behaviour of global destruction of objects and other references. See Section 30.8.1 [perlhacktips PERL_DESTRUCT_LEVEL], page 596 for more information.

PERL_DL_NONLAZY

Set to `"1"` to have Perl resolve *all* undefined symbols when it loads a dynamic library. The default behaviour is to resolve symbols when they are used. Setting this variable is useful during testing of extensions, as it ensures that you get an error on misspelled function names even if the test suite doesn't call them.

PERL_ENCODING

If using the `use encoding` pragma without an explicit encoding name, the PERL_ENCODING environment variable is consulted for an encoding name.

PERL_HASH_SEED

(Since Perl 5.8.1, new semantics in Perl 5.18.0) Used to override the randomization of Perl's internal hash function. The value is expressed in hexadecimal, and may include a leading 0x. Truncated patterns are treated as though they are suffixed with sufficient 0's as required.

If the option is provided, and `PERL_PERTURB_KEYS` is NOT set, then a value of '0' implies `PERL_PERTURB_KEYS=0` and any other value implies `PERL_PERTURB_KEYS=2`.

PLEASE NOTE: The hash seed is sensitive information. Hashes are randomized to protect against local and remote attacks against Perl code. By manually setting a seed, this protection may be partially or completely lost.

See Section 70.4.9 [perlsec Algorithmic Complexity Attacks], page 1205, [PERL_PERTURB_KEYS], page 1195, and [PERL_HASH_SEED_DEBUG], page 1195 for more information.

PERL_PERTURB_KEYS

(Since Perl 5.18.0) Set to `"0"` or `"NO"` then traversing keys will be repeatable from run to run for the same PERL_HASH_SEED. Insertion into a hash will not change the order, except to provide for more space in the hash. When combined with setting PERL_HASH_SEED this mode is as close to pre 5.18 behavior as you can get.

When set to `"1"` or `"RANDOM"` then traversing keys will be randomized. Every time a hash is inserted into the key order will change in a random fashion. The order may not be repeatable in a following program run even if the PERL_HASH_SEED has been specified. This is the default mode for perl.

When set to `"2"` or `"DETERMINISTIC"` then inserting keys into a hash will cause the key order to change, but in a way that is repeatable from program run to program run.

NOTE: Use of this option is considered insecure, and is intended only for debugging non-deterministic behavior in Perl's hash function. Do not use it in production.

See Section 70.4.9 [perlsec Algorithmic Complexity Attacks], page 1205 and [PERL_HASH_SEED], page 1195 and [PERL_HASH_SEED_DEBUG], page 1195 for more information. You can get and set the key traversal mask for a specific hash by using the `hash_traversal_mask()` function from `Hash-Util`.

PERL_HASH_SEED_DEBUG

(Since Perl 5.8.1.) Set to `"1"` to display (to STDERR) information about the hash function, seed, and what type of key traversal randomization is in effect at the beginning of execution. This, combined with [PERL_HASH_SEED], page 1195 and [PERL_PERTURB_KEYS], page 1195 is intended to aid in debugging nondeterministic behaviour caused by hash randomization.

Note that any information about the hash function, especially the hash seed is **sensitive information**: by knowing it, one can craft a denial-of-service attack against Perl code, even remotely; see Section 70.4.9 [perlsec Algorithmic

Complexity Attacks], page 1205 for more information. **Do not disclose the hash seed** to people who don't need to know it. See also `hash_seed()` and `key_traversal_mask()` in `Hash-Util`.

An example output might be:

```
 HASH_FUNCTION = ONE_AT_A_TIME_HARD HASH_SEED = 0x652e9b9349a7a032 PERTURB_KEY
```

PERL_MEM_LOG

If your Perl was configured with **-Accflags=-DPERL_MEM_LOG**, setting the environment variable `PERL_MEM_LOG` enables logging debug messages. The value has the form `<`*`number`*`>[m][s][t]`, where *`number`* is the file descriptor number you want to write to (2 is default), and the combination of letters specifies that you want information about (m)emory and/or (s)v, optionally with (t)imestamps. For example, `PERL_MEM_LOG=1mst` logs all information to stdout. You can write to other opened file descriptors in a variety of ways:

```
$ 3>foo3 PERL_MEM_LOG=3m perl ...
```

PERL_ROOT (specific to the VMS port)

A translation-concealed rooted logical name that contains Perl and the logical device for the @INC path on VMS only. Other logical names that affect Perl on VMS include PERLSHR, PERL_ENV_TABLES, and SYS$TIMEZONE_DIFFERENTIAL, but are optional and discussed further in Section 87.1 [perlvms NAME], page 1409 and in `README.vms` in the Perl source distribution.

PERL_SIGNALS

Available in Perls 5.8.1 and later. If set to `"unsafe"`, the pre-Perl-5.8.0 signal behaviour (which is immediate but unsafe) is restored. If set to `safe`, then safe (but deferred) signals are used. See Section 36.3.2 [perlipc Deferred Signals (Safe Signals)], page 671.

PERL_UNICODE

Equivalent to the **-C** command-line switch. Note that this is not a boolean variable. Setting this to `"1"` is not the right way to "enable Unicode" (whatever that would mean). You can use `"0"` to "disable Unicode", though (or alternatively unset PERL_UNICODE in your shell before starting Perl). See the description of the **-C** switch for more information.

SYS$LOGIN (specific to the VMS port)

Used if chdir has no argument and HOME and LOGDIR are not set.

Perl also has environment variables that control how Perl handles data specific to particular natural languages; see Section 38.1 [perllocale NAME], page 701.

Perl and its various modules and components, including its test frameworks, may sometimes make use of certain other environment variables. Some of these are specific to a particular platform. Please consult the appropriate module documentation and any documentation for your platform (like `perlsolaris`, `perllinux`, `perlmacosx`, `perlwin32`, etc) for variables peculiar to those specific situations.

Perl makes all environment variables available to the program being executed, and passes these along to any child processes it starts. However, programs running setuid would do well to execute the following lines before doing anything else, just to keep people honest:

```
$ENV{PATH}  = "/bin:/usr/bin";    # or whatever you need
$ENV{SHELL} = "/bin/sh" if exists $ENV{SHELL};
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

70 perlsec

70.1 NAME

perlsec - Perl security

70.2 DESCRIPTION

Perl is designed to make it easy to program securely even when running with extra privileges, like setuid or setgid programs. Unlike most command line shells, which are based on multiple substitution passes on each line of the script, Perl uses a more conventional evaluation scheme with fewer hidden snags. Additionally, because the language has more builtin functionality, it can rely less upon external (and possibly untrustworthy) programs to accomplish its purposes.

70.3 SECURITY VULNERABILITY CONTACT INFORMATION

If you believe you have found a security vulnerability in Perl, please email perl5-security-report@perl.org with details. This points to a closed subscription, unarchived mailing list. Please only use this address for security issues in the Perl core, not for modules independently distributed on CPAN.

70.4 SECURITY MECHANISMS AND CONCERNS

70.4.1 Taint mode

Perl automatically enables a set of special security checks, called *taint mode*, when it detects its program running with differing real and effective user or group IDs. The setuid bit in Unix permissions is mode 04000, the setgid bit mode 02000; either or both may be set. You can also enable taint mode explicitly by using the **-T** command line flag. This flag is *strongly* suggested for server programs and any program run on behalf of someone else, such as a CGI script. Once taint mode is on, it's on for the remainder of your script.

While in this mode, Perl takes special precautions called *taint checks* to prevent both obvious and subtle traps. Some of these checks are reasonably simple, such as verifying that path directories aren't writable by others; careful programmers have always used checks like these. Other checks, however, are best supported by the language itself, and it is these checks especially that contribute to making a set-id Perl program more secure than the corresponding C program.

You may not use data derived from outside your program to affect something else outside your program–at least, not by accident. All command line arguments, environment variables, locale information (see Section 38.1 [perllocale NAME], page 701), results of certain system calls (`readdir()`, `readlink()`, the variable of `shmread()`, the messages returned by `msgrcv()`, the password, gcos and shell fields returned by the `getpwxxx()` calls), and all file input are marked as `"tainted"`. Tainted data may not be used directly or indirectly in any command that invokes a sub-shell, nor in any command that modifies files, directories, or processes, **with the following exceptions**:

- Arguments to `print` and `syswrite` are **not** checked for taintedness.
- Symbolic methods

  ```
  $obj->$method(@args);
  ```

  and symbolic sub references

  ```
  &{$foo}(@args);
  $foo->(@args);
  ```

  are not checked for taintedness. This requires extra carefulness unless you want external data to affect your control flow. Unless you carefully limit what these symbolic values are, people are able to call functions **outside** your Perl code, such as POSIX::system, in which case they are able to run arbitrary external code.
- Hash keys are **never** tainted.

For efficiency reasons, Perl takes a conservative view of whether data is tainted. If an expression contains tainted data, any subexpression may be considered tainted, even if the value of the subexpression is not itself affected by the tainted data.

Because taintedness is associated with each scalar value, some elements of an array or hash can be tainted and others not. The keys of a hash are **never** tainted.

For example:

```
$arg = shift;               # $arg is tainted
$hid = $arg . 'bar';        # $hid is also tainted
$line = <>;                 # Tainted
$line = <STDIN>;            # Also tainted
open FOO, "/home/me/bar" or die $!;
$line = <FOO>;              # Still tainted
$path = $ENV{'PATH'};       # Tainted, but see below
$data = 'abc';              # Not tainted

system "echo $arg";         # Insecure
system "/bin/echo", $arg;   # Considered insecure
                            # (Perl doesn't know about /bin/echo)
system "echo $hid";         # Insecure
system "echo $data";        # Insecure until PATH set

$path = $ENV{'PATH'};       # $path now tainted

$ENV{'PATH'} = '/bin:/usr/bin';
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};

$path = $ENV{'PATH'};       # $path now NOT tainted
system "echo $data";        # Is secure now!

open(FOO, "< $arg");        # OK - read-only file
open(FOO, "> $arg");        # Not OK - trying to write

open(FOO,"echo $arg|");     # Not OK
```

```
open(FOO,"-|")
    or exec ’echo’, $arg;   # Also not OK

$shout = ‘echo $arg‘;       # Insecure, $shout now tainted

unlink $data, $arg;         # Insecure
umask $arg;                 # Insecure

exec "echo $arg";           # Insecure
exec "echo", $arg;          # Insecure
exec "sh", ’-c’, $arg;      # Very insecure!

@files = <*.c>;             # insecure (uses readdir() or similar)
@files = glob(’*.c’);       # insecure (uses readdir() or similar)

# In either case, the results of glob are tainted, since the list of
# filenames comes from outside of the program.

$bad = ($arg, 23);          # $bad will be tainted
$arg, ‘true‘;               # Insecure (although it isn’t really)
```

If you try to do something insecure, you will get a fatal error saying something like "Insecure dependency" or "Insecure $ENV{PATH}".

The exception to the principle of "one tainted value taints the whole expression" is with the ternary conditional operator `?:`. Since code with a ternary conditional

```
$result = $tainted_value ? "Untainted" : "Also untainted";
```

is effectively

```
if ( $tainted_value ) {
    $result = "Untainted";
} else {
    $result = "Also untainted";
}
```

it doesn’t make sense for `$result` to be tainted.

70.4.2 Laundering and Detecting Tainted Data

To test whether a variable contains tainted data, and whose use would thus trigger an "Insecure dependency" message, you can use the `tainted()` function of the Scalar::Util module, available in your nearby CPAN mirror, and included in Perl starting from the release 5.8.0. Or you may be able to use the following `is_tainted()` function.

```
sub is_tainted {
    local $@;   # Don’t pollute caller’s value.
    return ! eval { eval("#" . substr(join("", @_), 0, 0)); 1 };
}
```

This function makes use of the fact that the presence of tainted data anywhere within an expression renders the entire expression tainted. It would be inefficient for every operator to test every argument for taintedness. Instead, the slightly more efficient and conservative approach is used that if any tainted value has been accessed within the same expression, the whole expression is considered tainted.

But testing for taintedness gets you only so far. Sometimes you have just to clear your data's taintedness. Values may be untainted by using them as keys in a hash; otherwise the only way to bypass the tainting mechanism is by referencing subpatterns from a regular expression match. Perl presumes that if you reference a substring using $1, $2, etc. in a non-tainting pattern, that you knew what you were doing when you wrote that pattern. That means using a bit of thought–don't just blindly untaint anything, or you defeat the entire mechanism. It's better to verify that the variable has only good characters (for certain values of "good") rather than checking whether it has any bad characters. That's because it's far too easy to miss bad characters that you never thought of.

Here's a test to make sure that the data contains nothing but "word" characters (alphabetics, numerics, and underscores), a hyphen, an at sign, or a dot.

```
if ($data =~ /^([-\@\w.]+)$/) {
    $data = $1;                     # $data now untainted
} else {
    die "Bad data in '$data'";      # log this somewhere
}
```

This is fairly secure because `/\w+/` doesn't normally match shell metacharacters, nor are dot, dash, or at going to mean something special to the shell. Use of `/.+/` would have been insecure in theory because it lets everything through, but Perl doesn't check for that. The lesson is that when untainting, you must be exceedingly careful with your patterns. Laundering data using regular expression is the *only* mechanism for untainting dirty data, unless you use the strategy detailed below to fork a child of lesser privilege.

The example does not untaint `$data` if `use locale` is in effect, because the characters matched by `\w` are determined by the locale. Perl considers that locale definitions are untrustworthy because they contain data from outside the program. If you are writing a locale-aware program, and want to launder data with a regular expression containing `\w`, put `no locale` ahead of the expression in the same block. See Section 38.7 [perllocale SECURITY], page 716 for further discussion and examples.

70.4.3 Switches On the "#!" Line

When you make a script executable, in order to make it usable as a command, the system will pass switches to perl from the script's #! line. Perl checks that any command line switches given to a setuid (or setgid) script actually match the ones set on the #! line. Some Unix and Unix-like environments impose a one-switch limit on the #! line, so you may need to use something like `-wU` instead of `-w -U` under such systems. (This issue should arise only in Unix or Unix-like environments that support #! and setuid or setgid scripts.)

70.4.4 Taint mode and @INC

When the taint mode (`-T`) is in effect, the "." directory is removed from `@INC`, and the environment variables `PERL5LIB` and `PERLLIB` are ignored by Perl. You can still adjust `@INC` from outside the program by using the `-I` command line option as explained in Section 69.1 [perlrun NAME], page 1176. The two environment variables are ignored because they are obscured, and a user running a program could be unaware that they are set, whereas the `-I` option is clearly visible and therefore permitted.

Another way to modify @INC without modifying the program, is to use the lib pragma, e.g.:

```
perl -Mlib=/foo program
```

The benefit of using -Mlib=/foo over -I/foo, is that the former will automagically remove any duplicated directories, while the latter will not.

Note that if a tainted string is added to @INC, the following problem will be reported:

```
Insecure dependency in require while running with -T switch
```

70.4.5 Cleaning Up Your Path

For "Insecure $ENV{PATH}" messages, you need to set $ENV{'PATH'} to a known value, and each directory in the path must be absolute and non-writable by others than its owner and group. You may be surprised to get this message even if the pathname to your executable is fully qualified. This is *not* generated because you didn't supply a full path to the program; instead, it's generated because you never set your PATH environment variable, or you didn't set it to something that was safe. Because Perl can't guarantee that the executable in question isn't itself going to turn around and execute some other program that is dependent on your PATH, it makes sure you set the PATH.

The PATH isn't the only environment variable which can cause problems. Because some shells may use the variables IFS, CDPATH, ENV, and BASH_ENV, Perl checks that those are either empty or untainted when starting subprocesses. You may wish to add something like this to your setid and taint-checking scripts.

```
  delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};   # Make %ENV safer
```

It's also possible to get into trouble with other operations that don't care whether they use tainted values. Make judicious use of the file tests in dealing with any user-supplied filenames. When possible, do opens and such **after** properly dropping any special user (or group!) privileges. Perl doesn't prevent you from opening tainted filenames for reading, so be careful what you print out. The tainting mechanism is intended to prevent stupid mistakes, not to remove the need for thought.

Perl does not call the shell to expand wild cards when you pass system and exec explicit parameter lists instead of strings with possible shell wildcards in them. Unfortunately, the open, glob, and backtick functions provide no such alternate calling convention, so more subterfuge will be required.

Perl provides a reasonably safe way to open a file or pipe from a setuid or setgid program: just create a child process with reduced privilege who does the dirty work for you. First, fork a child using the special open syntax that connects the parent and child by a pipe. Now the child resets its ID set and any other per-process attributes, like environment variables, umasks, current working directories, back to the originals or known safe values. Then the child process, which no longer has any special permissions, does the open or other system call. Finally, the child passes the data it managed to access back to the parent. Because the file or pipe was opened in the child while running under less privilege than the parent, it's not apt to be tricked into doing something it shouldn't.

Here's a way to do backticks reasonably safely. Notice how the exec is not called with a string that the shell could expand. This is by far the best way to call something that might be subjected to shell escapes: just never call the shell at all.

```
use English;
die "Can't fork: $!" unless defined($pid = open(KID, "-|"));
if ($pid) {           # parent
    while (<KID>) {
        # do something
    }
    close KID;
} else {
    my @temp     = ($EUID, $EGID);
    my $orig_uid = $UID;
    my $orig_gid = $GID;
    $EUID = $UID;
    $EGID = $GID;
    # Drop privileges
    $UID  = $orig_uid;
    $GID  = $orig_gid;
    # Make sure privs are really gone
    ($EUID, $EGID) = @temp;
    die "Can't drop privileges"
        unless $UID == $EUID  && $GID eq $EGID;
    $ENV{PATH} = "/bin:/usr/bin"; # Minimal PATH.
    # Consider sanitizing the environment even more.
    exec 'myprog', 'arg1', 'arg2'
        or die "can't exec myprog: $!";
}
```

A similar strategy would work for wildcard expansion via `glob`, although you can use `readdir` instead.

Taint checking is most useful when although you trust yourself not to have written a program to give away the farm, you don't necessarily trust those who end up using it not to try to trick it into doing something bad. This is the kind of security checking that's useful for set-id programs and programs launched on someone else's behalf, like CGI programs.

This is quite different, however, from not even trusting the writer of the code not to try to do something evil. That's the kind of trust needed when someone hands you a program you've never seen before and says, "Here, run this." For that kind of safety, you might want to check out the Safe module, included standard in the Perl distribution. This module allows the programmer to set up special compartments in which all system operations are trapped and namespace access is carefully controlled. Safe should not be considered bullet-proof, though: it will not prevent the foreign code to set up infinite loops, allocate gigabytes of memory, or even abusing perl bugs to make the host interpreter crash or behave in unpredictable ways. In any case it's better avoided completely if you're really concerned about security.

70.4.6 Security Bugs

Beyond the obvious problems that stem from giving special privileges to systems as flexible as scripts, on many versions of Unix, set-id scripts are inherently insecure right from the start. The problem is a race condition in the kernel. Between the time the kernel opens the file to see which interpreter to run and when the (now-set-id) interpreter turns around and reopens the file to interpret it, the file in question may have changed, especially if you have symbolic links on your system.

Fortunately, sometimes this kernel "feature" can be disabled. Unfortunately, there are two ways to disable it. The system can simply outlaw scripts with any set-id bit set, which doesn't help much. Alternately, it can simply ignore the set-id bits on scripts.

However, if the kernel set-id script feature isn't disabled, Perl will complain loudly that your set-id script is insecure. You'll need to either disable the kernel set-id script feature, or put a C wrapper around the script. A C wrapper is just a compiled program that does nothing except call your Perl program. Compiled programs are not subject to the kernel bug that plagues set-id scripts. Here's a simple wrapper, written in C:

```
#define REAL_PATH "/path/to/script"
main(ac, av)
    char **av;
{
    execv(REAL_PATH, av);
}
```

Compile this wrapper into a binary executable and then make *it* rather than your script setuid or setgid.

In recent years, vendors have begun to supply systems free of this inherent security bug. On such systems, when the kernel passes the name of the set-id script to open to the interpreter, rather than using a pathname subject to meddling, it instead passes */dev/fd/3*. This is a special file already opened on the script, so that there can be no race condition for evil scripts to exploit. On these systems, Perl should be compiled with `-DSETUID_SCRIPTS_ARE_SECURE_NOW`. The `Configure` program that builds Perl tries to figure this out for itself, so you should never have to specify this yourself. Most modern releases of SysVr4 and BSD 4.4 use this approach to avoid the kernel race condition.

70.4.7 Protecting Your Programs

There are a number of ways to hide the source to your Perl programs, with varying levels of "security".

First of all, however, you *can't* take away read permission, because the source code has to be readable in order to be compiled and interpreted. (That doesn't mean that a CGI script's source is readable by people on the web, though.) So you have to leave the permissions at the socially friendly 0755 level. This lets people on your local system only see your source.

Some people mistakenly regard this as a security problem. If your program does insecure things, and relies on people not knowing how to exploit those insecurities, it is not secure. It is often possible for someone to determine the insecure things and exploit them without viewing the source. Security through obscurity, the name for hiding your bugs instead of fixing them, is little security indeed.

You can try using encryption via source filters (Filter::* from CPAN, or Filter::Util::Call and Filter::Simple since Perl 5.8). But crackers might be able to decrypt it. You can try using the byte code compiler and interpreter described below, but crackers might be able to de-compile it. You can try using the native-code compiler described below, but crackers might be able to disassemble it. These pose varying degrees of difficulty to people wanting to get at your code, but none can definitively conceal it (this is true of every language, not just Perl).

If you're concerned about people profiting from your code, then the bottom line is that nothing but a restrictive license will give you legal security. License your software and pepper it with threatening statements like "This is unpublished proprietary software of XYZ Corp. Your access to it does not give you permission to use it blah blah blah." You should see a lawyer to be sure your license's wording will stand up in court.

70.4.8 Unicode

Unicode is a new and complex technology and one may easily overlook certain security pitfalls. See Section 83.1 [perluniintro NAME], page 1352 for an overview and Section 81.1 [perlunicode NAME], page 1317 for details, and Section 81.2.13 [perlunicode Security Implications of Unicode], page 1339 for security implications in particular.

70.4.9 Algorithmic Complexity Attacks

Certain internal algorithms used in the implementation of Perl can be attacked by choosing the input carefully to consume large amounts of either time or space or both. This can lead into the so-called *Denial of Service* (DoS) attacks.

- Hash Algorithm - Hash algorithms like the one used in Perl are well known to be vulnerable to collision attacks on their hash function. Such attacks involve constructing a set of keys which collide into the same bucket producing inefficient behavior. Such attacks often depend on discovering the seed of the hash function used to map the keys to buckets. That seed is then used to brute-force a key set which can be used to mount a denial of service attack. In Perl 5.8.1 changes were introduced to harden Perl to such attacks, and then later in Perl 5.18.0 these features were enhanced and additional protections added.

  At the time of this writing, Perl 5.18.0 is considered to be well-hardened against algorithmic complexity attacks on its hash implementation. This is largely owed to the following measures mitigate attacks:

  Hash Seed Randomization

  In order to make it impossible to know what seed to generate an attack key set for, this seed is randomly initialized at process start. This may be overridden by using the PERL_HASH_SEED environment variable, see [perlrun PERL_HASH_SEED], page 1195. This environment variable controls how items are actually stored, not how they are presented via `keys`, `values` and `each`.

  Hash Traversal Randomization

  Independent of which seed is used in the hash function, `keys`, `values`, and `each` return items in a per-hash randomized order. Modifying a hash by insertion will change the iteration order of that hash. This behavior can be overridden by using `hash_traversal_mask()` from `Hash-Util` or by using the PERL_PERTURB_KEYS environment variable, see [perlrun PERL_PERTURB_KEYS], page 1195. Note that this feature controls the "visible" order of the keys, and not the actual order they are stored in.

Bucket Order Perturbance

When items collide into a given hash bucket the order they are stored in the chain is no longer predictable in Perl 5.18. This has the intention to make it harder to observe a collision. This behavior can be overridden by using the PERL_PERTURB_KEYS environment variable, see [perlrun PERL_PERTURB_KEYS], page 1195.

New Default Hash Function

The default hash function has been modified with the intention of making it harder to infer the hash seed.

Alternative Hash Functions

The source code includes multiple hash algorithms to choose from. While we believe that the default perl hash is robust to attack, we have included the hash function Siphash as a fall-back option. At the time of release of Perl 5.18.0 Siphash is believed to be of cryptographic strength. This is not the default as it is much slower than the default hash.

Without compiling a special Perl, there is no way to get the exact same behavior of any versions prior to Perl 5.18.0. The closest one can get is by setting PERL_PERTURB_KEYS to 0 and setting the PERL_HASH_SEED to a known value. We do not advise those settings for production use due to the above security considerations.

Perl has never guaranteed any ordering of the hash keys, and the ordering has already changed several times during the lifetime of Perl 5. Also, the ordering of hash keys has always been, and continues to be, affected by the insertion order and the history of changes made to the hash over its lifetime.

Also note that while the order of the hash elements might be randomized, this "pseudo-ordering" should **not** be used for applications like shuffling a list randomly (use `List::Util::shuffle()` for that, see `List-Util`, a standard core module since Perl 5.8.0; or the CPAN module `Algorithm::Numerical::Shuffle`), or for generating permutations (use e.g. the CPAN modules `Algorithm::Permute` or `Algorithm::FastPermute`), or for any cryptographic applications.

Tied hashes may have their own ordering and algorithmic complexity attacks.

- Regular expressions - Perl's regular expression engine is so called NFA (Non-deterministic Finite Automaton), which among other things means that it can rather easily consume large amounts of both time and space if the regular expression may match in several ways. Careful crafting of the regular expressions can help but quite often there really isn't much one can do (the book "Mastering Regular Expressions" is required reading, see `perlfaq2`). Running out of space manifests itself by Perl running out of memory.
- Sorting - the quicksort algorithm used in Perls before 5.8.0 to implement the sort() function is very easy to trick into misbehaving so that it consumes a lot of time. Starting from Perl 5.8.0 a different sorting algorithm, mergesort, is used by default. Mergesort cannot misbehave on any input.

See `http://www.cs.rice.edu/~scrosby/hash/` for more information, and any computer science textbook on algorithmic complexity.

70.5 SEE ALSO

Section 69.1 [perlrun NAME], page 1176 for its description of cleaning up environment variables.

71 perlsource

71.1 NAME

perlsource - A guide to the Perl source tree

71.2 DESCRIPTION

This document describes the layout of the Perl source tree. If you're hacking on the Perl core, this will help you find what you're looking for.

71.3 FINDING YOUR WAY AROUND

The Perl source tree is big. Here's some of the thing you'll find in it:

71.3.1 C code

The C source code and header files mostly live in the root of the source tree. There are a few platform-specific directories which contain C code. In addition, some of the modules shipped with Perl include C or XS code.

See Section 33.1 [perlinterp NAME], page 627 for more details on the files that make up the Perl interpreter, as well as details on how it works.

71.3.2 Core modules

Modules shipped as part of the Perl core live in four subdirectories. Two of these directories contain modules that live in the core, and two contain modules that can also be released separately on CPAN. Modules which can be released on cpan are known as "dual-life" modules.

- `lib/`

  This directory contains pure-Perl modules which are only released as part of the core. This directory contains *all* of the modules and their tests, unlike other core modules.

- `ext/`

  Like `lib/`, this directory contains modules which are only released as part of the core. Unlike `lib/`, however, a module under `ext/` generally has a CPAN-style directory- and file-layout and its own `Makefile.PL`. There is no expectation that a module under `ext/` will work with earlier versions of Perl 5. Hence, such a module may take full advantage of syntactical and other improvements in Perl 5 blead.

- `dist/`

  This directory is for dual-life modules where the blead source is canonical. Note that some modules in this directory may not yet have been released separately on CPAN. Modules under `dist/` should make an effort to work with earlier versions of Perl 5.

- `cpan/`

  This directory contains dual-life modules where the CPAN module is canonical. Do not patch these modules directly! Changes to these modules should be submitted to the maintainer of the CPAN module. Once those changes are applied and released, the new version of the module will be incorporated into the core.

For some dual-life modules, it has not yet been determined if the CPAN version or the blead source is canonical. Until that is done, those modules should be in `cpan/`.

71.3.3 Tests

The Perl core has an extensive test suite. If you add new tests (or new modules with tests), you may need to update the `t/TEST` file so that the tests are run.

- Module tests

  Tests for core modules in the `lib/` directory are right next to the module itself. For example, we have `lib/strict.pm` and `lib/strict.t`.

  Tests for modules in `ext/` and the dual-life modules are in `t/` subdirectories for each module, like a standard CPAN distribution.

- `t/base/`

  Tests for the absolute basic functionality of Perl. This includes `if`, basic file reads and writes, simple regexes, etc. These are run first in the test suite and if any of them fail, something is *really* broken.

- `t/cmd/`

  Tests for basic control structures, `if/else`, `while`, subroutines, etc.

- `t/comp/`

  Tests for basic issues of how Perl parses and compiles itself.

- `t/io/`

  Tests for built-in IO functions, including command line arguments.

- `t/mro/`

  Tests for perl's method resolution order implementations (see `mro`).

- `t/op/`

  Tests for perl's built in functions that don't fit into any of the other directories.

- `t/opbasic/`

  Tests for perl's built in functions which, like those in `t/op/`, do not fit into any of the other directories, but which, in addition, cannot use `t/test.pl`,as that program depends on functionality which the test file itself is testing.

- `t/re/`

  Tests for regex related functions or behaviour. (These used to live in t/op).

- `t/run/`

  Tests for features of how perl actually runs, including exit codes and handling of PERL* environment variables.

- `t/uni/`

  Tests for the core support of Unicode.

- `t/win32/`

  Windows-specific tests.

- `t/porting/`

  Tests the state of the source tree for various common errors. For example, it tests that everyone who is listed in the git log has a corresponding entry in the `AUTHORS` file.

- `t/lib/`

  The old home for the module tests, you shouldn't put anything new in here. There are still some bits and pieces hanging around in here that need to be moved. Perhaps you could move them? Thanks!

71.3.4 Documentation

All of the core documentation intended for end users lives in `pod/`. Individual modules in `lib/`, `ext/`, `dist/`, and `cpan/` usually have their own documentation, either in the `Module.pm` file or an accompanying `Module.pod` file.

Finally, documentation intended for core Perl developers lives in the `Porting/` directory.

71.3.5 Hacking tools and documentation

The `Porting` directory contains a grab bag of code and documentation intended to help porters work on Perl. Some of the highlights include:

- `check*`

  These are scripts which will check the source things like ANSI C violations, POD encoding issues, etc.

- `Maintainers`, `Maintainers.pl`, and `Maintainers.pm`

  These files contain information on who maintains which modules. Run `perl Porting/Maintainers -M Module::Name` to find out more information about a dual-life module.

- `podtidy`

  Tidies a pod file. It's a good idea to run this on a pod file you've patched.

71.3.6 Build system

The Perl build system starts with the `Configure` script in the root directory.

Platform-specific pieces of the build system also live in platform-specific directories like `win32/`, `vms/`, etc.

The `Configure` script is ultimately responsible for generating a `Makefile`.

The build system that Perl uses is called metaconfig. This system is maintained separately from the Perl core.

The metaconfig system has its own git repository. Please see its README file in `http://perl5.git.perl.org/metaconfig.git/` for more details.

The `Cross` directory contains various files related to cross-compiling Perl. See `Cross/README` for more details.

71.3.7 `AUTHORS`

This file lists everyone who's contributed to Perl. If you submit a patch, you should add your name to this file as part of the patch.

71.3.8 `MANIFEST`

The `MANIFEST` file in the root of the source tree contains a list of every file in the Perl core, as well as a brief description of each file.

You can get an overview of all the files with this command:

```
% perl -lne ’print if /^[^\/]+\.[ch]\s+/’ MANIFEST
```

72 perlstyle

72.1 NAME

perlstyle - Perl style guide

72.2 DESCRIPTION

Each programmer will, of course, have his or her own preferences in regards to formatting, but there are some general guidelines that will make your programs easier to read, understand, and maintain.

The most important thing is to run your programs under the **-w** flag at all times. You may turn it off explicitly for particular portions of code via the `no warnings` pragma or the `$^W` variable if you must. You should also always run under `use strict` or know the reason why not. The `use sigtrap` and even `use diagnostics` pragmas may also prove useful.

Regarding aesthetics of code lay out, about the only thing Larry cares strongly about is that the closing curly bracket of a multi-line BLOCK should line up with the keyword that started the construct. Beyond that, he has other preferences that aren't so strong:

- 4-column indent.
- Opening curly on same line as keyword, if possible, otherwise line up.
- Space before the opening curly of a multi-line BLOCK.
- One-line BLOCK may be put on one line, including curlies.
- No space before the semicolon.
- Semicolon omitted in "short" one-line BLOCK.
- Space around most operators.
- Space around a "complex" subscript (inside brackets).
- Blank lines between chunks that do different things.
- Uncuddled elses.
- No space between function name and its opening parenthesis.
- Space after each comma.
- Long lines broken after an operator (except `and` and `or`).
- Space after last parenthesis matching on current line.
- Line up corresponding items vertically.
- Omit redundant punctuation as long as clarity doesn't suffer.

Larry has his reasons for each of these things, but he doesn't claim that everyone else's mind works the same as his does.

Here are some other more substantive style issues to think about:

- Just because you *CAN* do something a particular way doesn't mean that you *SHOULD* do it that way. Perl is designed to give you several ways to do anything, so consider picking the most readable one. For instance

  ```
  open(FOO,$foo) || die "Can't open $foo: $!";
  ```

  is better than

```
die "Can't open $foo: $!" unless open(FOO,$foo);
```

because the second way hides the main point of the statement in a modifier. On the other hand

```
print "Starting analysis\n" if $verbose;
```

is better than

```
$verbose && print "Starting analysis\n";
```

because the main point isn't whether the user typed **-v** or not.

Similarly, just because an operator lets you assume default arguments doesn't mean that you have to make use of the defaults. The defaults are there for lazy systems programmers writing one-shot programs. If you want your program to be readable, consider supplying the argument.

Along the same lines, just because you *CAN* omit parentheses in many places doesn't mean that you ought to:

```
return print reverse sort num values %array;
return print(reverse(sort num (values(%array))));
```

When in doubt, parenthesize. At the very least it will let some poor schmuck bounce on the % key in **vi**.

Even if you aren't in doubt, consider the mental welfare of the person who has to maintain the code after you, and who will probably put parentheses in the wrong place.

- Don't go through silly contortions to exit a loop at the top or the bottom, when Perl provides the `last` operator so you can exit in the middle. Just "outdent" it a little to make it more visible:

```
LINE:
    for (;;) {
        statements;
      last LINE if $foo;
        next LINE if /^#/;
        statements;
    }
```

- Don't be afraid to use loop labels–they're there to enhance readability as well as to allow multilevel loop breaks. See the previous example.
- Avoid using `grep()` (or `map()`) or 'backticks' in a void context, that is, when you just throw away their return values. Those functions all have return values, so use them. Otherwise use a `foreach()` loop or the `system()` function instead.
- For portability, when using features that may not be implemented on every machine, test the construct in an eval to see if it fails. If you know what version or patchlevel a particular feature was implemented, you can test `$]` (`$PERL_VERSION` in `English`) to see if it will be there. The `Config` module will also let you interrogate values determined by the **Configure** program when Perl was installed.
- Choose mnemonic identifiers. If you can't remember what mnemonic means, you've got a problem.
- While short identifiers like `$gotit` are probably ok, use underscores to separate words in longer identifiers. It is generally easier to read `$var_names_like_this` than

$VarNamesLikeThis, especially for non-native speakers of English. It's also a simple rule that works consistently with VAR_NAMES_LIKE_THIS.

Package names are sometimes an exception to this rule. Perl informally reserves lowercase module names for "pragma" modules like integer and strict. Other modules should begin with a capital letter and use mixed case, but probably without underscores due to limitations in primitive file systems' representations of module names as files that must fit into a few sparse bytes.

- You may find it helpful to use letter case to indicate the scope or nature of a variable. For example:

  ```
  $ALL_CAPS_HERE   constants only (beware clashes with perl vars!)
  $Some_Caps_Here  package-wide global/static
  $no_caps_here    function scope my() or local() variables
  ```

  Function and method names seem to work best as all lowercase. E.g., $obj->as_string().

  You can use a leading underscore to indicate that a variable or function should not be used outside the package that defined it.

- If you have a really hairy regular expression, use the /x modifier and put in some whitespace to make it look a little less like line noise. Don't use slash as a delimiter when your regexp has slashes or backslashes.
- Use the new and and or operators to avoid having to parenthesize list operators so much, and to reduce the incidence of punctuation operators like && and ||. Call your subroutines as if they were functions or list operators to avoid excessive ampersands and parentheses.
- Use here documents instead of repeated print() statements.
- Line up corresponding things vertically, especially if it'd be too long to fit on one line anyway.

  ```
  $IDX = $ST_MTIME;
  $IDX = $ST_ATIME       if $opt_u;
  $IDX = $ST_CTIME       if $opt_c;
  $IDX = $ST_SIZE        if $opt_s;

  mkdir $tmpdir, 0700 or die "can't mkdir $tmpdir: $!";
  chdir($tmpdir)      or die "can't chdir $tmpdir: $!";
  mkdir 'tmp',   0777 or die "can't mkdir $tmpdir/tmp: $!";
  ```

- Always check the return codes of system calls. Good error messages should go to STDERR, include which program caused the problem, what the failed system call and arguments were, and (VERY IMPORTANT) should contain the standard system error message for what went wrong. Here's a simple but sufficient example:

  ```
  opendir(D, $dir)     or die "can't opendir $dir: $!";
  ```

- Line up your transliterations when it makes sense:

  ```
  tr [abc]
     [xyz];
  ```

- Think about reusability. Why waste brainpower on a one-shot when you might want to do something like it again? Consider generalizing your code. Consider writing a module or object class. Consider making your code run cleanly with `use strict` and `use warnings` (or `-w`) in effect. Consider giving away your code. Consider changing your whole world view. Consider... oh, never mind.

- Try to document your code and use Pod formatting in a consistent way. Here are commonly expected conventions:
  - use `C<>` for function, variable and module names (and more generally anything that can be considered part of code, like filehandles or specific values). Note that function names are considered more readable with parentheses after their name, that is `function()`.
  - use `B<>` for commands names like **cat** or **grep**.
  - use `F<>` or `C<>` for file names. `F<>` should be the only Pod code for file names, but as most Pod formatters render it as italic, Unix and Windows paths with their slashes and backslashes may be less readable, and better rendered with `C<>`.
- Be consistent.
- Be nice.

73 perlsub

73.1 NAME

perlsub - Perl subroutines

73.2 SYNOPSIS

To declare subroutines:

```
sub NAME;                     # A "forward" declaration.
sub NAME(PROTO);              #  ditto, but with prototypes
sub NAME : ATTRS;             #  with attributes
sub NAME(PROTO) : ATTRS;      #  with attributes and prototypes

sub NAME BLOCK                # A declaration and a definition.
sub NAME(PROTO) BLOCK         #  ditto, but with prototypes
sub NAME(SIG) BLOCK           #  with a signature instead
sub NAME : ATTRS BLOCK        #  with attributes
sub NAME(PROTO) : ATTRS BLOCK #  with prototypes and attributes
sub NAME(SIG) : ATTRS BLOCK   #  with a signature and attributes
```

To define an anonymous subroutine at runtime:

```
$subref = sub BLOCK;                 # no proto
$subref = sub (PROTO) BLOCK;         # with proto
$subref = sub (SIG) BLOCK;           # with signature
$subref = sub : ATTRS BLOCK;         # with attributes
$subref = sub (PROTO) : ATTRS BLOCK; # with proto and attributes
$subref = sub (SIG) : ATTRS BLOCK;   # with signature and attributes
```

To import subroutines:

```
use MODULE qw(NAME1 NAME2 NAME3);
```

To call subroutines:

```
NAME(LIST);    # & is optional with parentheses.
NAME LIST;     # Parentheses optional if predeclared/imported.
&NAME(LIST);   # Circumvent prototypes.
&NAME;         # Makes current @_ visible to called subroutine.
```

73.3 DESCRIPTION

Like many languages, Perl provides for user-defined subroutines. These may be located anywhere in the main program, loaded in from other files via the `do`, `require`, or `use` keywords, or generated on the fly using `eval` or anonymous subroutines. You can even call a function indirectly using a variable containing its name or a CODE reference.

The Perl model for function call and return values is simple: all functions are passed as parameters one single flat list of scalars, and all functions likewise return to their caller one single flat list of scalars. Any arrays or hashes in these call and return lists will collapse, losing their identities–but you may always use pass-by-reference instead to avoid this. Both call and return lists may contain as many or as few scalar elements as you'd like. (Often a function without an explicit return statement is called a subroutine, but there's really no difference from Perl's perspective.)

Any arguments passed in show up in the array `@_`. (They may also show up in lexical variables introduced by a signature; see Section 73.3.1 [Signatures], page 1220 below.) Therefore, if you called a function with two arguments, those would be stored in `$_[0]` and `$_[1]`. The array `@_` is a local array, but its elements are aliases for the actual scalar parameters. In particular, if an element `$_[0]` is updated, the corresponding argument is updated (or an error occurs if it is not updatable). If an argument is an array or hash element which did not exist when the function was called, that element is created only when (and if) it is modified or a reference to it is taken. (Some earlier versions of Perl created the element whether or not the element was assigned to.) Assigning to the whole array `@_` removes that aliasing, and does not update any arguments.

A `return` statement may be used to exit a subroutine, optionally specifying the returned value, which will be evaluated in the appropriate context (list, scalar, or void) depending on the context of the subroutine call. If you specify no return value, the subroutine returns an empty list in list context, the undefined value in scalar context, or nothing in void context. If you return one or more aggregates (arrays and hashes), these will be flattened together into one large indistinguishable list.

If no `return` is found and if the last statement is an expression, its value is returned. If the last statement is a loop control structure like a `foreach` or a `while`, the returned value is unspecified. The empty sub returns the empty list.

Aside from an experimental facility (see Section 73.3.1 [Signatures], page 1220 below), Perl does not have named formal parameters. In practice all you do is assign to a `my()` list of these. Variables that aren't declared to be private are global variables. For gory details on creating private variables, see Section 73.3.2 [Private Variables via my()], page 1223 and Section 73.3.4 [Temporary Values via local()], page 1228. To create protected environments for a set of functions in a separate package (and probably a separate file), see Section 40.2.2 [perlmod Packages], page 732.

Example:

```
sub max {
    my $max = shift(@_);
    foreach $foo (@_) {
        $max = $foo if $max < $foo;
    }
    return $max;
}
$bestday = max($mon,$tue,$wed,$thu,$fri);
```

Example:

```
# get a line, combining continuation lines
#  that start with whitespace

sub get_line {
    $thisline = $lookahead;  # global variables!
    LINE: while (defined($lookahead = <STDIN>)) {
        if ($lookahead =~ /^[ \t]/) {
            $thisline .= $lookahead;
        }
        else {
            last LINE;
        }
    }
    return $thisline;
}

$lookahead = <STDIN>;       # get first line
while (defined($line = get_line())) {
    ...
}
```

Assigning to a list of private variables to name your arguments:

```
sub maybeset {
    my($key, $value) = @_;
    $Foo{$key} = $value unless $Foo{$key};
}
```

Because the assignment copies the values, this also has the effect of turning call-by-reference into call-by-value. Otherwise a function is free to do in-place modifications of `@_` and change its caller's values.

```
upcase_in($v1, $v2);  # this changes $v1 and $v2
sub upcase_in {
    for (@_) { tr/a-z/A-Z/ }
}
```

You aren't allowed to modify constants in this way, of course. If an argument were actually literal and you tried to change it, you'd take a (presumably fatal) exception. For example, this won't work:

```
upcase_in("frederick");
```

It would be much safer if the `upcase_in()` function were written to return a copy of its parameters instead of changing them in place:

```
($v3, $v4) = upcase($v1, $v2);  # this doesn't change $v1 and $v2
sub upcase {
    return unless defined wantarray;  # void context, do nothing
    my @parms = @_;
    for (@parms) { tr/a-z/A-Z/ }
    return wantarray ? @parms : $parms[0];
}
```

Notice how this (unprototyped) function doesn't care whether it was passed real scalars or arrays. Perl sees all arguments as one big, long, flat parameter list in `@_`. This is one area where Perl's simple argument-passing style shines. The `upcase()` function would work perfectly well without changing the `upcase()` definition even if we fed it things like this:

```
@newlist   = upcase(@list1, @list2);
@newlist   = upcase( split /:/, $var );
```

Do not, however, be tempted to do this:

```
(@a, @b)   = upcase(@list1, @list2);
```

Like the flattened incoming parameter list, the return list is also flattened on return. So all you have managed to do here is stored everything in @a and made @b empty. See Section 73.3.9 [Pass by Reference], page 1236 for alternatives.

A subroutine may be called using an explicit & prefix. The & is optional in modern Perl, as are parentheses if the subroutine has been predeclared. The & is *not* optional when just naming the subroutine, such as when it's used as an argument to defined() or undef(). Nor is it optional when you want to do an indirect subroutine call with a subroutine name or reference using the &$subref() or &{$subref}() constructs, although the $subref->() notation solves that problem. See Section 62.1 [perlref NAME], page 1077 for more about all that.

Subroutines may be called recursively. If a subroutine is called using the & form, the argument list is optional, and if omitted, no @_ array is set up for the subroutine: the @_ array at the time of the call is visible to subroutine instead. This is an efficiency mechanism that new users may wish to avoid.

```
&foo(1,2,3);        # pass three arguments
foo(1,2,3);         # the same

foo();              # pass a null list
&foo();             # the same

&foo;               # foo() get current args, like foo(@_) !!
foo;                # like foo() IFF sub foo predeclared, else "foo"
```

Not only does the & form make the argument list optional, it also disables any prototype checking on arguments you do provide. This is partly for historical reasons, and partly for having a convenient way to cheat if you know what you're doing. See Section 73.3.10 [Prototypes], page 1238 below.

Since Perl 5.16.0, the __SUB__ token is available under use feature 'current_sub' and use 5.16.0. It will evaluate to a reference to the currently-running sub, which allows for recursive calls without knowing your subroutine's name.

```
use 5.16.0;
my $factorial = sub {
  my ($x) = @_;
  return 1 if $x == 1;
  return($x * __SUB__->( $x - 1 ) );
};
```

The behavior of __SUB__ within a regex code block (such as /(?{...})/) is subject to change.

Subroutines whose names are in all upper case are reserved to the Perl core, as are modules whose names are in all lower case. A subroutine in all capitals is a loosely-held convention meaning it will be called indirectly by the run-time system itself, usually due to a triggered event. Subroutines whose name start with a left parenthesis are also reserved the same way. The following is a list of some subroutines that currently do special, pre-defined things.

documented later in this document

`AUTOLOAD`

documented in Section 40.1 [perlmod NAME], page 732

`CLONE`, `CLONE_SKIP`,

documented in Section 46.1 [perlobj NAME], page 769

`DESTROY`

documented in Section 76.1 [perltie NAME], page 1289

`BINMODE`, `CLEAR`, `CLOSE`, `DELETE`, `DESTROY`, `EOF`, `EXISTS`, `EXTEND`, `FETCH`, `FETCHSIZE`, `FILENO`, `FIRSTKEY`, `GETC`, `NEXTKEY`, `OPEN`, `POP`, `PRINT`, `PRINTF`, `PUSH`, `READ`, `READLINE`, `SCALAR`, `SEEK`, `SHIFT`, `SPLICE`, `STORE`, `STORESIZE`, `TELL`, `TIEARRAY`, `TIEHANDLE`, `TIEHASH`, `TIESCALAR`, `UNSHIFT`, `UNTIE`, `WRITE`

documented in `PerlIO-via`

`BINMODE`, `CLEARERR`, `CLOSE`, `EOF`, `ERROR`, `FDOPEN`, `FILENO`, `FILL`, `FLUSH`, `OPEN`, `POPPED`, `PUSHED`, `READ`, `SEEK`, `SETLINEBUF`, `SYSOPEN`, `TELL`, `UNREAD`, `UTF8`, `WRITE`

documented in Section 25.1 [perlfunc NAME], page 351

Section “use” in `_perlfunc`, Section “use” in `_perlfunc`, Section “require” in `_perlfunc`

documented in `UNIVERSAL`

`VERSION`

documented in Section 13.1 [perldebguts NAME], page 90

`DB::DB`, `DB::sub`, `DB::lsub`, `DB::goto`, `DB::postponed`

undocumented, used internally by the `overload` feature

any starting with `(`

The `BEGIN`, `UNITCHECK`, `CHECK`, `INIT` and `END` subroutines are not so much subroutines as named special code blocks, of which you can have more than one in a package, and which you can **not** call explicitly. See Section 40.2.4 [perlmod BEGIN, UNITCHECK, CHECK, INIT and END], page 736

73.3.1 Signatures

WARNING: Subroutine signatures are experimental. The feature may be modified or removed in future versions of Perl.

Perl has an experimental facility to allow a subroutine's formal parameters to be introduced by special syntax, separate from the procedural code of the subroutine body. The formal parameter list is known as a *signature*. The facility must be enabled first by a pragmatic declaration, `use feature 'signatures'`, and it will produce a warning unless the "experimental::signatures" warnings category is disabled.

The signature is part of a subroutine's body. Normally the body of a subroutine is simply a braced block of code. When using a signature, the signature is a parenthesised list that goes immediately after the subroutine name. The signature declares lexical variables that are in scope for the block. When the subroutine is called, the signature takes control first. It populates the signature variables from the list of arguments that were passed. If the argument list doesn't meet the requirements of the signature, then it will throw an exception. When the signature processing is complete, control passes to the block.

Positional parameters are handled by simply naming scalar variables in the signature. For example,

```
sub foo ($left, $right) {
    return $left + $right;
}
```

takes two positional parameters, which must be filled at runtime by two arguments. By default the parameters are mandatory, and it is not permitted to pass more arguments than expected. So the above is equivalent to

```
sub foo {
    die "Too many arguments for subroutine" unless @_ <= 2;
    die "Too few arguments for subroutine" unless @_ >= 2;
    my $left = $_[0];
    my $right = $_[1];
    return $left + $right;
}
```

An argument can be ignored by omitting the main part of the name from a parameter declaration, leaving just a bare $ sigil. For example,

```
sub foo ($first, $, $third) {
    return "first=$first, third=$third";
}
```

Although the ignored argument doesn't go into a variable, it is still mandatory for the caller to pass it.

A positional parameter is made optional by giving a default value, separated from the parameter name by =:

```
sub foo ($left, $right = 0) {
    return $left + $right;
}
```

The above subroutine may be called with either one or two arguments. The default value expression is evaluated when the subroutine is called, so it may provide different default values for different calls. It is only evaluated if the argument was actually omitted from the call. For example,

```
my $auto_id = 0;
sub foo ($thing, $id = $auto_id++) {
    print "$thing has ID $id";
}
```

automatically assigns distinct sequential IDs to things for which no ID was supplied by the caller. A default value expression may also refer to parameters earlier in the signature, making the default for one parameter vary according to the earlier parameters. For example,

```
sub foo ($first_name, $surname, $nickname = $first_name) {
    print "$first_name $surname is known as \"$nickname\"";
}
```

An optional parameter can be nameless just like a mandatory parameter. For example,

```
sub foo ($thing, $ = 1) {
    print $thing;
}
```

The parameter's default value will still be evaluated if the corresponding argument isn't supplied, even though the value won't be stored anywhere. This is in case evaluating it has important side effects. However, it will be evaluated in void context, so if it doesn't have side effects and is not trivial it will generate a warning if the "void" warning category is enabled. If a nameless optional parameter's default value is not important, it may be omitted just as the parameter's name was:

```
sub foo ($thing, $=) {
    print $thing;
}
```

Optional positional parameters must come after all mandatory positional parameters. (If there are no mandatory positional parameters then an optional positional parameters can be the first thing in the signature.) If there are multiple optional positional parameters and not enough arguments are supplied to fill them all, they will be filled from left to right.

After positional parameters, additional arguments may be captured in a slurpy parameter. The simplest form of this is just an array variable:

```
sub foo ($filter, @inputs) {
    print $filter->($_) foreach @inputs;
}
```

With a slurpy parameter in the signature, there is no upper limit on how many arguments may be passed. A slurpy array parameter may be nameless just like a positional parameter, in which case its only effect is to turn off the argument limit that would otherwise apply:

```
sub foo ($thing, @) {
    print $thing;
}
```

A slurpy parameter may instead be a hash, in which case the arguments available to it are interpreted as alternating keys and values. There must be as many keys as values: if there is an odd argument then an exception will be thrown. Keys will be stringified, and if there are duplicates then the later instance takes precedence over the earlier, as with standard hash construction.

```
sub foo ($filter, %inputs) {
    print $filter->($_, $inputs{$_}) foreach sort keys %inputs;
}
```

A slurpy hash parameter may be nameless just like other kinds of parameter. It still insists that the number of arguments available to it be even, even though they're not being put into a variable.

```
sub foo ($thing, %) {
    print $thing;
}
```

A slurpy parameter, either array or hash, must be the last thing in the signature. It may follow mandatory and optional positional parameters; it may also be the only thing in the signature. Slurpy parameters cannot have default values: if no arguments are supplied for them then you get an empty array or empty hash.

A signature may be entirely empty, in which case all it does is check that the caller passed no arguments:

```
sub foo () {
    return 123;
}
```

When using a signature, the arguments are still available in the special array variable `@_`, in addition to the lexical variables of the signature. There is a difference between the two ways of accessing the arguments: `@_` *aliases* the arguments, but the signature variables get *copies* of the arguments. So writing to a signature variable only changes that variable, and has no effect on the caller's variables, but writing to an element of `@_` modifies whatever the caller used to supply that argument.

There is a potential syntactic ambiguity between signatures and prototypes (see Section 73.3.10 [Prototypes], page 1238), because both start with an opening parenthesis and both can appear in some of the same places, such as just after the name in a subroutine declaration. For historical reasons, when signatures are not enabled, any opening parenthesis in such a context will trigger very forgiving prototype parsing. Most signatures will be interpreted as prototypes in those circumstances, but won't be valid prototypes. (A valid prototype cannot contain any alphabetic character.) This will lead to somewhat confusing error messages.

To avoid ambiguity, when signatures are enabled the special syntax for prototypes is disabled. There is no attempt to guess whether a parenthesised group was intended to be a prototype or a signature. To give a subroutine a prototype under these circumstances, use a Section "Built-in Attributes" in `attributes`. For example,

```
sub foo :prototype($) { $_[0] }
```

It is entirely possible for a subroutine to have both a prototype and a signature. They do different jobs: the prototype affects compilation of calls to the subroutine, and the signature puts argument values into lexical variables at runtime. You can therefore write

```
sub foo ($left, $right) : prototype($$) {
    return $left + $right;
}
```

The prototype attribute, and any other attributes, come after the signature.

73.3.2 Private Variables via my()

Synopsis:

```
my $foo;            # declare $foo lexically local
my (@wid, %get);    # declare list of variables local
my $foo = "flurp";  # declare $foo lexical, and init it
my @oof = @bar;     # declare @oof lexical, and init it
my $x : Foo = $y;   # similar, with an attribute applied
```

WARNING: The use of attribute lists on `my` declarations is still evolving. The current semantics and interface are subject to change. See `attributes` and `Attribute-Handlers`.

The `my` operator declares the listed variables to be lexically confined to the enclosing block, conditional (`if/unless/elsif/else`), loop (`for/foreach/while/until/continue`), subroutine, `eval`, or `do/require/use`'d file. If more than one value is listed, the list must be placed in parentheses. All listed elements must be legal lvalues. Only alphanumeric identifiers may be lexically scoped–magical built-ins like `$/` must currently be `localized` with `local` instead.

Unlike dynamic variables created by the `local` operator, lexical variables declared with `my` are totally hidden from the outside world, including any called subroutines. This is true if it's the same subroutine called from itself or elsewhere–every call gets its own copy.

This doesn't mean that a `my` variable declared in a statically enclosing lexical scope would be invisible. Only dynamic scopes are cut off. For example, the `bumpx()` function below has access to the lexical $x variable because both the `my` and the `sub` occurred at the same scope, presumably file scope.

```
my $x = 10;
sub bumpx { $x++ }
```

An `eval()`, however, can see lexical variables of the scope it is being evaluated in, so long as the names aren't hidden by declarations within the `eval()` itself. See Section 62.1 [perlref NAME], page 1077.

The parameter list to my() may be assigned to if desired, which allows you to initialize your variables. (If no initializer is given for a particular variable, it is created with the undefined value.) Commonly this is used to name input parameters to a subroutine. Examples:

```
    $arg = "fred";        # "global" variable
    $n = cube_root(27);
    print "$arg thinks the root is $n\n";
fred thinks the root is 3

    sub cube_root {
        my $arg = shift;  # name doesn't matter
        $arg **= 1/3;
        return $arg;
    }
```

The `my` is simply a modifier on something you might assign to. So when you do assign to variables in its argument list, `my` doesn't change whether those variables are viewed as a scalar or an array. So

```
my ($foo) = <STDIN>;                # WRONG?
my @FOO = <STDIN>;
```

both supply a list context to the right-hand side, while

```
my $foo = <STDIN>;
```

supplies a scalar context. But the following declares only one variable:

```
my $foo, $bar = 1;                  # WRONG
```

That has the same effect as

```
my $foo;
$bar = 1;
```

The declared variable is not introduced (is not visible) until after the current statement. Thus,

```
my $x = $x;
```

can be used to initialize a new $x with the value of the old $x, and the expression

```
my $x = 123 and $x == 123
```

is false unless the old $x happened to have the value 123.

Lexical scopes of control structures are not bounded precisely by the braces that delimit their controlled blocks; control expressions are part of that scope, too. Thus in the loop

```
while (my $line = <>) {
    $line = lc $line;
} continue {
    print $line;
}
```

the scope of $line extends from its declaration throughout the rest of the loop construct (including the continue clause), but not beyond it. Similarly, in the conditional

```
if ((my $answer = <STDIN>) =~ /^yes$/i) {
    user_agrees();
} elsif ($answer =~ /^no$/i) {
    user_disagrees();
} else {
    chomp $answer;
    die "'$answer' is neither 'yes' nor 'no'";
}
```

the scope of $answer extends from its declaration through the rest of that conditional, including any elsif and else clauses, but not beyond it. See Section 74.2.3 [perlsyn Simple Statements], page 1250 for information on the scope of variables in statements with modifiers.

The foreach loop defaults to scoping its index variable dynamically in the manner of local. However, if the index variable is prefixed with the keyword my, or if there is already a lexical by that name in scope, then a new lexical is created instead. Thus in the loop

```
for my $i (1, 2, 3) {
    some_function();
}
```

the scope of $i extends to the end of the loop, but not beyond it, rendering the value of $i inaccessible within some_function().

Some users may wish to encourage the use of lexically scoped variables. As an aid to catching implicit uses to package variables, which are always global, if you say

```
use strict 'vars';
```

then any variable mentioned from there to the end of the enclosing block must either refer to a lexical variable, be predeclared via our or use vars, or else must be fully qualified with the package name. A compilation error results otherwise. An inner block may countermand this with no strict 'vars'.

A my has both a compile-time and a run-time effect. At compile time, the compiler takes notice of it. The principal usefulness of this is to quiet use strict 'vars', but it is also essential for generation of closures as detailed in Section 62.1 [perlref NAME], page 1077.

Actual initialization is delayed until run time, though, so it gets executed at the appropriate time, such as each time through a loop, for example.

Variables declared with `my` are not part of any package and are therefore never fully qualified with the package name. In particular, you're not allowed to try to make a package variable (or other global) lexical:

```
my $pack::var;      # ERROR!  Illegal syntax
```

In fact, a dynamic variable (also known as package or global variables) are still accessible using the fully qualified `::` notation even while a lexical of the same name is also visible:

```
package main;
local $x = 10;
my    $x = 20;
print "$x and $::x\n";
```

That will print out 20 and 10.

You may declare `my` variables at the outermost scope of a file to hide any such identifiers from the world outside that file. This is similar in spirit to C's static variables when they are used at the file level. To do this with a subroutine requires the use of a closure (an anonymous function that accesses enclosing lexicals). If you want to create a private subroutine that cannot be called from outside that block, it can declare a lexical variable containing an anonymous sub reference:

```
my $secret_version = '1.001-beta';
my $secret_sub = sub { print $secret_version };
&$secret_sub();
```

As long as the reference is never returned by any function within the module, no outside module can see the subroutine, because its name is not in any package's symbol table. Remember that it's not *REALLY* called `$some_pack::secret_version` or anything; it's just $secret_version, unqualified and unqualifiable.

This does not work with object methods, however; all object methods have to be in the symbol table of some package to be found. See Section 62.3.7 [perlref Function Templates], page 1086 for something of a work-around to this.

73.3.3 Persistent Private Variables

There are two ways to build persistent private variables in Perl 5.10. First, you can simply use the `state` feature. Or, you can use closures, if you want to stay compatible with releases older than 5.10.

73.3.3.1 Persistent variables via state()

Beginning with Perl 5.10.0, you can declare variables with the `state` keyword in place of `my`. For that to work, though, you must have enabled that feature beforehand, either by using the `feature` pragma, or by using `-E` on one-liners (see `feature`). Beginning with Perl 5.16, the `CORE::state` form does not require the `feature` pragma.

The `state` keyword creates a lexical variable (following the same scoping rules as `my`) that persists from one subroutine call to the next. If a state variable resides inside an anonymous subroutine, then each copy of the subroutine has its own copy of the state variable. However, the value of the state variable will still persist between calls to the same copy of the anonymous subroutine. (Don't forget that `sub { ... }` creates a new subroutine each time it is executed.)

For example, the following code maintains a private counter, incremented each time the gimme_another() function is called:

```
use feature 'state';
sub gimme_another { state $x; return ++$x }
```

And this example uses anonymous subroutines to create separate counters:

```
use feature 'state';
sub create_counter {
    return sub { state $x; return ++$x }
}
```

Also, since `$x` is lexical, it can't be reached or modified by any Perl code outside.

When combined with variable declaration, simple scalar assignment to `state` variables (as in `state $x = 42`) is executed only the first time. When such statements are evaluated subsequent times, the assignment is ignored. The behavior of this sort of assignment to non-scalar variables is undefined.

73.3.3.2 Persistent variables with closures

Just because a lexical variable is lexically (also called statically) scoped to its enclosing block, `eval`, or `do` FILE, this doesn't mean that within a function it works like a C static. It normally works more like a C auto, but with implicit garbage collection.

Unlike local variables in C or C++, Perl's lexical variables don't necessarily get recycled just because their scope has exited. If something more permanent is still aware of the lexical, it will stick around. So long as something else references a lexical, that lexical won't be freed–which is as it should be. You wouldn't want memory being free until you were done using it, or kept around once you were done. Automatic garbage collection takes care of this for you.

This means that you can pass back or save away references to lexical variables, whereas to return a pointer to a C auto is a grave error. It also gives us a way to simulate C's function statics. Here's a mechanism for giving a function private variables with both lexical scoping and a static lifetime. If you do want to create something like C's static variables, just enclose the whole function in an extra block, and put the static variable outside the function but in the block.

```
{
    my $secret_val = 0;
    sub gimme_another {
        return ++$secret_val;
    }
}
# $secret_val now becomes unreachable by the outside
# world, but retains its value between calls to gimme_another
```

If this function is being sourced in from a separate file via `require` or `use`, then this is probably just fine. If it's all in the main program, you'll need to arrange for the `my` to be executed early, either by putting the whole block above your main program, or more likely, placing merely a BEGIN code block around it to make sure it gets executed before your program starts to run:

```
BEGIN {
    my $secret_val = 0;
    sub gimme_another {
        return ++$secret_val;
    }
}
```

See Section 40.2.4 [perlmod BEGIN, UNITCHECK, CHECK, INIT and END], page 736 about the special triggered code blocks, BEGIN, UNITCHECK, CHECK, INIT and END.

If declared at the outermost scope (the file scope), then lexicals work somewhat like C's file statics. They are available to all functions in that same file declared below them, but are inaccessible from outside that file. This strategy is sometimes used in modules to create private variables that the whole module can see.

73.3.4 Temporary Values via local()

WARNING: In general, you should be using my instead of local, because it's faster and safer. Exceptions to this include the global punctuation variables, global filehandles and formats, and direct manipulation of the Perl symbol table itself. local is mostly used when the current value of a variable must be visible to called subroutines.

Synopsis:

```
# localization of values

local $foo;                # make $foo dynamically local
local (@wid, %get);        # make list of variables local
local $foo = "flurp";      # make $foo dynamic, and init it
local @oof = @bar;         # make @oof dynamic, and init it

local $hash{key} = "val";  # sets a local value for this hash entry
delete local $hash{key};   # delete this entry for the current block
local ($cond ? $v1 : $v2); # several types of lvalues support
                           # localization

# localization of symbols

local *FH;                 # localize $FH, @FH, %FH, &FH  ...
local *merlyn = *randal;   # now $merlyn is really $randal, plus
                           #     @merlyn is really @randal, etc
local *merlyn = 'randal';  # SAME THING: promote 'randal' to *randal
local *merlyn = \$randal;  # just alias $merlyn, not @merlyn etc
```

A local modifies its listed variables to be "local" to the enclosing block, eval, or do FILE–and to *any subroutine called from within that block.* A local just gives temporary values to global (meaning package) variables. It does *not* create a local variable. This is known as dynamic scoping. Lexical scoping is done with my, which works more like C's auto declarations.

Some types of lvalues can be localized as well: hash and array elements and slices, conditionals (provided that their result is always localizable), and symbolic references. As for simple variables, this creates new, dynamically scoped values.

If more than one variable or expression is given to `local`, they must be placed in parentheses. This operator works by saving the current values of those variables in its argument list on a hidden stack and restoring them upon exiting the block, subroutine, or eval. This means that called subroutines can also reference the local variable, but not the global one. The argument list may be assigned to if desired, which allows you to initialize your local variables. (If no initializer is given for a particular variable, it is created with an undefined value.)

Because `local` is a run-time operator, it gets executed each time through a loop. Consequently, it's more efficient to localize your variables outside the loop.

73.3.4.1 Grammatical note on local()

A `local` is simply a modifier on an lvalue expression. When you assign to a `local`ized variable, the `local` doesn't change whether its list is viewed as a scalar or an array. So

```
local($foo) = <STDIN>;
local @FOO = <STDIN>;
```

both supply a list context to the right-hand side, while

```
local $foo = <STDIN>;
```

supplies a scalar context.

73.3.4.2 Localization of special variables

If you localize a special variable, you'll be giving a new value to it, but its magic won't go away. That means that all side-effects related to this magic still work with the localized value.

This feature allows code like this to work :

```
# Read the whole contents of FILE in $slurp
{ local $/ = undef; $slurp = <FILE>; }
```

Note, however, that this restricts localization of some values ; for example, the following statement dies, as of perl 5.10.0, with an error *Modification of a read-only value attempted*, because the $1 variable is magical and read-only :

```
local $1 = 2;
```

One exception is the default scalar variable: starting with perl 5.14 `local($_)` will always strip all magic from $_, to make it possible to safely reuse $_ in a subroutine.

WARNING: Localization of tied arrays and hashes does not currently work as described. This will be fixed in a future release of Perl; in the meantime, avoid code that relies on any particular behavior of localising tied arrays or hashes (localising individual elements is still okay). See Section "Localising Tied Arrays and Hashes Is Broken" in `perl58delta` for more details.

73.3.4.3 Localization of globs

The construct

```
local *name;
```

creates a whole new symbol table entry for the glob `name` in the current package. That means that all variables in its glob slot ($name, @name, %name, &name, and the `name` filehandle) are dynamically reset.

This implies, among other things, that any magic eventually carried by those variables is locally lost. In other words, saying `local */` will not have any effect on the internal value of the input record separator.

73.3.4.4 Localization of elements of composite types

It's also worth taking a moment to explain what happens when you `local`ize a member of a composite type (i.e. an array or hash element). In this case, the element is `local`ized *by name*. This means that when the scope of the `local()` ends, the saved value will be restored to the hash element whose key was named in the `local()`, or the array element whose index was named in the `local()`. If that element was deleted while the `local()` was in effect (e.g. by a `delete()` from a hash or a `shift()` of an array), it will spring back into existence, possibly extending an array and filling in the skipped elements with `undef`. For instance, if you say

```
%hash = ( 'This' => 'is', 'a' => 'test' );
@ary  = ( 0..5 );
{
     local($ary[5]) = 6;
     local($hash{'a'}) = 'drill';
     while (my $e = pop(@ary)) {
         print "$e . . .\n";
         last unless $e > 3;
     }
     if (@ary) {
         $hash{'only a'} = 'test';
         delete $hash{'a'};
     }
}
print join(' ', map { "$_ $hash{$_}" } sort keys %hash),".\n";
print "The array has ",scalar(@ary)," elements: ",
      join(', ', map { defined $_ ? $_ : 'undef' } @ary),"\n";
```

Perl will print

```
6 . . .
4 . . .
3 . . .
This is a test only a test.
The array has 6 elements: 0, 1, 2, undef, undef, 5
```

The behavior of local() on non-existent members of composite types is subject to change in future.

73.3.4.5 Localized deletion of elements of composite types

You can use the `delete local $array[$idx]` and `delete local $hash{key}` constructs to delete a composite type entry for the current block and restore it when it ends. They return the array/hash value before the localization, which means that they are respectively equivalent to

```
do {
    my $val = $array[$idx];
    local  $array[$idx];
    delete $array[$idx];
    $val
}
```

and

```
do {
    my $val = $hash{key};
    local  $hash{key};
    delete $hash{key};
    $val
}
```

except that for those the `local` is scoped to the `do` block. Slices are also accepted.

```
my %hash = (
 a => [ 7, 8, 9 ],
 b => 1,
)

{
 my $a = delete local $hash{a};
 # $a is [ 7, 8, 9 ]
 # %hash is (b => 1)

 {
  my @nums = delete local @$a[0, 2]
  # @nums is (7, 9)
  # $a is [ undef, 8 ]

  $a[0] = 999; # will be erased when the scope ends
 }
 # $a is back to [ 7, 8, 9 ]

}
# %hash is back to its original state
```

73.3.5 Lvalue subroutines

It is possible to return a modifiable value from a subroutine. To do this, you have to declare the subroutine to return an lvalue.

```
my $val;
sub canmod : lvalue {
    $val;  # or:  return $val;
}
```

```
sub nomod {
    $val;
}

canmod() = 5;   # assigns to $val
nomod()  = 5;   # ERROR
```

The scalar/list context for the subroutine and for the right-hand side of assignment is determined as if the subroutine call is replaced by a scalar. For example, consider:

```
data(2,3) = get_data(3,4);
```

Both subroutines here are called in a scalar context, while in:

```
(data(2,3)) = get_data(3,4);
```

and in:

```
(data(2),data(3)) = get_data(3,4);
```

all the subroutines are called in a list context.

Lvalue subroutines are convenient, but you have to keep in mind that, when used with objects, they may violate encapsulation. A normal mutator can check the supplied argument before setting the attribute it is protecting, an lvalue subroutine cannot. If you require any special processing when storing and retrieving the values, consider using the CPAN module Sentinel or something similar.

73.3.6 Lexical Subroutines

WARNING: Lexical subroutines are still experimental. The feature may be modified or removed in future versions of Perl.

Lexical subroutines are only available under the `use feature 'lexical_subs'` pragma, which produces a warning unless the "experimental::lexical_subs" warnings category is disabled.

Beginning with Perl 5.18, you can declare a private subroutine with `my` or `state`. As with state variables, the `state` keyword is only available under `use feature 'state'` or `use 5.010` or higher.

These subroutines are only visible within the block in which they are declared, and only after that declaration:

```
no warnings "experimental::lexical_subs";
use feature 'lexical_subs';

foo();              # calls the package/global subroutine
state sub foo {
    foo();          # also calls the package subroutine
}
foo();              # calls "state" sub
my $ref = \&foo;    # take a reference to "state" sub

my sub bar { ... }
bar();              # calls "my" sub
```

To use a lexical subroutine from inside the subroutine itself, you must predeclare it. The `sub foo {...}` subroutine definition syntax respects any previous `my sub;` or `state sub;` declaration.

```
my sub baz;         # predeclaration
sub baz {           # define the "my" sub
    baz();          # recursive call
}
```

73.3.6.1 state sub vs my sub

What is the difference between "state" subs and "my" subs? Each time that execution enters a block when "my" subs are declared, a new copy of each sub is created. "State" subroutines persist from one execution of the containing block to the next.

So, in general, "state" subroutines are faster. But "my" subs are necessary if you want to create closures:

```
no warnings "experimental::lexical_subs";
use feature 'lexical_subs';

sub whatever {
    my $x = shift;
    my sub inner {
        ... do something with $x ...
    }
    inner();
}
```

In this example, a new `$x` is created when `whatever` is called, and also a new `inner`, which can see the new `$x`. A "state" sub will only see the `$x` from the first call to `whatever`.

73.3.6.2 our subroutines

Like `our $variable`, `our sub` creates a lexical alias to the package subroutine of the same name.

The two main uses for this are to switch back to using the package sub inside an inner scope:

```
no warnings "experimental::lexical_subs";
use feature 'lexical_subs';

sub foo { ... }

sub bar {
    my sub foo { ... }
    {
        # need to use the outer foo here
        our sub foo;
        foo();
    }
}
```

and to make a subroutine visible to other packages in the same scope:

```
package MySneakyModule;

no warnings "experimental::lexical_subs";
use feature 'lexical_subs';

our sub do_something { ... }

sub do_something_with_caller {
    package DB;
    () = caller 1;          # sets @DB::args
    do_something(@args);    # uses MySneakyModule::do_something
}
```

73.3.7 Passing Symbol Table Entries (typeglobs)

WARNING: The mechanism described in this section was originally the only way to simulate pass-by-reference in older versions of Perl. While it still works fine in modern versions, the new reference mechanism is generally easier to work with. See below.

Sometimes you don't want to pass the value of an array to a subroutine but rather the name of it, so that the subroutine can modify the global copy of it rather than working with a local copy. In perl you can refer to all objects of a particular name by prefixing the name with a star: `*foo`. This is often known as a "typeglob", because the star on the front can be thought of as a wildcard match for all the funny prefix characters on variables and subroutines and such.

When evaluated, the typeglob produces a scalar value that represents all the objects of that name, including any filehandle, format, or subroutine. When assigned to, it causes the name mentioned to refer to whatever `*` value was assigned to it. Example:

```
sub doubleary {
    local(*someary) = @_;
    foreach $elem (@someary) {
        $elem *= 2;
    }
}
doubleary(*foo);
doubleary(*bar);
```

Scalars are already passed by reference, so you can modify scalar arguments without using this mechanism by referring explicitly to `$_[0]` etc. You can modify all the elements of an array by passing all the elements as scalars, but you have to use the `*` mechanism (or the equivalent reference mechanism) to `push`, `pop`, or change the size of an array. It will certainly be faster to pass the typeglob (or reference).

Even if you don't want to modify an array, this mechanism is useful for passing multiple arrays in a single LIST, because normally the LIST mechanism will merge all the array values so that you can't extract out the individual arrays. For more on typeglobs, see Section 11.2.10 [perldata Typeglobs and Filehandles], page 85.

73.3.8 When to Still Use local()

Despite the existence of my, there are still three places where the local operator still shines. In fact, in these three places, you *must* use local instead of my.

1. You need to give a global variable a temporary value, especially $_.

   The global variables, like @ARGV or the punctuation variables, must be localized with local(). This block reads in /etc/motd, and splits it up into chunks separated by lines of equal signs, which are placed in @Fields.

   ```
   {
       local @ARGV = ("/etc/motd");
       local $/ = undef;
       local $_ = <>;
       @Fields = split /^\s*=+\s*$/;
   }
   ```

   It particular, it's important to localize $_ in any routine that assigns to it. Look out for implicit assignments in while conditionals.

2. You need to create a local file or directory handle or a local function.

   A function that needs a filehandle of its own must use local() on a complete typeglob. This can be used to create new symbol table entries:

   ```
   sub ioqueue {
       local  (*READER, *WRITER);    # not my!
       pipe    (READER,  WRITER)     or die "pipe: $!";
       return (*READER, *WRITER);
   }
   ($head, $tail) = ioqueue();
   ```

   See the Symbol module for a way to create anonymous symbol table entries.

   Because assignment of a reference to a typeglob creates an alias, this can be used to create what is effectively a local function, or at least, a local alias.

   ```
   {
       local *grow = \&shrink; # only until this block exits
       grow();                 # really calls shrink()
       move();                 # if move() grow()s, it shrink()s too
   }
   grow();                     # get the real grow() again
   ```

   See Section 62.3.7 [perlref Function Templates], page 1086 for more about manipulating functions by name in this way.

3. You want to temporarily change just one element of an array or hash.

   You can localize just one element of an aggregate. Usually this is done on dynamics:

   ```
   {
       local $SIG{INT} = 'IGNORE';
       funct();                            # uninterruptible
   }
   # interruptibility automatically restored here
   ```

   But it also works on lexically declared aggregates.

73.3.9 Pass by Reference

If you want to pass more than one array or hash into a function–or return them from it–and have them maintain their integrity, then you're going to have to use an explicit pass-by-reference. Before you do that, you need to understand references as detailed in Section 62.1 [perlref NAME], page 1077. This section may not make much sense to you otherwise.

Here are a few simple examples. First, let's pass in several arrays to a function and have it pop all of then, returning a new list of all their former last elements:

```
@tailings = popmany ( \@a, \@b, \@c, \@d );

sub popmany {
    my $aref;
    my @retlist = ();
    foreach $aref ( @_ ) {
        push @retlist, pop @$aref;
    }
    return @retlist;
}
```

Here's how you might write a function that returns a list of keys occurring in all the hashes passed to it:

```
@common = inter( \%foo, \%bar, \%joe );
sub inter {
    my ($k, $href, %seen); # locals
    foreach $href (@_) {
        while ( $k = each %$href ) {
            $seen{$k}++;
        }
    }
    return grep { $seen{$_} == @_ } keys %seen;
}
```

So far, we're using just the normal list return mechanism. What happens if you want to pass or return a hash? Well, if you're using only one of them, or you don't mind them concatenating, then the normal calling convention is ok, although a little expensive.

Where people get into trouble is here:

```
(@a, @b) = func(@c, @d);
```

or

```
(%a, %b) = func(%c, %d);
```

That syntax simply won't work. It sets just @a or %a and clears the @b or %b. Plus the function didn't get passed into two separate arrays or hashes: it got one long list in @_, as always.

If you can arrange for everyone to deal with this through references, it's cleaner code, although not so nice to look at. Here's a function that takes two array references as arguments, returning the two array elements in order of how many elements they have in them:

```
($aref, $bref) = func(\@c, \@d);
```

```
print "@$aref has more than @$bref\n";
sub func {
    my ($cref, $dref) = @_;
    if (@$cref > @$dref) {
        return ($cref, $dref);
    } else {
        return ($dref, $cref);
    }
}
```

It turns out that you can actually do this also:

```
(*a, *b) = func(\@c, \@d);
print "@a has more than @b\n";
sub func {
    local (*c, *d) = @_;
    if (@c > @d) {
        return (\@c, \@d);
    } else {
        return (\@d, \@c);
    }
}
```

Here we're using the typeglobs to do symbol table aliasing. It's a tad subtle, though, and also won't work if you're using `my` variables, because only globals (even in disguise as `local`s) are in the symbol table.

If you're passing around filehandles, you could usually just use the bare typeglob, like `*STDOUT`, but typeglobs references work, too. For example:

```
splutter(\*STDOUT);
sub splutter {
    my $fh = shift;
    print $fh "her um well a hmmm\n";
}

$rec = get_rec(\*STDIN);
sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

If you're planning on generating new filehandles, you could do this. Notice to pass back just the bare *FH, not its reference.

```
sub openit {
    my $path = shift;
    local *FH;
    return open (FH, $path) ? *FH : undef;
}
```

73.3.10 Prototypes

Perl supports a very limited kind of compile-time argument checking using function prototyping. This can be declared in either the PROTO section or with a Section "Built-in Attributes" in `attributes`. If you declare either of

```
sub mypush (+@)
sub mypush :prototype(+@)
```

then `mypush()` takes arguments exactly like `push()` does.

If subroutine signatures are enabled (see Section 73.3.1 [Signatures], page 1220), then the shorter PROTO syntax is unavailable, because it would clash with signatures. In that case, a prototype can only be declared in the form of an attribute.

The function declaration must be visible at compile time. The prototype affects only interpretation of new-style calls to the function, where new-style is defined as not using the `&` character. In other words, if you call it like a built-in function, then it behaves like a built-in function. If you call it like an old-fashioned subroutine, then it behaves like an old-fashioned subroutine. It naturally falls out from this rule that prototypes have no influence on subroutine references like `\&foo` or on indirect subroutine calls like `&{$subref}` or `$subref->()`.

Method calls are not influenced by prototypes either, because the function to be called is indeterminate at compile time, since the exact code called depends on inheritance.

Because the intent of this feature is primarily to let you define subroutines that work like built-in functions, here are prototypes for some other functions that parse almost exactly like the corresponding built-in.

```
Declared as             Called as

sub mylink ($$)         mylink $old, $new
sub myvec ($$$)         myvec $var, $offset, 1
sub myindex ($$;$)      myindex &getstring, "substr"
sub mysyswrite ($$$;$)  mysyswrite $buf, 0, length($buf) - $off, $off
sub myreverse (@)       myreverse $a, $b, $c
sub myjoin ($@)         myjoin ":", $a, $b, $c
sub mypop (+)           mypop @array
sub mysplice (+$$@)     mysplice @array, 0, 2, @pushme
sub mykeys (+)          mykeys %{$hashref}
sub myopen (*;$)        myopen HANDLE, $name
sub mypipe (**)         mypipe READHANDLE, WRITEHANDLE
sub mygrep (&@)         mygrep { /foo/ } $a, $b, $c
sub myrand (;$)         myrand 42
sub mytime ()           mytime
```

Any backslashed prototype character represents an actual argument that must start with that character (optionally preceded by `my`, `our` or `local`), with the exception of `$`, which will accept any scalar lvalue expression, such as `$foo = 7` or `my_function()->[0]`. The value passed as part of `@_` will be a reference to the actual argument given in the subroutine call, obtained by applying `\` to that argument.

You can use the `\[]` backslash group notation to specify more than one allowed argument type. For example:

```
sub myref (\[$@%&*])
```

will allow calling myref() as

```
myref $var
myref @array
myref %hash
myref &sub
myref *glob
```

and the first argument of myref() will be a reference to a scalar, an array, a hash, a code, or a glob.

Unbackslashed prototype characters have special meanings. Any unbackslashed `@` or `%` eats all remaining arguments, and forces list context. An argument represented by `$` forces scalar context. An `&` requires an anonymous subroutine, which, if passed as the first argument, does not require the `sub` keyword or a subsequent comma.

A `*` allows the subroutine to accept a bareword, constant, scalar expression, typeglob, or a reference to a typeglob in that slot. The value will be available to the subroutine either as a simple scalar, or (in the latter two cases) as a reference to the typeglob. If you wish to always convert such arguments to a typeglob reference, use Symbol::qualify_to_ref() as follows:

```
use Symbol 'qualify_to_ref';

sub foo (*) {
    my $fh = qualify_to_ref(shift, caller);
    ...
}
```

The `+` prototype is a special alternative to `$` that will act like `\[@%]` when given a literal array or hash variable, but will otherwise force scalar context on the argument. This is useful for functions which should accept either a literal array or an array reference as the argument:

```
sub mypush (+@) {
    my $aref = shift;
    die "Not an array or arrayref" unless ref $aref eq 'ARRAY';
    push @$aref, @_;
}
```

When using the `+` prototype, your function must check that the argument is of an acceptable type.

A semicolon (`;`) separates mandatory arguments from optional arguments. It is redundant before `@` or `%`, which gobble up everything else.

As the last character of a prototype, or just before a semicolon, a `@` or a `%`, you can use `_` in place of `$`: if this argument is not provided, `$_` will be used instead.

Note how the last three examples in the table above are treated specially by the parser. `mygrep()` is parsed as a true list operator, `myrand()` is parsed as a true unary operator with unary precedence the same as `rand()`, and `mytime()` is truly without arguments, just like `time()`. That is, if you say

```
mytime +2;
```

you'll get `mytime() + 2`, not `mytime(2)`, which is how it would be parsed without a prototype. If you want to force a unary function to have the same precedence as a list operator, add `;` to the end of the prototype:

```
sub mygetprotobynumber($;);
mygetprotobynumber $a > $b; # parsed as mygetprotobynumber($a > $b)
```

The interesting thing about `&` is that you can generate new syntax with it, provided it's in the initial position:

```
sub try (&@) {
    my($try,$catch) = @_;
    eval { &$try };
    if ($@) {
        local $_ = $@;
        &$catch;
    }
}
sub catch (&) { $_[0] }

try {
    die "phooey";
} catch {
    /phooey/ and print "unphooey\n";
};
```

That prints `"unphooey"`. (Yes, there are still unresolved issues having to do with visibility of `@_`. I'm ignoring that question for the moment. (But note that if we make `@_` lexically scoped, those anonymous subroutines can act like closures... (Gee, is this sounding a little Lispish? (Never mind.))))

And here's a reimplementation of the Perl `grep` operator:

```
sub mygrep (&@) {
    my $code = shift;
    my @result;
    foreach $_ (@_) {
        push(@result, $_) if &$code;
    }
    @result;
}
```

Some folks would prefer full alphanumeric prototypes. Alphanumerics have been intentionally left out of prototypes for the express purpose of someday in the future adding named, formal parameters. The current mechanism's main goal is to let module writers provide better diagnostics for module users. Larry feels the notation quite understandable to Perl programmers, and that it will not intrude greatly upon the meat of the module, nor make it harder to read. The line noise is visually encapsulated into a small pill that's easy to swallow.

If you try to use an alphanumeric sequence in a prototype you will generate an optional warning - "Illegal character in prototype...". Unfortunately earlier versions of Perl allowed the prototype to be used as long as its prefix was a valid prototype. The warning may be upgraded to a fatal error in a future version of Perl once the majority of offending code is fixed.

It's probably best to prototype new functions, not retrofit prototyping into older ones. That's because you must be especially careful about silent impositions of differing list versus scalar contexts. For example, if you decide that a function should take just one parameter, like this:

```
sub func ($) {
    my $n = shift;
    print "you gave me $n\n";
}
```

and someone has been calling it with an array or expression returning a list:

```
func(@foo);
func( split /:/ );
```

Then you've just supplied an automatic `scalar` in front of their argument, which can be more than a bit surprising. The old `@foo` which used to hold one thing doesn't get passed in. Instead, `func()` now gets passed in a `1`; that is, the number of elements in `@foo`. And the `split` gets called in scalar context so it starts scribbling on your `@_` parameter list. Ouch!

If a sub has both a PROTO and a BLOCK, the prototype is not applied until after the BLOCK is completely defined. This means that a recursive function with a prototype has to be predeclared for the prototype to take effect, like so:

```
    sub foo($$);
    sub foo($$) {
            foo 1, 2;
    }
```

This is all very powerful, of course, and should be used only in moderation to make the world a better place.

73.3.11 Constant Functions

Functions with a prototype of `()` are potential candidates for inlining. If the result after optimization and constant folding is either a constant or a lexically-scoped scalar which has no other references, then it will be used in place of function calls made without `&`. Calls made using `&` are never inlined. (See `constant.pm` for an easy way to declare most constants.)

The following functions would all be inlined:

```
sub pi ()           { 3.14159 }             # Not exact, but close.
sub PI ()           { 4 * atan2 1, 1 }      # As good as it gets,
                                            # and it's inlined, too!
sub ST_DEV ()       { 0 }
sub ST_INO ()       { 1 }

sub FLAG_FOO ()     { 1 << 8 }
sub FLAG_BAR ()     { 1 << 9 }
sub FLAG_MASK ()    { FLAG_FOO | FLAG_BAR }

sub OPT_BAZ ()      { not (0x1B58 & FLAG_MASK) }

sub N () { int(OPT_BAZ) / 3 }

sub FOO_SET () { 1 if FLAG_MASK & FLAG_FOO }
sub FOO_SET2 () { if (FLAG_MASK & FLAG_FOO) { 1 } }
```

(Be aware that the last example was not always inlined in Perl 5.20 and earlier, which did not behave consistently with subroutines containing inner scopes.) You can countermand inlining by using an explicit `return`:

```
sub baz_val () {
    if (OPT_BAZ) {
        return 23;
    }
    else {
        return 42;
    }
}
sub bonk_val () { return 12345 }
```

As alluded to earlier you can also declare inlined subs dynamically at BEGIN time if their body consists of a lexically-scoped scalar which has no other references. Only the first example here will be inlined:

```
BEGIN {
    my $var = 1;
    no strict 'refs';
    *INLINED = sub () { $var };
}

BEGIN {
    my $var = 1;
    my $ref = \$var;
    no strict 'refs';
    *NOT_INLINED = sub () { $var };
}
```

A not so obvious caveat with this (see [RT #79908]) is that the variable will be immediately inlined, and will stop behaving like a normal lexical variable, e.g. this will print 79907, not 79908:

```
BEGIN {
    my $x = 79907;
    *RT_79908 = sub () { $x };
    $x++;
}
print RT_79908(); # prints 79907
```

As of Perl 5.22, this buggy behavior, while preserved for backward compatibility, is detected and emits a deprecation warning. If you want the subroutine to be inlined (with no warning), make sure the variable is not used in a context where it could be modified aside from where it is declared.

```
# Fine, no warning
BEGIN {
    my $x = 54321;
    *INLINED = sub () { $x };
}
# Warns.  Future Perl versions will stop inlining it.
BEGIN {
    my $x;
    $x = 54321;
    *ALSO_INLINED = sub () { $x };
}
```

Perl 5.22 also introduces the experimental "const" attribute as an alternative. (Disable the "experimental::const_attr" warnings if you want to use it.) When applied to an anonymous subroutine, it forces the sub to be called when the `sub` expression is evaluated. The return value is captured and turned into a constant subroutine:

```
my $x = 54321;
*INLINED = sub : const { $x };
$x++;
```

The return value of `INLINED` in this example will always be 54321, regardless of later modifications to $x. You can also put any arbitrary code inside the sub, at it will be executed immediately and its return value captured the same way.

If you really want a subroutine with a `()` prototype that returns a lexical variable you can easily force it to not be inlined by adding an explicit `return`:

```
BEGIN {
    my $x = 79907;
    *RT_79908 = sub () { return $x };
    $x++;
}
print RT_79908(); # prints 79908
```

The easiest way to tell if a subroutine was inlined is by using `B-Deparse`. Consider this example of two subroutines returning `1`, one with a `()` prototype causing it to be inlined, and one without (with deparse output truncated for clarity):

```
$ perl -MO=Deparse -le 'sub ONE { 1 } if (ONE) { print ONE if ONE }'
sub ONE {
    1;
}
if (ONE ) {
    print ONE() if ONE ;
}
$ perl -MO=Deparse -le 'sub ONE () { 1 } if (ONE) { print ONE if ONE }'
sub ONE () { 1 }
```

```
do {
    print 1
};
```

If you redefine a subroutine that was eligible for inlining, you'll get a warning by default. You can use this warning to tell whether or not a particular subroutine is considered inlinable, since it's different than the warning for overriding non-inlined subroutines:

```
$ perl -e 'sub one () {1} sub one () {2}'
Constant subroutine one redefined at -e line 1.
$ perl -we 'sub one {1} sub one {2}'
Subroutine one redefined at -e line 1.
```

The warning is considered severe enough not to be affected by the **-w** switch (or its absence) because previously compiled invocations of the function will still be using the old value of the function. If you need to be able to redefine the subroutine, you need to ensure that it isn't inlined, either by dropping the () prototype (which changes calling semantics, so beware) or by thwarting the inlining mechanism in some other way, e.g. by adding an explicit return, as mentioned above:

```
sub not_inlined () { return 23 }
```

73.3.12 Overriding Built-in Functions

Many built-in functions may be overridden, though this should be tried only occasionally and for good reason. Typically this might be done by a package attempting to emulate missing built-in functionality on a non-Unix system.

Overriding may be done only by importing the name from a module at compile time–ordinary predeclaration isn't good enough. However, the use subs pragma lets you, in effect, predeclare subs via the import syntax, and these names may then override built-in ones:

```
use subs 'chdir', 'chroot', 'chmod', 'chown';
chdir $somewhere;
sub chdir { ... }
```

To unambiguously refer to the built-in form, precede the built-in name with the special package qualifier CORE::. For example, saying CORE::open() always refers to the built-in open(), even if the current package has imported some other subroutine called &open() from elsewhere. Even though it looks like a regular function call, it isn't: the CORE:: prefix in that case is part of Perl's syntax, and works for any keyword, regardless of what is in the CORE package. Taking a reference to it, that is, \&CORE::open, only works for some keywords. See CORE.

Library modules should not in general export built-in names like open or chdir as part of their default @EXPORT list, because these may sneak into someone else's namespace and change the semantics unexpectedly. Instead, if the module adds that name to @EXPORT_OK, then it's possible for a user to import the name explicitly, but not implicitly. That is, they could say

```
use Module 'open';
```

and it would import the open override. But if they said

```
use Module;
```

they would get the default imports without overrides.

The foregoing mechanism for overriding built-in is restricted, quite deliberately, to the package that requests the import. There is a second method that is sometimes applicable when you wish to override a built-in everywhere, without regard to namespace boundaries. This is achieved by importing a sub into the special namespace `CORE::GLOBAL::`. Here is an example that quite brazenly replaces the `glob` operator with something that understands regular expressions.

```
package REGlob;
require Exporter;
@ISA = 'Exporter';
@EXPORT_OK = 'glob';

sub import {
    my $pkg = shift;
    return unless @_;
    my $sym = shift;
    my $where = ($sym =~ s/^GLOBAL_// ? 'CORE::GLOBAL' : caller(0));
    $pkg->export($where, $sym, @_);
}

sub glob {
    my $pat = shift;
    my @got;
    if (opendir my $d, '.') {
        @got = grep /$pat/, readdir $d;
        closedir $d;
    }
    return @got;
}
1;
```

And here's how it could be (ab)used:

```
#use REGlob 'GLOBAL_glob';      # override glob() in ALL namespaces
package Foo;
use REGlob 'glob';              # override glob() in Foo:: only
print for <^[a-z_]+\.pm\$>;     # show all pragmatic modules
```

The initial comment shows a contrived, even dangerous example. By overriding `glob` globally, you would be forcing the new (and subversive) behavior for the `glob` operator for *every* namespace, without the complete cognizance or cooperation of the modules that own those namespaces. Naturally, this should be done with extreme caution–if it must be done at all.

The `REGlob` example above does not implement all the support needed to cleanly override perl's `glob` operator. The built-in `glob` has different behaviors depending on whether it appears in a scalar or list context, but our `REGlob` doesn't. Indeed, many perl built-in have such context sensitive behaviors, and these must be adequately supported by a properly written override. For a fully functional example of overriding `glob`, study the implementation of `File::DosGlob` in the standard library.

When you override a built-in, your replacement should be consistent (if possible) with the built-in native syntax. You can achieve this by using a suitable prototype. To get the prototype of an overridable built-in, use the `prototype` function with an argument of `"CORE::builtin_name"` (see [perlfunc prototype], page 429).

Note however that some built-ins can't have their syntax expressed by a prototype (such as `system` or `chomp`). If you override them you won't be able to fully mimic their original syntax.

The built-ins `do`, `require` and `glob` can also be overridden, but due to special magic, their original syntax is preserved, and you don't have to define a prototype for their replacements. (You can't override the `do BLOCK` syntax, though).

`require` has special additional dark magic: if you invoke your `require` replacement as `require Foo::Bar`, it will actually receive the argument `"Foo/Bar.pm"` in @_. See [perlfunc require], page 437.

And, as you'll have noticed from the previous example, if you override `glob`, the `<*>` glob operator is overridden as well.

In a similar fashion, overriding the `readline` function also overrides the equivalent I/O operator `<FILEHANDLE>`. Also, overriding `readpipe` also overrides the operators ` `` ` and `qx//`.

Finally, some built-ins (e.g. `exists` or `grep`) can't be overridden.

73.3.13 Autoloading

If you call a subroutine that is undefined, you would ordinarily get an immediate, fatal error complaining that the subroutine doesn't exist. (Likewise for subroutines being used as methods, when the method doesn't exist in any base class of the class's package.) However, if an `AUTOLOAD` subroutine is defined in the package or packages used to locate the original subroutine, then that `AUTOLOAD` subroutine is called with the arguments that would have been passed to the original subroutine. The fully qualified name of the original subroutine magically appears in the global $AUTOLOAD variable of the same package as the `AUTOLOAD` routine. The name is not passed as an ordinary argument because, er, well, just because, that's why. (As an exception, a method call to a nonexistent `import` or `unimport` method is just skipped instead. Also, if the AUTOLOAD subroutine is an XSUB, there are other ways to retrieve the subroutine name. See Section 28.4.2 [perlguts Autoloading with XSUBs], page 538 for details.)

Many `AUTOLOAD` routines load in a definition for the requested subroutine using eval(), then execute that subroutine using a special form of goto() that erases the stack frame of the `AUTOLOAD` routine without a trace. (See the source to the standard module documented in `AutoLoader`, for example.) But an `AUTOLOAD` routine can also just emulate the routine and never define it. For example, let's pretend that a function that wasn't defined should just invoke `system` with those arguments. All you'd do is:

```
sub AUTOLOAD {
    my $program = $AUTOLOAD;
    $program =~ s/.*:://;
    system($program, @_);
}
date();
```

```
who('am', 'i');
ls('-l');
```

In fact, if you predeclare functions you want to call that way, you don't even need parentheses:

```
use subs qw(date who ls);
date;
who "am", "i";
ls '-l';
```

A more complete example of this is the Shell module on CPAN, which can treat undefined subroutine calls as calls to external programs.

Mechanisms are available to help modules writers split their modules into autoloadable files. See the standard AutoLoader module described in `AutoLoader` and in `AutoSplit`, the standard SelfLoader modules in `SelfLoader`, and the document on adding C functions to Perl code in `perlxs`.

73.3.14 Subroutine Attributes

A subroutine declaration or definition may have a list of attributes associated with it. If such an attribute list is present, it is broken up at space or colon boundaries and treated as though a `use attributes` had been seen. See `attributes` for details about what attributes are currently supported. Unlike the limitation with the obsolescent `use attrs`, the `sub : ATTRLIST` syntax works to associate the attributes with a pre-declaration, and not just with a subroutine definition.

The attributes must be valid as simple identifier names (without any punctuation other than the '_' character). They may have a parameter list appended, which is only checked for whether its parentheses ('(',')') nest properly.

Examples of valid syntax (even though the attributes are unknown):

```
sub fnord (&\%) : switch(10,foo(7,3))  :  expensive;
sub plugh () : Ugly('\(") :Bad;
sub xyzzy : _5x5 { ... }
```

Examples of invalid syntax:

```
sub fnord : switch(10,foo(); # ()-string not balanced
sub snoid : Ugly('(');        # ()-string not balanced
sub xyzzy : 5x5;              # "5x5" not a valid identifier
sub plugh : Y2::north;        # "Y2::north" not a simple identifier
sub snurt : foo + bar;        # "+" not a colon or space
```

The attribute list is passed as a list of constant strings to the code which associates them with the subroutine. In particular, the second example of valid syntax above currently looks like this in terms of how it's parsed and invoked:

```
use attributes __PACKAGE__, \&plugh, q[Ugly('\(")], 'Bad';
```

For further details on attribute lists and their manipulation, see `attributes` and `Attribute-Handlers`.

73.4 SEE ALSO

See Section 62.3.7 [perlref Function Templates], page 1086 for more about references and closures. See `perlxs` if you'd like to learn about calling C subroutines from Perl. See Section 20.1 [perlembed NAME], page 301 if you'd like to learn about calling Perl subroutines from C. See Section 40.1 [perlmod NAME], page 732 to learn about bundling up your functions in separate files. See `perlmodlib` to learn what library modules come standard on your system. See Section 47.1 [perlootut NAME], page 786 to learn how to make object method calls.

74 perlsyn

74.1 NAME

perlsyn - Perl syntax

74.2 DESCRIPTION

A Perl program consists of a sequence of declarations and statements which run from the top to the bottom. Loops, subroutines, and other control structures allow you to jump around within the code.

Perl is a **free-form** language: you can format and indent it however you like. Whitespace serves mostly to separate tokens, unlike languages like Python where it is an important part of the syntax, or Fortran where it is immaterial.

Many of Perl's syntactic elements are **optional**. Rather than requiring you to put parentheses around every function call and declare every variable, you can often leave such explicit elements off and Perl will figure out what you meant. This is known as **Do What I Mean**, abbreviated **DWIM**. It allows programmers to be **lazy** and to code in a style with which they are comfortable.

Perl **borrows syntax** and concepts from many languages: awk, sed, C, Bourne Shell, Smalltalk, Lisp and even English. Other languages have borrowed syntax from Perl, particularly its regular expression extensions. So if you have programmed in another language you will see familiar pieces in Perl. They often work the same, but see Section 80.1 [perltrap NAME], page 1312 for information about how they differ.

74.2.1 Declarations

The only things you need to declare in Perl are report formats and subroutines (and sometimes not even subroutines). A scalar variable holds the undefined value (`undef`) until it has been assigned a defined value, which is anything other than `undef`. When used as a number, `undef` is treated as `0`; when used as a string, it is treated as the empty string, `""`; and when used as a reference that isn't being assigned to, it is treated as an error. If you enable warnings, you'll be notified of an uninitialized value whenever you treat `undef` as a string or a number. Well, usually. Boolean contexts, such as:

```
    if ($a) {}
```

are exempt from warnings (because they care about truth rather than definedness). Operators such as `++`, `--`, `+=`, `-=`, and `.=`, that operate on undefined variables such as:

```
    undef $a;
    $a++;
```

are also always exempt from such warnings.

A declaration can be put anywhere a statement can, but has no effect on the execution of the primary sequence of statements: declarations all take effect at compile time. All declarations are typically put at the beginning or the end of the script. However, if you're using lexically-scoped private variables created with `my()`, `state()`, or `our()`, you'll have to make sure your format or subroutine definition is within the same block scope as the my if you expect to be able to access those private variables.

Declaring a subroutine allows a subroutine name to be used as if it were a list operator from that point forward in the program. You can declare a subroutine without defining it by saying `sub name`, thus:

```
  sub myname;
  $me = myname $0             or die "can't get myname";
```

A bare declaration like that declares the function to be a list operator, not a unary operator, so you have to be careful to use parentheses (or `or` instead of `||`.) The `||` operator binds too tightly to use after list operators; it becomes part of the last element. You can always use parentheses around the list operators arguments to turn the list operator back into something that behaves more like a function call. Alternatively, you can use the prototype `($)` to turn the subroutine into a unary operator:

```
sub myname ($);
$me = myname $0             || die "can't get myname";
```

That now parses as you'd expect, but you still ought to get in the habit of using parentheses in that situation. For more on prototypes, see Section 73.1 [perlsub NAME], page 1216.

Subroutines declarations can also be loaded up with the `require` statement or both loaded and imported into your namespace with a `use` statement. See Section 40.1 [perlmod NAME], page 732 for details on this.

A statement sequence may contain declarations of lexically-scoped variables, but apart from declaring a variable name, the declaration acts like an ordinary statement, and is elaborated within the sequence of statements as if it were an ordinary statement. That means it actually has both compile-time and run-time effects.

74.2.2 Comments

Text from a `"#"` character until the end of the line is a comment, and is ignored. Exceptions include `"#"` inside a string or regular expression.

74.2.3 Simple Statements

The only kind of simple statement is an expression evaluated for its side-effects. Every simple statement must be terminated with a semicolon, unless it is the final statement in a block, in which case the semicolon is optional. But put the semicolon in anyway if the block takes up more than one line, because you may eventually add another line. Note that there are operators like `eval {}`, `sub {}`, and `do {}` that *look* like compound statements, but aren't–they're just TERMs in an expression–and thus need an explicit termination when used as the last item in a statement.

74.2.4 Truth and Falsehood

The number 0, the strings `'0'` and `""`, the empty list `()`, and `undef` are all false in a boolean context. All other values are true. Negation of a true value by `!` or `not` returns a special false value. When evaluated as a string it is treated as `""`, but as a number, it is treated as 0. Most Perl operators that return true or false behave this way.

74.2.5 Statement Modifiers

Any simple statement may optionally be followed by a *SINGLE* modifier, just before the terminating semicolon (or block ending). The possible modifiers are:

```
if EXPR
unless EXPR
while EXPR
until EXPR
for LIST
foreach LIST
when EXPR
```

The `EXPR` following the modifier is referred to as the "condition". Its truth or falsehood determines how the modifier will behave.

`if` executes the statement once *if* and only if the condition is true. `unless` is the opposite, it executes the statement *unless* the condition is true (that is, if the condition is false).

```
print "Basset hounds got long ears" if length $ear >= 10;
go_outside() and play() unless $is_raining;
```

The `for(each)` modifier is an iterator: it executes the statement once for each item in the LIST (with `$_` aliased to each item in turn).

```
print "Hello $_!\n" for qw(world Dolly nurse);
```

`while` repeats the statement *while* the condition is true. `until` does the opposite, it repeats the statement *until* the condition is true (or while the condition is false):

```
# Both of these count from 0 to 10.
print $i++ while $i <= 10;
print $j++ until $j >  10;
```

The `while` and `until` modifiers have the usual "`while` loop" semantics (conditional evaluated first), except when applied to a `do`-BLOCK (or to the Perl4 `do`-SUBROUTINE statement), in which case the block executes once before the conditional is evaluated.

This is so that you can write loops like:

```
do {
    $line = <STDIN>;
    ...
} until !defined($line) || $line eq ".\n"
```

See ⟨undefined⟩ [perlfunc do], page ⟨undefined⟩. Note also that the loop control statements described later will *NOT* work in this construct, because modifiers don't take loop labels. Sorry. You can always put another block inside of it (for `next`) or around it (for `last`) to do that sort of thing. For `next`, just double the braces:

```
do {{
    next if $x == $y;
    # do something here
}} until $x++ > $z;
```

For `last`, you have to be more elaborate:

```
LOOP: {
        do {
            last if $x = $y**2;
            # do something here
        } while $x++ <= $z;
}
```

NOTE: The behaviour of a `my`, `state`, or `our` modified with a statement modifier conditional or loop construct (for example, `my $x if ...`) is **undefined**. The value of the `my` variable may be `undef`, any previously assigned value, or possibly anything else. Don't rely on it. Future versions of perl might do something different from the version of perl you try it out on. Here be dragons.

The `when` modifier is an experimental feature that first appeared in Perl 5.14. To use it, you should include a `use v5.14` declaration. (Technically, it requires only the `switch` feature, but that aspect of it was not available before 5.14.) Operative only from within a `foreach` loop or a `given` block, it executes the statement only if the smartmatch `$_ ~~` *`EXPR`* is true. If the statement executes, it is followed by a `next` from inside a `foreach` and `break` from inside a `given`.

Under the current implementation, the `foreach` loop can be anywhere within the `when` modifier's dynamic scope, but must be within the `given` block's lexical scope. This restricted may be relaxed in a future release. See Section 74.2.11 [Switch Statements], page 1258 below.

74.2.6 Compound Statements

In Perl, a sequence of statements that defines a scope is called a block. Sometimes a block is delimited by the file containing it (in the case of a required file, or the program as a whole), and sometimes a block is delimited by the extent of a string (in the case of an eval).

But generally, a block is delimited by curly brackets, also known as braces. We will call this syntactic construct a BLOCK.

The following compound statements may be used to control flow:

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ...
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK

unless (EXPR) BLOCK
unless (EXPR) BLOCK else BLOCK
unless (EXPR) BLOCK elsif (EXPR) BLOCK ...
unless (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK

given (EXPR) BLOCK

LABEL while (EXPR) BLOCK
LABEL while (EXPR) BLOCK continue BLOCK

LABEL until (EXPR) BLOCK
LABEL until (EXPR) BLOCK continue BLOCK

LABEL for (EXPR; EXPR; EXPR) BLOCK
LABEL for VAR (LIST) BLOCK
LABEL for VAR (LIST) BLOCK continue BLOCK
LABEL foreach (EXPR; EXPR; EXPR) BLOCK
LABEL foreach VAR (LIST) BLOCK
LABEL foreach VAR (LIST) BLOCK continue BLOCK

LABEL BLOCK
LABEL BLOCK continue BLOCK

PHASE BLOCK
```

The experimental `given` statement is *not automatically enabled*; see Section 74.2.11 [Switch Statements], page 1258 below for how to do so, and the attendant caveats.

Unlike in C and Pascal, in Perl these are all defined in terms of BLOCKs, not statements. This means that the curly brackets are *required*–no dangling statements allowed. If you want to write conditionals without curly brackets, there are several other ways to do it. The following all do the same thing:

```
if (!open(FOO)) { die "Can't open $FOO: $!" }
die "Can't open $FOO: $!" unless open(FOO);
open(FOO)  || die "Can't open $FOO: $!";
open(FOO) ? () : die "Can't open $FOO: $!";
    # a bit exotic, that last one
```

The `if` statement is straightforward. Because BLOCKs are always bounded by curly brackets, there is never any ambiguity about which `if` an `else` goes with. If you use `unless` in place of `if`, the sense of the test is reversed. Like `if`, `unless` can be followed by `else`. `unless` can even be followed by one or more `elsif` statements, though you may want to think twice before using that particular language construct, as everyone reading your code will have to think at least twice before they can understand what's going on.

The `while` statement executes the block as long as the expression is Section 74.2.4 [true], page 1250. The `until` statement executes the block as long as the expression is false. The LABEL is optional, and if present, consists of an identifier followed by a colon. The LABEL identifies the loop for the loop control statements `next`, `last`, and `redo`. If the LABEL is omitted, the loop control statement refers to the innermost enclosing loop. This may include dynamically looking back your call-stack at run time to find the LABEL. Such desperate behavior triggers a warning if you use the `use warnings` pragma or the `-w` flag.

If there is a `continue` BLOCK, it is always executed just before the conditional is about to be evaluated again. Thus it can be used to increment a loop variable, even when the loop has been continued via the `next` statement.

When a block is preceding by a compilation phase keyword such as `BEGIN`, `END`, `INIT`, `CHECK`, or `UNITCHECK`, then the block will run only during the corresponding phase of execution. See Section 40.1 [perlmod NAME], page 732 for more details.

Extension modules can also hook into the Perl parser to define new kinds of compound statements. These are introduced by a keyword which the extension recognizes, and the syntax following the keyword is defined entirely by the extension. If you are an implementor, see Section "PL_keyword_plugin" in `perlapi` for the mechanism. If you are using such a module, see the module's documentation for details of the syntax that it defines.

74.2.7 Loop Control

The next command starts the next iteration of the loop:

```
LINE: while (<STDIN>) {
    next LINE if /^#/;      # discard comments
    ...
}
```

The last command immediately exits the loop in question. The continue block, if any, is not executed:

```
LINE: while (<STDIN>) {
    last LINE if /^$/;      # exit when done with header
    ...
}
```

The redo command restarts the loop block without evaluating the conditional again. The continue block, if any, is *not* executed. This command is normally used by programs that want to lie to themselves about what was just input.

For example, when processing a file like /etc/termcap. If your input lines might end in backslashes to indicate continuation, you want to skip ahead and get the next record.

```
while (<>) {
    chomp;
    if (s/\\$//) {
        $_ .= <>;
        redo unless eof();
    }
    # now process $_
}
```

which is Perl shorthand for the more explicitly written version:

```
LINE: while (defined($line = <ARGV>)) {
    chomp($line);
    if ($line =~ s/\\$//) {
        $line .= <ARGV>;
        redo LINE unless eof(); # not eof(ARGV)!
    }
    # now process $line
}
```

Note that if there were a continue block on the above code, it would get executed only on lines discarded by the regex (since redo skips the continue block). A continue block is often used to reset line counters or m?pat? one-time matches:

```
# inspired by :1,$g/fred/s//WILMA/
while (<>) {
    m?(fred)?    && s//WILMA $1 WILMA/;
    m?(barney)?  && s//BETTY $1 BETTY/;
    m?(homer)?   && s//MARGE $1 MARGE/;
} continue {
    print "$ARGV $.: $_";
```

```
        close ARGV  if eof;             # reset $.
        reset       if eof;             # reset ?pat?
    }
```

If the word while is replaced by the word until, the sense of the test is reversed, but the conditional is still tested before the first iteration.

Loop control statements don't work in an if or unless, since they aren't loops. You can double the braces to make them such, though.

```
    if (/pattern/) {{
        last if /fred/;
        next if /barney/; # same effect as "last",
                          # but doesn't document as well
        # do something here
    }}
```

This is caused by the fact that a block by itself acts as a loop that executes once, see Section 74.2.10 [Basic BLOCKs], page 1257.

The form while/if BLOCK BLOCK, available in Perl 4, is no longer available. Replace any occurrence of if BLOCK by if (do BLOCK).

74.2.8 For Loops

Perl's C-style for loop works like the corresponding while loop; that means that this:

```
    for ($i = 1; $i < 10; $i++) {
        ...
    }
```

is the same as this:

```
    $i = 1;
    while ($i < 10) {
        ...
    } continue {
        $i++;
    }
```

There is one minor difference: if variables are declared with my in the initialization section of the for, the lexical scope of those variables is exactly the for loop (the body of the loop and the control sections).

As a special case, if the test in the for loop (or the corresponding while loop) is empty, it is treated as true. That is, both

```
    for (;;) {
        ...
    }
```

and

```
    while () {
        ...
    }
```

are treated as infinite loops.

Besides the normal array index looping, for can lend itself to many other interesting applications. Here's one that avoids the problem you get into if you explicitly test for end-of-file on an interactive file descriptor causing your program to appear to hang.

```
$on_a_tty = -t STDIN && -t STDOUT;
sub prompt { print "yes? " if $on_a_tty }
for ( prompt(); <STDIN>; prompt() ) {
    # do something
}
```

Using readline (or the operator form, <EXPR>) as the conditional of a for loop is shorthand for the following. This behaviour is the same as a while loop conditional. >>

```
for ( prompt(); defined( $_ = <STDIN> ); prompt() ) {
    # do something
}
```

74.2.9 Foreach Loops

The foreach loop iterates over a normal list value and sets the scalar variable VAR to be each element of the list in turn. If the variable is preceded with the keyword my, then it is lexically scoped, and is therefore visible only within the loop. Otherwise, the variable is implicitly local to the loop and regains its former value upon exiting the loop. If the variable was previously declared with my, it uses that variable instead of the global one, but it's still localized to the loop. This implicit localization occurs *only* in a foreach loop.

The foreach keyword is actually a synonym for the for keyword, so you can use either. If VAR is omitted, $_ is set to each value.

If any element of LIST is an lvalue, you can modify it by modifying VAR inside the loop. Conversely, if any element of LIST is NOT an lvalue, any attempt to modify that element will fail. In other words, the foreach loop index variable is an implicit alias for each item in the list that you're looping over.

If any part of LIST is an array, foreach will get very confused if you add or remove elements within the loop body, for example with splice. So don't do that.

foreach probably won't do what you expect if VAR is a tied or other special variable. Don't do that either.

As of Perl 5.22, there is an experimental variant of this loop that accepts a variable preceded by a backslash for VAR, in which case the items in the LIST must be references. The backslashed variable will become an alias to each referenced item in the LIST, which must be of the correct type. The variable needn't be a scalar in this case, and the backslash may be followed by my. To use this form, you must enable the refaliasing feature via use feature. (See feature. See also Section 62.6 [perlref Assigning to References], page 1089.)

Examples:

```
for (@ary) { s/foo/bar/ }

for my $elem (@elements) {
    $elem *= 2;
}
```

```
for $count (reverse(1..10), "BOOM") {
    print $count, "\n";
    sleep(1);
}

for (1..15) { print "Merry Christmas\n"; }

foreach $item (split(/:[\\\n:]*/, $ENV{TERMCAP})) {
    print "Item: $item\n";
}

use feature "refaliasing";
no warnings "experimental::refaliasing";
foreach \my %hash (@array_of_hash_references) {
    # do something which each %hash
}
```

Here's how a C programmer might code up a particular algorithm in Perl:

```
for (my $i = 0; $i < @ary1; $i++) {
    for (my $j = 0; $j < @ary2; $j++) {
        if ($ary1[$i] > $ary2[$j]) {
            last; # can't go to outer :-(
        }
        $ary1[$i] += $ary2[$j];
    }
    # this is where that last takes me
}
```

Whereas here's how a Perl programmer more comfortable with the idiom might do it:

```
OUTER: for my $wid (@ary1) {
INNER:   for my $jet (@ary2) {
            next OUTER if $wid > $jet;
            $wid += $jet;
         }
      }
```

See how much easier this is? It's cleaner, safer, and faster. It's cleaner because it's less noisy. It's safer because if code gets added between the inner and outer loops later on, the new code won't be accidentally executed. The `next` explicitly iterates the other loop rather than merely terminating the inner one. And it's faster because Perl executes a `foreach` statement more rapidly than it would the equivalent `for` loop.

Perceptive Perl hackers may have noticed that a `for` loop has a return value, and that this value can be captured by wrapping the loop in a `do` block. The reward for this discovery is this cautionary advice: The return value of a `for` loop is unspecified and may change without notice. Do not rely on it.

74.2.10 Basic BLOCKs

A BLOCK by itself (labeled or not) is semantically equivalent to a loop that executes once. Thus you can use any of the loop control statements in it to leave or restart the block.

(Note that this is *NOT* true in `eval{}`, `sub{}`, or contrary to popular belief `do{}` blocks, which do *NOT* count as loops.) The `continue` block is optional.

The BLOCK construct can be used to emulate case structures.

```
SWITCH: {
    if (/^abc/) { $abc = 1; last SWITCH; }
    if (/^def/) { $def = 1; last SWITCH; }
    if (/^xyz/) { $xyz = 1; last SWITCH; }
    $nothing = 1;
}
```

You'll also find that `foreach` loop used to create a topicalizer and a switch:

```
SWITCH:
for ($var) {
    if (/^abc/) { $abc = 1; last SWITCH; }
    if (/^def/) { $def = 1; last SWITCH; }
    if (/^xyz/) { $xyz = 1; last SWITCH; }
    $nothing = 1;
}
```

Such constructs are quite frequently used, both because older versions of Perl had no official `switch` statement, and also because the new version described immediately below remains experimental and can sometimes be confusing.

74.2.11 Switch Statements

Starting from Perl 5.10.1 (well, 5.10.0, but it didn't work right), you can say

```
use feature "switch";
```

to enable an experimental switch feature. This is loosely based on an old version of a Perl 6 proposal, but it no longer resembles the Perl 6 construct. You also get the switch feature whenever you declare that your code prefers to run under a version of Perl that is 5.10 or later. For example:

```
use v5.14;
```

Under the "switch" feature, Perl gains the experimental keywords `given`, `when`, `default`, `continue`, and `break`. Starting from Perl 5.16, one can prefix the switch keywords with `CORE::` to access the feature without a `use feature` statement. The keywords `given` and `when` are analogous to `switch` and `case` in other languages, so the code in the previous section could be rewritten as

```
use v5.10.1;
for ($var) {
    when (/^abc/) { $abc = 1 }
    when (/^def/) { $def = 1 }
    when (/^xyz/) { $xyz = 1 }
    default       { $nothing = 1 }
}
```

The `foreach` is the non-experimental way to set a topicalizer. If you wish to use the highly experimental `given`, that could be written like this:

```
use v5.10.1;
given ($var) {
    when (/^abc/) { $abc = 1 }
    when (/^def/) { $def = 1 }
    when (/^xyz/) { $xyz = 1 }
    default       { $nothing = 1 }
}
```

As of 5.14, that can also be written this way:

```
use v5.14;
for ($var) {
    $abc = 1 when /^abc/;
    $def = 1 when /^def/;
    $xyz = 1 when /^xyz/;
    default { $nothing = 1 }
}
```

Or if you don't care to play it safe, like this:

```
use v5.14;
given ($var) {
    $abc = 1 when /^abc/;
    $def = 1 when /^def/;
    $xyz = 1 when /^xyz/;
    default { $nothing = 1 }
}
```

The arguments to `given` and `when` are in scalar context, and `given` assigns the `$_` variable its topic value.

Exactly what the *EXPR* argument to `when` does is hard to describe precisely, but in general, it tries to guess what you want done. Sometimes it is interpreted as `$_ ~~ EXPR`, and sometimes it is not. It also behaves differently when lexically enclosed by a `given` block than it does when dynamically enclosed by a `foreach` loop. The rules are far too difficult to understand to be described here. See Section 74.2.16 [Experimental Details on given and when], page 1263 later on.

Due to an unfortunate bug in how `given` was implemented between Perl 5.10 and 5.16, under those implementations the version of `$_` governed by `given` is merely a lexically scoped copy of the original, not a dynamically scoped alias to the original, as it would be if it were a `foreach` or under both the original and the current Perl 6 language specification. This bug was fixed in Perl 5.18. If you really want a lexical `$_`, specify that explicitly, but note that `my $_` is now deprecated and will warn unless warnings have been disabled:

```
given(my $_ = EXPR) { ... }
```

If your code still needs to run on older versions, stick to `foreach` for your topicalizer and you will be less unhappy.

74.2.12 Goto

Although not for the faint of heart, Perl does support a `goto` statement. There are three forms: `goto`-LABEL, `goto`-EXPR, and `goto`-&NAME. A loop's LABEL is not actually a valid target for a `goto`; it's just the name of the loop.

The goto-LABEL form finds the statement labeled with LABEL and resumes execution there. It may not be used to go into any construct that requires initialization, such as a subroutine or a foreach loop. It also can't be used to go into a construct that is optimized away. It can be used to go almost anywhere else within the dynamic scope, including out of subroutines, but it's usually better to use some other construct such as last or die. The author of Perl has never felt the need to use this form of goto (in Perl, that is–C is another matter).

The goto-EXPR form expects a label name, whose scope will be resolved dynamically. This allows for computed gotos per FORTRAN, but isn't necessarily recommended if you're optimizing for maintainability:

```
goto(("FOO", "BAR", "GLARCH")[$i]);
```

The goto-&NAME form is highly magical, and substitutes a call to the named subroutine for the currently running subroutine. This is used by AUTOLOAD() subroutines that wish to load another subroutine and then pretend that the other subroutine had been called in the first place (except that any modifications to @_ in the current subroutine are propagated to the other subroutine.) After the goto, not even caller() will be able to tell that this routine was called first.

In almost all cases like this, it's usually a far, far better idea to use the structured control flow mechanisms of next, last, or redo instead of resorting to a goto. For certain applications, the catch and throw pair of eval{} and die() for exception processing can also be a prudent approach.

74.2.13 The Ellipsis Statement

Beginning in Perl 5.12, Perl accepts an ellipsis, "...", as a placeholder for code that you haven't implemented yet. This form of ellipsis, the unimplemented statement, should not be confused with the binary flip-flop ... operator. One is a statement and the other an operator. (Perl doesn't usually confuse them because usually Perl can tell whether it wants an operator or a statement, but see below for exceptions.)

When Perl 5.12 or later encounters an ellipsis statement, it parses this without error, but if and when you should actually try to execute it, Perl throws an exception with the text Unimplemented:

```
use v5.12;
sub unimplemented { ... }
eval { unimplemented() };
if ($@ =~ /^Unimplemented at /) {
    say "I found an ellipsis!";
}
```

You can only use the elliptical statement to stand in for a complete statement. These examples of how the ellipsis works:

```
use v5.12;
{ ... }
sub foo { ... }
...;
eval { ... };
sub somemeth {
    my $self = shift;
    ...;
}
$x = do {
    my $n;
    ...;
    say "Hurrah!";
    $n;
};
```

The elliptical statement cannot stand in for an expression that is part of a larger statement, since the ... is also the three-dot version of the flip-flop operator (see Section 48.2.20 [perlop Range Operators], page 812).

These examples of attempts to use an ellipsis are syntax errors:

```
use v5.12;

print ...;
open(my $fh, ">", "/dev/passwd") or ...;
if ($condition && ... ) { say "Howdy" };
```

There are some cases where Perl can't immediately tell the difference between an expression and a statement. For instance, the syntax for a block and an anonymous hash reference constructor look the same unless there's something in the braces to give Perl a hint. The ellipsis is a syntax error if Perl doesn't guess that the { ... } is a block. In that case, it doesn't think the ... is an ellipsis because it's expecting an expression instead of a statement:

```
@transformed = map { ... } @input;    # syntax error
```

Inside your block, you can use a ; before the ellipsis to denote that the { ... } is a block and not a hash reference constructor. Now the ellipsis works:

```
@transformed = map {; ... } @input;   # ';' disambiguates
```

Note: Some folks colloquially refer to this bit of punctuation as a "yada-yada" or "triple-dot", but its true name is actually an ellipsis.

74.2.14 PODs: Embedded Documentation

Perl has a mechanism for intermixing documentation with source code. While it's expecting the beginning of a new statement, if the compiler encounters a line that begins with an equal sign and a word, like this

```
=head1 Here There Be Pods!
```

Then that text and all remaining text up through and including a line beginning with =cut will be ignored. The format of the intervening text is described in Section 52.1 [perlpod NAME], page 900.

This allows you to intermix your source code and your documentation text freely, as in

```
=item snazzle($)

The snazzle() function will behave in the most spectacular
form that you can possibly imagine, not even excepting
```

```
cybernetic pyrotechnics.

=cut back to the compiler, nuff of this pod stuff!

sub snazzle($) {
    my $thingie = shift;
    .........
}
```

Note that pod translators should look at only paragraphs beginning with a pod directive (it makes parsing easier), whereas the compiler actually knows to look for pod escapes even in the middle of a paragraph. This means that the following secret stuff will be ignored by both the compiler and the translators.

```
$a=3;
=secret stuff
 warn "Neither POD nor CODE!?"
=cut back
print "got $a\n";
```

You probably shouldn’t rely upon the `warn()` being podded out forever. Not all pod translators are well-behaved in this regard, and perhaps the compiler will become pickier.

One may also use pod directives to quickly comment out a section of code.

74.2.15 Plain Old Comments (Not!)

Perl can process line directives, much like the C preprocessor. Using this, one can control Perl’s idea of filenames and line numbers in error or warning messages (especially for strings that are processed with `eval()`). The syntax for this mechanism is almost the same as for most C preprocessors: it matches the regular expression

```
# example: ’# line 42 "new_filename.plx"’
/^\#   \s*
  line \s+ (\d+)   \s*
  (?:\s("?)([^"]+)\g2)? \s*
 $/x
```

with `$1` being the line number for the next line, and `$3` being the optional filename (specified with or without quotes). Note that no whitespace may precede the `#`, unlike modern C preprocessors.

There is a fairly obvious gotcha included with the line directive: Debuggers and profilers will only show the last source line to appear at a particular line number in a given file. Care should be taken not to cause line number collisions in code you’d like to debug later.

Here are some examples that you should be able to type into your command shell:

```
% perl
# line 200 "bzzzt"
# the ’#’ on the previous line must be the first char on line
die ’foo’;
__END__
foo at bzzzt line 201.
```

```
% perl
# line 200 "bzzzt"
eval qq[\n#line 2001 ""\ndie 'foo']; print $@;
__END__
foo at - line 2001.

% perl
eval qq[\n#line 200 "foo bar"\ndie 'foo']; print $@;
__END__
foo at foo bar line 200.

% perl
# line 345 "goop"
eval "\n#line " . __LINE__ . ' "' . __FILE__ ."\"\ndie 'foo'";
print $@;
__END__
foo at goop line 345.
```

74.2.16 Experimental Details on given and when

As previously mentioned, the "switch" feature is considered highly experimental; it is subject to change with little notice. In particular, `when` has tricky behaviours that are expected to change to become less tricky in the future. Do not rely upon its current (mis)implementation. Before Perl 5.18, `given` also had tricky behaviours that you should still beware of if your code must run on older versions of Perl.

Here is a longer example of `given`:

```
use feature ":5.10";
given ($foo) {
    when (undef) {
        say '$foo is undefined';
    }
    when ("foo") {
        say '$foo is the string "foo"';
    }
    when ([1,3,5,7,9]) {
        say '$foo is an odd digit';
        continue; # Fall through
    }
    when ($_ < 100) {
        say '$foo is numerically less than 100';
    }
    when (\&complicated_check) {
        say 'a complicated check for $foo is true';
    }
    default {
        die q(I don't know what to do with $foo);
    }
}
```

Before Perl 5.18, `given(EXPR)` assigned the value of *EXPR* to merely a lexically scoped **copy** (!) of `$_`, not a dynamically scoped alias the way `foreach` does. That made it similar to

```
    do { my $_ = EXPR; ... }
```

except that the block was automatically broken out of by a successful `when` or an explicit `break`. Because it was only a copy, and because it was only lexically scoped, not dynamically scoped, you could not do the things with it that you are used to in a `foreach` loop. In particular, it did not work for arbitrary function calls if those functions might try to access $_. Best stick to `foreach` for that.

Most of the power comes from the implicit smartmatching that can sometimes apply. Most of the time, `when(EXPR)` is treated as an implicit smartmatch of `$_`, that is, `$_ ~~ EXPR`. (See Section 48.2.14 [perlop Smartmatch Operator], page 805 for more information on smartmatching.) But when *EXPR* is one of the 10 exceptional cases (or things like them) listed below, it is used directly as a boolean.

1.

   A user-defined subroutine call or a method invocation.

2.

   A regular expression match in the form of `/REGEX/`, `$foo =~ /REGEX/`, or `$foo =~ EXPR`. Also, a negated regular expression match in the form `!/REGEX/`, `$foo !~ /REGEX/`, or `$foo !~ EXPR`.

3.

   A smart match that uses an explicit `~~` operator, such as `EXPR ~~ EXPR`.

   NOTE: You will often have to use `$c ~~ $_` because the default case uses `$_ ~~ $c` , which is frequentlythe opposite of what you want.

4.

   A boolean comparison operator such as `$_ < 10` or `$x eq "abc"`. The relational operators that this applies to are the six numeric comparisons (`<`, `>`, `<=`, `>=`, `==`, and `!=`), and the six string comparisons (`lt`, `gt`, `le`, `ge`, `eq`, and `ne`).

5.

   At least the three builtin functions `defined(...)`, `exists(...)`, and `eof(...)`. We might someday add more of these later if we think of them.

6.

   A negated expression, whether `!(EXPR)` or `not(EXPR)`, or a logical exclusive-or, `(EXPR1) xor (EXPR2)`. The bitwise versions (`~` and `^`) are not included.

7.

   A filetest operator, with exactly 4 exceptions: `-s`, `-M`, `-A`, and `-C`, as these return numerical values, not boolean ones. The `-z` filetest operator is not included in the exception list.

8.

   The `..` and `...` flip-flop operators. Note that the `...` flip-flop operator is completely different from the `...` elliptical statement just described.

In those 8 cases above, the value of EXPR is used directly as a boolean, so no smartmatching is done. You may think of `when` as a smartsmartmatch.

Furthermore, Perl inspects the operands of logical operators to decide whether to use smartmatching for each one by applying the above test to the operands:

9.

If EXPR is `EXPR1 && EXPR2` or `EXPR1 and EXPR2`, the test is applied *recursively* to both EXPR1 and EXPR2. Only if *both* operands also pass the test, *recursively*, will the expression be treated as boolean. Otherwise, smartmatching is used.

10.

If EXPR is `EXPR1 || EXPR2`, `EXPR1 // EXPR2`, or `EXPR1 or EXPR2`, the test is applied *recursively* to EXPR1 only (which might itself be a higher-precedence AND operator, for example, and thus subject to the previous rule), not to EXPR2. If EXPR1 is to use smartmatching, then EXPR2 also does so, no matter what EXPR2 contains. But if EXPR2 does not get to use smartmatching, then the second argument will not be either. This is quite different from the `&&` case just described, so be careful.

These rules are complicated, but the goal is for them to do what you want (even if you don't quite understand why they are doing it). For example:

```
when (/^\d+$/ && $_ < 75) { ... }
```

will be treated as a boolean match because the rules say both a regex match and an explicit test on `$_` will be treated as boolean.

Also:

```
when ([qw(foo bar)] && /baz/) { ... }
```

will use smartmatching because only *one* of the operands is a boolean: the other uses smartmatching, and that wins.

Further:

```
when ([qw(foo bar)] || /^baz/) { ... }
```

will use smart matching (only the first operand is considered), whereas

```
when (/^baz/ || [qw(foo bar)]) { ... }
```

will test only the regex, which causes both operands to be treated as boolean. Watch out for this one, then, because an arrayref is always a true value, which makes it effectively redundant. Not a good idea.

Tautologous boolean operators are still going to be optimized away. Don't be tempted to write

```
when ("foo" or "bar") { ... }
```

This will optimize down to `"foo"`, so `"bar"` will never be considered (even though the rules say to use a smartmatch on `"foo"`). For an alternation like this, an array ref will work, because this will instigate smartmatching:

```
when ([qw(foo bar)] { ... }
```

This is somewhat equivalent to the C-style switch statement's fallthrough functionality (not to be confused with *Perl's* fallthrough functionality–see below), wherein the same block is used for several `case` statements.

Another useful shortcut is that, if you use a literal array or hash as the argument to `given`, it is turned into a reference. So `given(@foo)` is the same as `given(\@foo)`, for example.

`default` behaves exactly like `when(1 == 1)`, which is to say that it always matches.

74.2.16.1 Breaking out

You can use the `break` keyword to break out of the enclosing `given` block. Every `when` block is implicitly ended with a `break`.

74.2.16.2 Fall-through

You can use the `continue` keyword to fall through from one case to the next:

```
given($foo) {
    when (/x/) { say '$foo contains an x'; continue }
    when (/y/) { say '$foo contains a y'            }
    default    { say '$foo does not contain a y'    }
}
```

74.2.16.3 Return value

When a `given` statement is also a valid expression (for example, when it's the last statement of a block), it evaluates to:

- An empty list as soon as an explicit `break` is encountered.
- The value of the last evaluated expression of the successful `when`/`default` clause, if there happens to be one.
- The value of the last evaluated expression of the `given` block if no condition is true.

In both last cases, the last expression is evaluated in the context that was applied to the `given` block.

Note that, unlike `if` and `unless`, failed `when` statements always evaluate to an empty list.

```
my $price = do {
    given ($item) {
        when (["pear", "apple"]) { 1 }
        break when "vote";      # My vote cannot be bought
        1e10  when /Mona Lisa/;
        "unknown";
    }
};
```

Currently, `given` blocks can't always be used as proper expressions. This may be addressed in a future version of Perl.

74.2.16.4 Switching in a loop

Instead of using `given()`, you can use a `foreach()` loop. For example, here's one way to count how many times a particular string occurs in an array:

```
use v5.10.1;
my $count = 0;
for (@array) {
    when ("foo") { ++$count }
}
print "\@array contains $count copies of 'foo'\n";
```

Or in a more recent version:

```
use v5.14;
my $count = 0;
for (@array) {
    ++$count when "foo";
}
print "\@array contains $count copies of 'foo'\n";
```

At the end of all `when` blocks, there is an implicit `next`. You can override that with an explicit `last` if you're interested in only the first match alone.

This doesn't work if you explicitly specify a loop variable, as in `for $item (@array)`. You have to use the default variable `$_`.

74.2.16.5 Differences from Perl 6

The Perl 5 smartmatch and `given`/`when` constructs are not compatible with their Perl 6 analogues. The most visible difference and least important difference is that, in Perl 5, parentheses are required around the argument to `given()` and `when()` (except when this last one is used as a statement modifier). Parentheses in Perl 6 are always optional in a control construct such as `if()`, `while()`, or `when()`; they can't be made optional in Perl 5 without a great deal of potential confusion, because Perl 5 would parse the expression

```
given $foo {
    ...
}
```

as though the argument to `given` were an element of the hash `%foo`, interpreting the braces as hash-element syntax.

However, their are many, many other differences. For example, this works in Perl 5:

```
use v5.12;
my @primary = ("red", "blue", "green");

if (@primary ~~ "red") {
    say "primary smartmatches red";
}

if ("red" ~~ @primary) {
    say "red smartmatches primary";
}
```

```
    say "that’s all, folks!";
```

But it doesn’t work at all in Perl 6. Instead, you should use the (parallelizable) `any` operator:

```
if any(@primary) eq "red" {
    say "primary smartmatches red";
}

if "red" eq any(@primary) {
    say "red smartmatches primary";
}
```

The table of smartmatches in Section 48.2.14 [perlop Smartmatch Operator], page 805 is not identical to that proposed by the Perl 6 specification, mainly due to differences between Perl 6’s and Perl 5’s data models, but also because the Perl 6 spec has changed since Perl 5 rushed into early adoption.

In Perl 6, `when()` will always do an implicit smartmatch with its argument, while in Perl 5 it is convenient (albeit potentially confusing) to suppress this implicit smartmatch in various rather loosely-defined situations, as roughly outlined above. (The difference is largely because Perl 5 does not have, even internally, a boolean type.)

75 perlthrtut

75.1 NAME

perlthrtut - Tutorial on threads in Perl

75.2 DESCRIPTION

This tutorial describes the use of Perl interpreter threads (sometimes referred to as *ithreads*). In this model, each thread runs in its own Perl interpreter, and any data sharing between threads must be explicit. The user-level interface for *ithreads* uses the `threads` class.

NOTE: There was another older Perl threading flavor called the 5.005 model that used the `threads` class. This old model was known to have problems, is deprecated, and was removed for release 5.10. You are strongly encouraged to migrate any existing 5.005 threads code to the new model as soon as possible.

You can see which (or neither) threading flavour you have by running `perl -V` and looking at the `Platform` section. If you have `useithreads=define` you have ithreads, if you have `use5005threads=define` you have 5.005 threads. If you have neither, you don't have any thread support built in. If you have both, you are in trouble.

The `threads` and `threads-shared` modules are included in the core Perl distribution. Additionally, they are maintained as a separate modules on CPAN, so you can check there for any updates.

75.3 What Is A Thread Anyway?

A thread is a flow of control through a program with a single execution point.

Sounds an awful lot like a process, doesn't it? Well, it should. Threads are one of the pieces of a process. Every process has at least one thread and, up until now, every process running Perl had only one thread. With 5.8, though, you can create extra threads. We're going to show you how, when, and why.

75.4 Threaded Program Models

There are three basic ways that you can structure a threaded program. Which model you choose depends on what you need your program to do. For many non-trivial threaded programs, you'll need to choose different models for different pieces of your program.

75.4.1 Boss/Worker

The boss/worker model usually has one *boss* thread and one or more *worker* threads. The boss thread gathers or generates tasks that need to be done, then parcels those tasks out to the appropriate worker thread.

This model is common in GUI and server programs, where a main thread waits for some event and then passes that event to the appropriate worker threads for processing. Once the event has been passed on, the boss thread goes back to waiting for another event.

The boss thread does relatively little work. While tasks aren't necessarily performed faster than with any other method, it tends to have the best user-response times.

75.4.2 Work Crew

In the work crew model, several threads are created that do essentially the same thing to different pieces of data. It closely mirrors classical parallel processing and vector processors, where a large array of processors do the exact same thing to many pieces of data.

This model is particularly useful if the system running the program will distribute multiple threads across different processors. It can also be useful in ray tracing or rendering engines, where the individual threads can pass on interim results to give the user visual feedback.

75.4.3 Pipeline

The pipeline model divides up a task into a series of steps, and passes the results of one step on to the thread processing the next. Each thread does one thing to each piece of data and passes the results to the next thread in line.

This model makes the most sense if you have multiple processors so two or more threads will be executing in parallel, though it can often make sense in other contexts as well. It tends to keep the individual tasks small and simple, as well as allowing some parts of the pipeline to block (on I/O or system calls, for example) while other parts keep going. If you're running different parts of the pipeline on different processors you may also take advantage of the caches on each processor.

This model is also handy for a form of recursive programming where, rather than having a subroutine call itself, it instead creates another thread. Prime and Fibonacci generators both map well to this form of the pipeline model. (A version of a prime number generator is presented later on.)

75.5 What kind of threads are Perl threads?

If you have experience with other thread implementations, you might find that things aren't quite what you expect. It's very important to remember when dealing with Perl threads that *Perl Threads Are Not X Threads* for all values of X. They aren't POSIX threads, or DecThreads, or Java's Green threads, or Win32 threads. There are similarities, and the broad concepts are the same, but if you start looking for implementation details you're going to be either disappointed or confused. Possibly both.

This is not to say that Perl threads are completely different from everything that's ever come before. They're not. Perl's threading model owes a lot to other thread models, especially POSIX. Just as Perl is not C, though, Perl threads are not POSIX threads. So if you find yourself looking for mutexes, or thread priorities, it's time to step back a bit and think about what you want to do and how Perl can do it.

However, it is important to remember that Perl threads cannot magically do things unless your operating system's threads allow it. So if your system blocks the entire process on `sleep()`, Perl usually will, as well.

Perl Threads Are Different.

75.6 Thread-Safe Modules

The addition of threads has changed Perl's internals substantially. There are implications for people who write modules with XS code or external libraries. However, since Perl data is not shared among threads by default, Perl modules stand a high chance of being thread-safe or can be made thread-safe easily. Modules that are not tagged as thread-safe should be tested or code reviewed before being used in production code.

Not all modules that you might use are thread-safe, and you should always assume a module is unsafe unless the documentation says otherwise. This includes modules that are distributed as part of the core. Threads are a relatively new feature, and even some of the standard modules aren't thread-safe.

Even if a module is thread-safe, it doesn't mean that the module is optimized to work well with threads. A module could possibly be rewritten to utilize the new features in threaded Perl to increase performance in a threaded environment.

If you're using a module that's not thread-safe for some reason, you can protect yourself by using it from one, and only one thread at all. If you need multiple threads to access such a module, you can use semaphores and lots of programming discipline to control access to it. Semaphores are covered in Section 75.9.5 [Basic semaphores], page 1280.

See also Section 75.15 [Thread-Safety of System Libraries], page 1286.

75.7 Thread Basics

The `threads` module provides the basic functions you need to write threaded programs. In the following sections, we'll cover the basics, showing you what you need to do to create a threaded program. After that, we'll go over some of the features of the `threads` module that make threaded programming easier.

75.7.1 Basic Thread Support

Thread support is a Perl compile-time option. It's something that's turned on or off when Perl is built at your site, rather than when your programs are compiled. If your Perl wasn't compiled with thread support enabled, then any attempt to use threads will fail.

Your programs can use the Config module to check whether threads are enabled. If your program can't run without them, you can say something like:

```
use Config;
$Config{useithreads} or
    die('Recompile Perl with threads to run this program.');
```

A possibly-threaded program using a possibly-threaded module might have code like this:

```
use Config;
use MyMod;

BEGIN {
    if ($Config{useithreads}) {
        # We have threads
        require MyMod_threaded;
        import MyMod_threaded;
    } else {
        require MyMod_unthreaded;
        import MyMod_unthreaded;
        }
    }
```

Since code that runs both with and without threads is usually pretty messy, it's best to isolate the thread-specific code in its own module. In our example above, that's what `MyMod_threaded` is, and it's only imported if we're running on a threaded Perl.

75.7.2 A Note about the Examples

In a real situation, care should be taken that all threads are finished executing before the program exits. That care has **not** been taken in these examples in the interest of simplicity. Running these examples *as is* will produce error messages, usually caused by the fact that there are still threads running when the program exits. You should not be alarmed by this.

75.7.3 Creating Threads

The `threads` module provides the tools you need to create new threads. Like any other module, you need to tell Perl that you want to use it; `use threads;` imports all the pieces you need to create basic threads.

The simplest, most straightforward way to create a thread is with `create()`:

```
    use threads;

    my $thr = threads->create(\&sub1);

    sub sub1 {
        print("In the thread\n");
    }
```

The `create()` method takes a reference to a subroutine and creates a new thread that starts executing in the referenced subroutine. Control then passes both to the subroutine and the caller.

If you need to, your program can pass parameters to the subroutine as part of the thread startup. Just include the list of parameters as part of the `threads->create()` call, like this:

```
    use threads;

    my $Param3 = 'foo';
    my $thr1 = threads->create(\&sub1, 'Param 1', 'Param 2', $Param3);
    my @ParamList = (42, 'Hello', 3.14);
    my $thr2 = threads->create(\&sub1, @ParamList);
    my $thr3 = threads->create(\&sub1, qw(Param1 Param2 Param3));

    sub sub1 {
        my @InboundParameters = @_;
        print("In the thread\n");
        print('Got parameters >', join('<>',@InboundParameters), "<\n");
    }
```

The last example illustrates another feature of threads. You can spawn off several threads using the same subroutine. Each thread executes the same subroutine, but in a separate thread with a separate environment and potentially separate arguments.

`new()` is a synonym for `create()`.

75.7.4 Waiting For A Thread To Exit

Since threads are also subroutines, they can return values. To wait for a thread to exit and extract any values it might return, you can use the `join()` method:

```
use threads;

my ($thr) = threads->create(\&sub1);

my @ReturnData = $thr->join();
print('Thread returned ', join(', ', @ReturnData), "\n");

sub sub1 { return ('Fifty-six', 'foo', 2); }
```

In the example above, the `join()` method returns as soon as the thread ends. In addition to waiting for a thread to finish and gathering up any values that the thread might have returned, `join()` also performs any OS cleanup necessary for the thread. That cleanup might be important, especially for long-running programs that spawn lots of threads. If you don't want the return values and don't want to wait for the thread to finish, you should call the `detach()` method instead, as described next.

NOTE: In the example above, the thread returns a list, thus necessitating that the thread creation call be made in list context (i.e., `my ($thr)`). See Section "$thr->join()" in `threads` and Section "THREAD CONTEXT" in `threads` for more details on thread context and return values.

75.7.5 Ignoring A Thread

`join()` does three things: it waits for a thread to exit, cleans up after it, and returns any data the thread may have produced. But what if you're not interested in the thread's return values, and you don't really care when the thread finishes? All you want is for the thread to get cleaned up after when it's done.

In this case, you use the `detach()` method. Once a thread is detached, it'll run until it's finished; then Perl will clean up after it automatically.

```
use threads;

my $thr = threads->create(\&sub1);   # Spawn the thread

$thr->detach();   # Now we officially don't care any more

sleep(15);        # Let thread run for awhile

sub sub1 {
    my $count = 0;
    while (1) {
        $count++;
        print("\$count is $count\n");
        sleep(1);
    }
}
```

Once a thread is detached, it may not be joined, and any return data that it might have produced (if it was done and waiting for a join) is lost.

`detach()` can also be called as a class method to allow a thread to detach itself:

```
    use threads;

    my $thr = threads->create(\&sub1);

    sub sub1 {
        threads->detach();
        # Do more work
    }
```

75.7.6 Process and Thread Termination

With threads one must be careful to make sure they all have a chance to run to completion, assuming that is what you want.

An action that terminates a process will terminate *all* running threads. die() and exit() have this property, and perl does an exit when the main thread exits, perhaps implicitly by falling off the end of your code, even if that's not what you want.

As an example of this case, this code prints the message "Perl exited with active threads: 2 running and unjoined":

```
    use threads;
    my $thr1 = threads->new(\&thrsub, "test1");
    my $thr2 = threads->new(\&thrsub, "test2");
    sub thrsub {
       my ($message) = @_;
       sleep 1;
       print "thread $message\n";
    }
```

But when the following lines are added at the end:

```
    $thr1->join();
    $thr2->join();
```

it prints two lines of output, a perhaps more useful outcome.

75.8 Threads And Data

Now that we've covered the basics of threads, it's time for our next topic: Data. Threading introduces a couple of complications to data access that non-threaded programs never need to worry about.

75.8.1 Shared And Unshared Data

The biggest difference between Perl *ithreads* and the old 5.005 style threading, or for that matter, to most other threading systems out there, is that by default, no data is shared. When a new Perl thread is created, all the data associated with the current thread is copied to the new thread, and is subsequently private to that new thread! This is similar in feel to what happens when a Unix process forks, except that in this case, the data is just copied to a different part of memory within the same process rather than a real fork taking place.

To make use of threading, however, one usually wants the threads to share at least some data between themselves. This is done with the `threads-shared` module and the `:shared` attribute:

```
use threads;
use threads::shared;

my $foo :shared = 1;
my $bar = 1;
threads->create(sub { $foo++; $bar++; })->join();

print("$foo\n");  # Prints 2 since $foo is shared
print("$bar\n");  # Prints 1 since $bar is not shared
```

In the case of a shared array, all the array's elements are shared, and for a shared hash, all the keys and values are shared. This places restrictions on what may be assigned to shared array and hash elements: only simple values or references to shared variables are allowed - this is so that a private variable can't accidentally become shared. A bad assignment will cause the thread to die. For example:

```
use threads;
use threads::shared;

my $var          = 1;
my $svar :shared = 2;
my %hash :shared;

... create some threads ...

$hash{a} = 1;       # All threads see exists($hash{a})
                    # and $hash{a} == 1
$hash{a} = $var;    # okay - copy-by-value: same effect as previous
$hash{a} = $svar;   # okay - copy-by-value: same effect as previous
$hash{a} = \$svar;  # okay - a reference to a shared variable
$hash{a} = \$var;   # This will die
delete($hash{a});   # okay - all threads will see !exists($hash{a})
```

Note that a shared variable guarantees that if two or more threads try to modify it at the same time, the internal state of the variable will not become corrupted. However, there are no guarantees beyond this, as explained in the next section.

75.8.2 Thread Pitfalls: Races

While threads bring a new set of useful tools, they also bring a number of pitfalls. One pitfall is the race condition:

```
use threads;
use threads::shared;

my $x :shared = 1;
my $thr1 = threads->create(\&sub1);
my $thr2 = threads->create(\&sub2);

$thr1->join();
$thr2->join();
print("$x\n");

sub sub1 { my $foo = $x; $x = $foo + 1; }
sub sub2 { my $bar = $x; $x = $bar + 1; }
```

What do you think `$x` will be? The answer, unfortunately, is *it depends.* Both `sub1()` and `sub2()` access the global variable `$x`, once to read and once to write. Depending on factors ranging from your thread implementation's scheduling algorithm to the phase of the moon, `$x` can be 2 or 3.

Race conditions are caused by unsynchronized access to shared data. Without explicit synchronization, there's no way to be sure that nothing has happened to the shared data between the time you access it and the time you update it. Even this simple code fragment has the possibility of error:

```
use threads;
my $x :shared = 2;
my $y :shared;
my $z :shared;
my $thr1 = threads->create(sub { $y = $x; $x = $y + 1; });
my $thr2 = threads->create(sub { $z = $x; $x = $z + 1; });
$thr1->join();
$thr2->join();
```

Two threads both access `$x`. Each thread can potentially be interrupted at any point, or be executed in any order. At the end, `$x` could be 3 or 4, and both `$y` and `$z` could be 2 or 3.

Even `$x += 5` or `$x++` are not guaranteed to be atomic.

Whenever your program accesses data or resources that can be accessed by other threads, you must take steps to coordinate access or risk data inconsistency and race conditions. Note that Perl will protect its internals from your race conditions, but it won't protect you from you.

75.9 Synchronization and control

Perl provides a number of mechanisms to coordinate the interactions between themselves and their data, to avoid race conditions and the like. Some of these are designed to resemble the common techniques used in thread libraries such as pthreads; others are Perl-specific. Often, the standard techniques are clumsy and difficult to get right (such as condition waits). Where possible, it is usually easier to use Perlish techniques such as queues, which remove some of the hard work involved.

75.9.1 Controlling access: lock()

The lock() function takes a shared variable and puts a lock on it. No other thread may lock the variable until the variable is unlocked by the thread holding the lock. Unlocking happens automatically when the locking thread exits the block that contains the call to the lock() function. Using lock() is straightforward: This example has several threads doing some calculations in parallel, and occasionally updating a running total:

```
use threads;
use threads::shared;

my $total :shared = 0;

sub calc {
    while (1) {
        my $result;
        # (... do some calculations and set $result ...)
        {
            lock($total);  # Block until we obtain the lock
            $total += $result;
        } # Lock implicitly released at end of scope
        last if $result == 0;
    }
}

my $thr1 = threads->create(\&calc);
my $thr2 = threads->create(\&calc);
my $thr3 = threads->create(\&calc);
$thr1->join();
$thr2->join();
$thr3->join();
print("total=$total\n");
```

lock() blocks the thread until the variable being locked is available. When lock() returns, your thread can be sure that no other thread can lock that variable until the block containing the lock exits.

It's important to note that locks don't prevent access to the variable in question, only lock attempts. This is in keeping with Perl's longstanding tradition of courteous programming, and the advisory file locking that flock() gives you.

You may lock arrays and hashes as well as scalars. Locking an array, though, will not block subsequent locks on array elements, just lock attempts on the array itself.

Locks are recursive, which means it's okay for a thread to lock a variable more than once. The lock will last until the outermost lock() on the variable goes out of scope. For example:

```
my $x :shared;
doit();

sub doit {
    {
        {
            lock($x); # Wait for lock
            lock($x); # NOOP - we already have the lock
            {
                lock($x); # NOOP
                {
                    lock($x); # NOOP
                    lockit_some_more();
                }
            }
        } # *** Implicit unlock here ***
    }
}

sub lockit_some_more {
    lock($x); # NOOP
} # Nothing happens here
```

Note that there is no `unlock()` function - the only way to unlock a variable is to allow it to go out of scope.

A lock can either be used to guard the data contained within the variable being locked, or it can be used to guard something else, like a section of code. In this latter case, the variable in question does not hold any useful data, and exists only for the purpose of being locked. In this respect, the variable behaves like the mutexes and basic semaphores of traditional thread libraries.

75.9.2 A Thread Pitfall: Deadlocks

Locks are a handy tool to synchronize access to data, and using them properly is the key to safe shared data. Unfortunately, locks aren't without their dangers, especially when multiple locks are involved. Consider the following code:

```
use threads;

my $x :shared = 4;
my $y :shared = 'foo';
my $thr1 = threads->create(sub {
    lock($x);
    sleep(20);
    lock($y);
});
my $thr2 = threads->create(sub {
    lock($y);
    sleep(20);
```

```
        lock($x);
    });
```

This program will probably hang until you kill it. The only way it won't hang is if one of the two threads acquires both locks first. A guaranteed-to-hang version is more complicated, but the principle is the same.

The first thread will grab a lock on `$x`, then, after a pause during which the second thread has probably had time to do some work, try to grab a lock on `$y`. Meanwhile, the second thread grabs a lock on `$y`, then later tries to grab a lock on `$x`. The second lock attempt for both threads will block, each waiting for the other to release its lock.

This condition is called a deadlock, and it occurs whenever two or more threads are trying to get locks on resources that the others own. Each thread will block, waiting for the other to release a lock on a resource. That never happens, though, since the thread with the resource is itself waiting for a lock to be released.

There are a number of ways to handle this sort of problem. The best way is to always have all threads acquire locks in the exact same order. If, for example, you lock variables `$x`, `$y`, and `$z`, always lock `$x` before `$y`, and `$y` before `$z`. It's also best to hold on to locks for as short a period of time to minimize the risks of deadlock.

The other synchronization primitives described below can suffer from similar problems.

75.9.3 Queues: Passing Data Around

A queue is a special thread-safe object that lets you put data in one end and take it out the other without having to worry about synchronization issues. They're pretty straightforward, and look like this:

```
    use threads;
    use Thread::Queue;

    my $DataQueue = Thread::Queue->new();
    my $thr = threads->create(sub {
        while (my $DataElement = $DataQueue->dequeue()) {
            print("Popped $DataElement off the queue\n");
        }
    });

    $DataQueue->enqueue(12);
    $DataQueue->enqueue("A", "B", "C");
    sleep(10);
    $DataQueue->enqueue(undef);
    $thr->join();
```

You create the queue with `Thread::Queue->new()`. Then you can add lists of scalars onto the end with `enqueue()`, and pop scalars off the front of it with `dequeue()`. A queue has no fixed size, and can grow as needed to hold everything pushed on to it.

If a queue is empty, `dequeue()` blocks until another thread enqueues something. This makes queues ideal for event loops and other communications between threads.

75.9.4 Semaphores: Synchronizing Data Access

Semaphores are a kind of generic locking mechanism. In their most basic form, they behave very much like lockable scalars, except that they can't hold data, and that they must be explicitly unlocked. In their advanced form, they act like a kind of counter, and can allow multiple threads to have the *lock* at any one time.

75.9.5 Basic semaphores

Semaphores have two methods, `down()` and `up()`: `down()` decrements the resource count, while `up()` increments it. Calls to `down()` will block if the semaphore's current count would decrement below zero. This program gives a quick demonstration:

```
use threads;
use Thread::Semaphore;

my $semaphore = Thread::Semaphore->new();
my $GlobalVariable :shared = 0;

$thr1 = threads->create(\&sample_sub, 1);
$thr2 = threads->create(\&sample_sub, 2);
$thr3 = threads->create(\&sample_sub, 3);

sub sample_sub {
    my $SubNumber = shift(@_);
    my $TryCount = 10;
    my $LocalCopy;
    sleep(1);
    while ($TryCount--) {
        $semaphore->down();
        $LocalCopy = $GlobalVariable;
        print("$TryCount tries left for sub $SubNumber "
             ."(\$GlobalVariable is $GlobalVariable)\n");
        sleep(2);
        $LocalCopy++;
        $GlobalVariable = $LocalCopy;
        $semaphore->up();
    }
}

$thr1->join();
$thr2->join();
$thr3->join();
```

The three invocations of the subroutine all operate in sync. The semaphore, though, makes sure that only one thread is accessing the global variable at once.

75.9.6 Advanced Semaphores

By default, semaphores behave like locks, letting only one thread `down()` them at a time. However, there are other uses for semaphores.

Each semaphore has a counter attached to it. By default, semaphores are created with the counter set to one, down() decrements the counter by one, and up() increments by one. However, we can override any or all of these defaults simply by passing in different values:

```
use threads;
use Thread::Semaphore;

my $semaphore = Thread::Semaphore->new(5);
                # Creates a semaphore with the counter set to five

my $thr1 = threads->create(\&sub1);
my $thr2 = threads->create(\&sub1);

sub sub1 {
    $semaphore->down(5); # Decrements the counter by five
    # Do stuff here
    $semaphore->up(5); # Increment the counter by five
}

$thr1->detach();
$thr2->detach();
```

If down() attempts to decrement the counter below zero, it blocks until the counter is large enough. Note that while a semaphore can be created with a starting count of zero, any up() or down() always changes the counter by at least one, and so $semaphore->down(0) is the same as $semaphore->down(1).

The question, of course, is why would you do something like this? Why create a semaphore with a starting count that's not one, or why decrement or increment it by more than one? The answer is resource availability. Many resources that you want to manage access for can be safely used by more than one thread at once.

For example, let's take a GUI driven program. It has a semaphore that it uses to synchronize access to the display, so only one thread is ever drawing at once. Handy, but of course you don't want any thread to start drawing until things are properly set up. In this case, you can create a semaphore with a counter set to zero, and up it when things are ready for drawing.

Semaphores with counters greater than one are also useful for establishing quotas. Say, for example, that you have a number of threads that can do I/O at once. You don't want all the threads reading or writing at once though, since that can potentially swamp your I/O channels, or deplete your process's quota of filehandles. You can use a semaphore initialized to the number of concurrent I/O requests (or open files) that you want at any one time, and have your threads quietly block and unblock themselves.

Larger increments or decrements are handy in those cases where a thread needs to check out or return a number of resources at once.

75.9.7 Waiting for a Condition

The functions cond_wait() and cond_signal() can be used in conjunction with locks to notify co-operating threads that a resource has become available. They are very similar in use to the functions found in pthreads. However for most purposes, queues are simpler to use and more intuitive. See threads-shared for more details.

75.9.8 Giving up control

There are times when you may find it useful to have a thread explicitly give up the CPU to another thread. You may be doing something processor-intensive and want to make sure that the user-interface thread gets called frequently. Regardless, there are times that you might want a thread to give up the processor.

Perl's threading package provides the yield() function that does this. yield() is pretty straightforward, and works like this:

```
use threads;

sub loop {
    my $thread = shift;
    my $foo = 50;
    while($foo--) { print("In thread $thread\n"); }
    threads->yield();
    $foo = 50;
    while($foo--) { print("In thread $thread\n"); }
}

my $thr1 = threads->create(\&loop, 'first');
my $thr2 = threads->create(\&loop, 'second');
my $thr3 = threads->create(\&loop, 'third');
```

It is important to remember that yield() is only a hint to give up the CPU, it depends on your hardware, OS and threading libraries what actually happens. **On many operating systems, yield() is a no-op.** Therefore it is important to note that one should not build the scheduling of the threads around yield() calls. It might work on your platform but it won't work on another platform.

75.10 General Thread Utility Routines

We've covered the workhorse parts of Perl's threading package, and with these tools you should be well on your way to writing threaded code and packages. There are a few useful little pieces that didn't really fit in anyplace else.

75.10.1 What Thread Am I In?

The threads->self() class method provides your program with a way to get an object representing the thread it's currently in. You can use this object in the same way as the ones returned from thread creation.

75.10.2 Thread IDs

tid() is a thread object method that returns the thread ID of the thread the object represents. Thread IDs are integers, with the main thread in a program being 0. Currently Perl assigns a unique TID to every thread ever created in your program, assigning the first thread to be created a TID of 1, and increasing the TID by 1 for each new thread that's created. When used as a class method, `threads->tid()` can be used by a thread to get its own TID.

75.10.3 Are These Threads The Same?

The `equal()` method takes two thread objects and returns true if the objects represent the same thread, and false if they don't.

Thread objects also have an overloaded `==` comparison so that you can do comparison on them as you would with normal objects.

75.10.4 What Threads Are Running?

`threads->list()` returns a list of thread objects, one for each thread that's currently running and not detached. Handy for a number of things, including cleaning up at the end of your program (from the main Perl thread, of course):

```
# Loop through all the threads
foreach my $thr (threads->list()) {
    $thr->join();
}
```

If some threads have not finished running when the main Perl thread ends, Perl will warn you about it and die, since it is impossible for Perl to clean up itself while other threads are running.

NOTE: The main Perl thread (thread 0) is in a *detached* state, and so does not appear in the list returned by `threads->list()`.

75.11 A Complete Example

Confused yet? It's time for an example program to show some of the things we've covered. This program finds prime numbers using threads.

```
#!/usr/bin/perl
# prime-pthread, courtesy of Tom Christiansen

use strict;
use warnings;

use threads;
use Thread::Queue;

sub check_num {
    my ($upstream, $cur_prime) = @_;
    my $kid;
    my $downstream = Thread::Queue->new();
    while (my $num = $upstream->dequeue()) {
        next unless ($num % $cur_prime);
        if ($kid) {
            $downstream->enqueue($num);
        } else {
            print("Found prime: $num\n");
20             $kid = threads->create(\&check_num, $downstream, $num);
21             if (! $kid) {
22                 warn("Sorry.  Ran out of threads.\n");
23                 last;
24             }
25         }
26     }
27     if ($kid) {
28         $downstream->enqueue(undef);
29         $kid->join();
30     }
31 }
32
33 my $stream = Thread::Queue->new(3..1000, undef);
34 check_num($stream, 2);
```

This program uses the pipeline model to generate prime numbers. Each thread in the pipeline has an input queue that feeds numbers to be checked, a prime number that it's responsible for, and an output queue into which it funnels numbers that have failed the check. If the thread has a number that's failed its check and there's no child thread, then the thread must have found a new prime number. In that case, a new child thread is created for that prime and stuck on the end of the pipeline.

This probably sounds a bit more confusing than it really is, so let's go through this program piece by piece and see what it does. (For those of you who might be trying to remember exactly what a prime number is, it's a number that's only evenly divisible by itself and 1.)

The bulk of the work is done by the `check_num()` subroutine, which takes a reference to its input queue and a prime number that it's responsible for. After pulling in the input queue and the prime that the subroutine is checking (line 11), we create a new queue (line 13) and reserve a scalar for the thread that we're likely to create later (line 12).

The while loop from line 14 to line 26 grabs a scalar off the input queue and checks against the prime this thread is responsible for. Line 15 checks to see if there's a remainder when we divide the number to be checked by our prime. If there is one, the number must not be evenly divisible by our prime, so we need to either pass it on to the next thread if we've created one (line 17) or create a new thread if we haven't.

The new thread creation is line 20. We pass on to it a reference to the queue we've created, and the prime number we've found. In lines 21 through 24, we check to make sure that our new thread got created, and if not, we stop checking any remaining numbers in the queue.

Finally, once the loop terminates (because we got a 0 or `undef` in the queue, which serves as a note to terminate), we pass on the notice to our child, and wait for it to exit if we've created a child (lines 27 and 30).

Meanwhile, back in the main thread, we first create a queue (line 33) and queue up all the numbers from 3 to 1000 for checking, plus a termination notice. Then all we have to do to get the ball rolling is pass the queue and the first prime to the `check_num()` subroutine (line 34).

That's how it works. It's pretty simple; as with many Perl programs, the explanation is much longer than the program.

75.12 Different implementations of threads

Some background on thread implementations from the operating system viewpoint. There are three basic categories of threads: user-mode threads, kernel threads, and multiprocessor kernel threads.

User-mode threads are threads that live entirely within a program and its libraries. In this model, the OS knows nothing about threads. As far as it's concerned, your process is just a process.

This is the easiest way to implement threads, and the way most OSes start. The big disadvantage is that, since the OS knows nothing about threads, if one thread blocks they all do. Typical blocking activities include most system calls, most I/O, and things like `sleep()`.

Kernel threads are the next step in thread evolution. The OS knows about kernel threads, and makes allowances for them. The main difference between a kernel thread and a user-mode thread is blocking. With kernel threads, things that block a single thread don't block other threads. This is not the case with user-mode threads, where the kernel blocks at the process level and not the thread level.

This is a big step forward, and can give a threaded program quite a performance boost over non-threaded programs. Threads that block performing I/O, for example, won't block threads that are doing other things. Each process still has only one thread running at once, though, regardless of how many CPUs a system might have.

Since kernel threading can interrupt a thread at any time, they will uncover some of the implicit locking assumptions you may make in your program. For example, something as simple as `$x = $x + 2` can behave unpredictably with kernel threads if `$x` is visible to other threads, as another thread may have changed `$x` between the time it was fetched on the right hand side and the time the new value is stored.

Multiprocessor kernel threads are the final step in thread support. With multiprocessor kernel threads on a machine with multiple CPUs, the OS may schedule two or more threads to run simultaneously on different CPUs.

This can give a serious performance boost to your threaded program, since more than one thread will be executing at the same time. As a tradeoff, though, any of those nagging synchronization issues that might not have shown with basic kernel threads will appear with a vengeance.

In addition to the different levels of OS involvement in threads, different OSes (and different thread implementations for a particular OS) allocate CPU cycles to threads in different ways.

Cooperative multitasking systems have running threads give up control if one of two things happen. If a thread calls a yield function, it gives up control. It also gives up control if the thread does something that would cause it to block, such as perform I/O. In a cooperative multitasking implementation, one thread can starve all the others for CPU time if it so chooses.

Preemptive multitasking systems interrupt threads at regular intervals while the system decides which thread should run next. In a preemptive multitasking system, one thread usually won't monopolize the CPU.

On some systems, there can be cooperative and preemptive threads running simultaneously. (Threads running with realtime priorities often behave cooperatively, for example, while threads running at normal priorities behave preemptively.)

Most modern operating systems support preemptive multitasking nowadays.

75.13 Performance considerations

The main thing to bear in mind when comparing Perl's *ithreads* to other threading models is the fact that for each new thread created, a complete copy of all the variables and data of the parent thread has to be taken. Thus, thread creation can be quite expensive, both in terms of memory usage and time spent in creation. The ideal way to reduce these costs is to have a relatively short number of long-lived threads, all created fairly early on (before the base thread has accumulated too much data). Of course, this may not always be possible, so compromises have to be made. However, after a thread has been created, its performance and extra memory usage should be little different than ordinary code.

Also note that under the current implementation, shared variables use a little more memory and are a little slower than ordinary variables.

75.14 Process-scope Changes

Note that while threads themselves are separate execution threads and Perl data is thread-private unless explicitly shared, the threads can affect process-scope state, affecting all the threads.

The most common example of this is changing the current working directory using `chdir()`. One thread calls `chdir()`, and the working directory of all the threads changes.

Even more drastic example of a process-scope change is `chroot()`: the root directory of all the threads changes, and no thread can undo it (as opposed to `chdir()`).

Further examples of process-scope changes include `umask()` and changing uids and gids.

Thinking of mixing `fork()` and threads? Please lie down and wait until the feeling passes. Be aware that the semantics of `fork()` vary between platforms. For example, some Unix systems copy all the current threads into the child process, while others only copy the thread that called `fork()`. You have been warned!

Similarly, mixing signals and threads may be problematic. Implementations are platform-dependent, and even the POSIX semantics may not be what you expect (and Perl doesn't even give you the full POSIX API). For example, there is no way to guarantee that a signal sent to a multi-threaded Perl application will get intercepted by any particular thread. (However, a recently added feature does provide the capability to send signals between threads. See Section "THREAD SIGNALLING" in `threads` for more details.)

75.15 Thread-Safety of System Libraries

Whether various library calls are thread-safe is outside the control of Perl. Calls often suffering from not being thread-safe include: `localtime()`, `gmtime()`, functions fetching user, group and network information (such as `getgrent()`, `gethostent()`, `getnetent()` and so on), `readdir()`, `rand()`, and `srand()`. In general, calls that depend on some global external state.

If the system Perl is compiled in has thread-safe variants of such calls, they will be used. Beyond that, Perl is at the mercy of the thread-safety or -unsafety of the calls. Please consult your C library call documentation.

On some platforms the thread-safe library interfaces may fail if the result buffer is too small (for example the user group databases may be rather large, and the reentrant interfaces may have to carry around a full snapshot of those databases). Perl will start with a small buffer, but keep retrying and growing the result buffer until the result fits. If this limitless growing sounds bad for security or memory consumption reasons you can recompile Perl with `PERL_REENTRANT_MAXSIZE` defined to the maximum number of bytes you will allow.

75.16 Conclusion

A complete thread tutorial could fill a book (and has, many times), but with what we've covered in this introduction, you should be well on your way to becoming a threaded Perl expert.

75.17 SEE ALSO

Annotated POD for `threads`: `http://annocpan.org/?mode=search&field=Module&name=threads`

Latest version of `threads` on CPAN: `http://search.cpan.org/search?module=threads`

Annotated POD for `threads-shared`: `http://annocpan.org/?mode=search&field=Module&name=threads%3A%3Ashared`

Latest version of `threads-shared` on CPAN: `http://search.cpan.org/search?module=threads%3A%3Ashared`

Perl threads mailing list: `http://lists.perl.org/list/ithreads.html`

75.18 Bibliography

Here's a short bibliography courtesy of Jürgen Christoffel:

75.18.1 Introductory Texts

Birrell, Andrew D. An Introduction to Programming with Threads. Digital Equipment Corporation, 1989, DEC-SRC Research Report #35 online as ftp://ftp.dec.com/pub/DEC/SRC/research-reports/SRC-035.pdf (highly recommended)

Robbins, Kay. A., and Steven Robbins. Practical Unix Programming: A Guide to Concurrency, Communication, and Multithreading. Prentice-Hall, 1996.

Lewis, Bill, and Daniel J. Berg. Multithreaded Programming with Pthreads. Prentice Hall, 1997, ISBN 0-13-443698-9 (a well-written introduction to threads).

Nelson, Greg (editor). Systems Programming with Modula-3. Prentice Hall, 1991, ISBN 0-13-590464-1.

Nichols, Bradford, Dick Buttlar, and Jacqueline Proulx Farrell. Pthreads Programming. O'Reilly & Associates, 1996, ISBN 156592-115-1 (covers POSIX threads).

75.18.2 OS-Related References

Boykin, Joseph, David Kirschen, Alan Langerman, and Susan LoVerso. Programming under Mach. Addison-Wesley, 1994, ISBN 0-201-52739-1.

Tanenbaum, Andrew S. Distributed Operating Systems. Prentice Hall, 1995, ISBN 0-13-219908-4 (great textbook).

Silberschatz, Abraham, and Peter B. Galvin. Operating System Concepts, 4th ed. Addison-Wesley, 1995, ISBN 0-201-59292-4

75.18.3 Other References

Arnold, Ken and James Gosling. The Java Programming Language, 2nd ed. Addison-Wesley, 1998, ISBN 0-201-31006-6.

comp.programming.threads FAQ, `http://www.serpentine.com/~bos/threads-faq/`

Le Sergent, T. and B. Berthomieu. "Incremental MultiThreaded Garbage Collection on Virtually Shared Memory Architectures" in Memory Management: Proc. of the International Workshop IWMM 92, St. Malo, France, September 1992, Yves Bekkers and Jacques Cohen, eds. Springer, 1992, ISBN 3540-55940-X (real-life thread applications).

Artur Bergman, "Where Wizards Fear To Tread", June 11, 2002, `http://www.perl.com/pub/a/2002/06/11/threads.html`

75.19 Acknowledgements

Thanks (in no particular order) to Chaim Frenkel, Steve Fink, Gurusamy Sarathy, Ilya Zakharevich, Benjamin Sugars, Jürgen Christoffel, Joshua Pritikin, and Alan Burlison, for their help in reality-checking and polishing this article. Big thanks to Tom Christiansen for his rewrite of the prime number generator.

75.20 AUTHOR

Dan Sugalski <dan@sidhe.org<gt>

Slightly modified by Arthur Bergman to fit the new thread model/module.

Reworked slightly by Jörg Walter <jwalt@cpan.org<gt> to be more concise about thread-safety of Perl code.

Rearranged slightly by Elizabeth Mattijsen <liz@dijkmat.nl<gt> to put less emphasis on yield().

75.21 Copyrights

76 perltie

76.1 NAME

perltie - how to hide an object class in a simple variable

76.2 SYNOPSIS

```
tie VARIABLE, CLASSNAME, LIST

$object = tied VARIABLE

untie VARIABLE
```

76.3 DESCRIPTION

Prior to release 5.0 of Perl, a programmer could use dbmopen() to connect an on-disk database in the standard Unix dbm(3x) format magically to a %HASH in their program. However, their Perl was either built with one particular dbm library or another, but not both, and you couldn't extend this mechanism to other packages or types of variables.

Now you can.

The tie() function binds a variable to a class (package) that will provide the implementation for access methods for that variable. Once this magic has been performed, accessing a tied variable automatically triggers method calls in the proper class. The complexity of the class is hidden behind magic methods calls. The method names are in ALL CAPS, which is a convention that Perl uses to indicate that they're called implicitly rather than explicitly–just like the BEGIN() and END() functions.

In the tie() call, `VARIABLE` is the name of the variable to be enchanted. `CLASSNAME` is the name of a class implementing objects of the correct type. Any additional arguments in the `LIST` are passed to the appropriate constructor method for that class–meaning TIESCALAR(), TIEARRAY(), TIEHASH(), or TIEHANDLE(). (Typically these are arguments such as might be passed to the dbminit() function of C.) The object returned by the "new" method is also returned by the tie() function, which would be useful if you wanted to access other methods in `CLASSNAME`. (You don't actually have to return a reference to a right "type" (e.g., HASH or `CLASSNAME`) so long as it's a properly blessed object.) You can also retrieve a reference to the underlying object using the tied() function.

Unlike dbmopen(), the tie() function will not `use` or `require` a module for you–you need to do that explicitly yourself.

76.3.1 Tying Scalars

A class implementing a tied scalar should define the following methods: TIESCALAR, FETCH, STORE, and possibly UNTIE and/or DESTROY.

Let's look at each in turn, using as an example a tie class for scalars that allows the user to do something like:

```
    tie $his_speed, 'Nice', getppid();
    tie $my_speed,  'Nice', $$;
```

And now whenever either of those variables is accessed, its current system priority is retrieved and returned. If those variables are set, then the process's priority is changed!

We'll use Jarkko Hietaniemi `<jhi@iki.fi>`'s BSD::Resource class (not included) to access the PRIO_PROCESS, PRIO_MIN, and PRIO_MAX constants from your system, as well as the getpriority() and setpriority() system calls. Here's the preamble of the class.

```
package Nice;
use Carp;
use BSD::Resource;
use strict;
$Nice::DEBUG = 0 unless defined $Nice::DEBUG;
```

TIESCALAR classname, LIST

This is the constructor for the class. That means it is expected to return a blessed reference to a new scalar (probably anonymous) that it's creating. For example:

```
sub TIESCALAR {
    my $class = shift;
    my $pid = shift || $$; # 0 means me

    if ($pid !~ /^\d+$/) {
        carp "Nice::Tie::Scalar got non-numeric pid $pid" if $^W;
        return undef;
    }

    unless (kill 0, $pid) { # EPERM or ERSCH, no doubt
        carp "Nice::Tie::Scalar got bad pid $pid: $!" if $^W;
        return undef;
    }

    return bless \$pid, $class;
}
```

This tie class has chosen to return an error rather than raising an exception if its constructor should fail. While this is how dbmopen() works, other classes may well not wish to be so forgiving. It checks the global variable `$^W` to see whether to emit a bit of noise anyway.

FETCH this

This method will be triggered every time the tied variable is accessed (read). It takes no arguments beyond its self reference, which is the object representing the scalar we're dealing with. Because in this case we're using just a SCALAR ref for the tied scalar object, a simple $$self allows the method to get at the real value stored there. In our example below, that real value is the process ID to which we've tied our variable.

```
sub FETCH {
    my $self = shift;
    confess "wrong type" unless ref $self;
    croak "usage error" if @_;
```

```
        my $nicety;
        local($!) = 0;
        $nicety = getpriority(PRIO_PROCESS, $$self);
        if ($!) { croak "getpriority failed: $!" }
        return $nicety;
    }
```

This time we've decided to blow up (raise an exception) if the renice fails–there's no place for us to return an error otherwise, and it's probably the right thing to do.

STORE this, value

This method will be triggered every time the tied variable is set (assigned). Beyond its self reference, it also expects one (and only one) argument: the new value the user is trying to assign. Don't worry about returning a value from STORE; the semantic of assignment returning the assigned value is implemented with FETCH.

```
    sub STORE {
        my $self = shift;
        confess "wrong type" unless ref $self;
        my $new_nicety = shift;
        croak "usage error" if @_;

        if ($new_nicety < PRIO_MIN) {
            carp sprintf
              "WARNING: priority %d less than minimum system priority %d",
                  $new_nicety, PRIO_MIN if $^W;
            $new_nicety = PRIO_MIN;
        }

        if ($new_nicety > PRIO_MAX) {
            carp sprintf
              "WARNING: priority %d greater than maximum system priority %d",
                  $new_nicety, PRIO_MAX if $^W;
            $new_nicety = PRIO_MAX;
        }

        unless (defined setpriority(PRIO_PROCESS, $$self, $new_nicety)) {
            confess "setpriority failed: $!";
        }
    }
```

UNTIE this

This method will be triggered when the `untie` occurs. This can be useful if the class needs to know when no further calls will be made. (Except DESTROY of course.) See Section 76.3.6 [The untie Gotcha], page 1305 below for more details.

DESTROY this

This method will be triggered when the tied variable needs to be destructed. As with other object classes, such a method is seldom necessary, because Perl deallocates its moribund object's memory for you automatically–this isn't C++, you know. We'll use a DESTROY method here for debugging purposes only.

```
sub DESTROY {
    my $self = shift;
    confess "wrong type" unless ref $self;
    carp "[ Nice::DESTROY pid $$self ]" if $Nice::DEBUG;
}
```

That's about all there is to it. Actually, it's more than all there is to it, because we've done a few nice things here for the sake of completeness, robustness, and general aesthetics. Simpler TIESCALAR classes are certainly possible.

76.3.2 Tying Arrays

A class implementing a tied ordinary array should define the following methods: TIEARRAY, FETCH, STORE, FETCHSIZE, STORESIZE, CLEAR and perhaps UNTIE and/or DESTROY.

FETCHSIZE and STORESIZE are used to provide `$#array` and equivalent `scalar(@array)` access.

The methods POP, PUSH, SHIFT, UNSHIFT, SPLICE, DELETE, and EXISTS are required if the perl operator with the corresponding (but lowercase) name is to operate on the tied array. The **Tie::Array** class can be used as a base class to implement the first five of these in terms of the basic methods above. The default implementations of DELETE and EXISTS in **Tie::Array** simply `croak`.

In addition EXTEND will be called when perl would have pre-extended allocation in a real array.

For this discussion, we'll implement an array whose elements are a fixed size at creation. If you try to create an element larger than the fixed size, you'll take an exception. For example:

```
use FixedElem_Array;
tie @array, 'FixedElem_Array', 3;
$array[0] = 'cat';  # ok.
$array[1] = 'dogs'; # exception, length('dogs') > 3.
```

The preamble code for the class is as follows:

```
package FixedElem_Array;
use Carp;
use strict;
```

TIEARRAY classname, LIST

This is the constructor for the class. That means it is expected to return a blessed reference through which the new array (probably an anonymous ARRAY ref) will be accessed.

In our example, just to show you that you don't *really* have to return an ARRAY reference, we'll choose a HASH reference to represent our object. A HASH works out well as a generic record type: the `{ELEMSIZE}` field will store the maximum element size allowed, and the `{ARRAY}` field will hold the true ARRAY ref. If someone outside the class tries to dereference the object returned (doubtless thinking it an ARRAY ref), they'll blow up. This just goes to show you that you should respect an object's privacy.

```
sub TIEARRAY {
  my $class    = shift;
  my $elemsize = shift;
  if ( @_ || $elemsize =~ /\D/ ) {
    croak "usage: tie ARRAY, ’" . __PACKAGE__ . "’, elem_size";
  }
  return bless {
    ELEMSIZE => $elemsize,
    ARRAY    => [],
  }, $class;
}
```

FETCH this, index

This method will be triggered every time an individual element the tied array is accessed (read). It takes one argument beyond its self reference: the index whose value we're trying to fetch.

```
sub FETCH {
  my $self  = shift;
  my $index = shift;
  return $self->{ARRAY}->[$index];
}
```

If a negative array index is used to read from an array, the index will be translated to a positive one internally by calling FETCHSIZE before being passed to FETCH. You may disable this feature by assigning a true value to the variable `$NEGATIVE_INDICES` in the tied array class.

As you may have noticed, the name of the FETCH method (et al.) is the same for all accesses, even though the constructors differ in names (TIESCALAR vs TIEARRAY). While in theory you could have the same class servicing several tied types, in practice this becomes cumbersome, and it's easiest to keep them at simply one tie type per class.

STORE this, index, value

This method will be triggered every time an element in the tied array is set (written). It takes two arguments beyond its self reference: the index at which we're trying to store something and the value we're trying to put there.

In our example, `undef` is really `$self->{ELEMSIZE}` number of spaces so we have a little more work to do here:

```
sub STORE {
  my $self = shift;
  my( $index, $value ) = @_;
  if ( length $value > $self->{ELEMSIZE} ) {
    croak "length of $value is greater than $self->{ELEMSIZE}";
  }
  # fill in the blanks
  $self->EXTEND( $index ) if $index > $self->FETCHSIZE();
  # right justify to keep element size for smaller elements
  $self->{ARRAY}->[$index] = sprintf "%$self->{ELEMSIZE}s", $value;
}
```

Negative indexes are treated the same as with FETCH.

FETCHSIZE this

Returns the total number of items in the tied array associated with object *this*. (Equivalent to `scalar(@array)`). For example:

```
  sub FETCHSIZE {
    my $self = shift;
    return scalar @{$self->{ARRAY}};
  }
```

STORESIZE this, count

Sets the total number of items in the tied array associated with object *this* to be *count*. If this makes the array larger then class's mapping of `undef` should be returned for new positions. If the array becomes smaller then entries beyond count should be deleted.

In our example, 'undef' is really an element containing `$self->{ELEMSIZE}` number of spaces. Observe:

```
  sub STORESIZE {
    my $self  = shift;
    my $count = shift;
    if ( $count > $self->FETCHSIZE() ) {
      foreach ( $count - $self->FETCHSIZE() .. $count ) {
        $self->STORE( $_, '' );
      }
    } elsif ( $count < $self->FETCHSIZE() ) {
      foreach ( 0 .. $self->FETCHSIZE() - $count - 2 ) {
        $self->POP();
      }
    }
  }
```

EXTEND this, count

Informative call that array is likely to grow to have *count* entries. Can be used to optimize allocation. This method need do nothing.

In our example, we want to make sure there are no blank (`undef`) entries, so `EXTEND` will make use of `STORESIZE` to fill elements as needed:

```
  sub EXTEND {
    my $self  = shift;
    my $count = shift;
    $self->STORESIZE( $count );
  }
```

EXISTS this, key

Verify that the element at index *key* exists in the tied array *this*.

In our example, we will determine that if an element consists of `$self->{ELEMSIZE}` spaces only, it does not exist:

```
sub EXISTS {
  my $self  = shift;
  my $index = shift;
  return 0 if ! defined $self->{ARRAY}->[$index] ||
              $self->{ARRAY}->[$index] eq ’ ’ x $self->{ELEMSIZE};
  return 1;
}
```

DELETE this, key

Delete the element at index *key* from the tied array *this*.

In our example, a deleted item is `$self->{ELEMSIZE}` spaces:

```
sub DELETE {
  my $self  = shift;
  my $index = shift;
  return $self->STORE( $index, ’’ );
}
```

CLEAR this

Clear (remove, delete, ...) all values from the tied array associated with object *this*. For example:

```
sub CLEAR {
  my $self = shift;
  return $self->{ARRAY} = [];
}
```

PUSH this, LIST

Append elements of *LIST* to the array. For example:

```
sub PUSH {
  my $self = shift;
  my @list = @_;
  my $last = $self->FETCHSIZE();
  $self->STORE( $last + $_, $list[$_] ) foreach 0 .. $#list;
  return $self->FETCHSIZE();
}
```

POP this

Remove last element of the array and return it. For example:

```
sub POP {
  my $self = shift;
  return pop @{$self->{ARRAY}};
}
```

SHIFT this

Remove the first element of the array (shifting other elements down) and return it. For example:

```
sub SHIFT {
  my $self = shift;
  return shift @{$self->{ARRAY}};
}
```

UNSHIFT this, LIST

Insert LIST elements at the beginning of the array, moving existing elements up to make room. For example:

```
sub UNSHIFT {
  my $self = shift;
  my @list = @_;
  my $size = scalar( @list );
  # make room for our list
  @{$self->{ARRAY}}[ $size .. $#{$self->{ARRAY}} + $size ]
   = @{$self->{ARRAY}};
  $self->STORE( $_, $list[$_] ) foreach 0 .. $#list;
}
```

SPLICE this, offset, length, LIST

Perform the equivalent of `splice` on the array.

offset is optional and defaults to zero, negative values count back from the end of the array.

length is optional and defaults to rest of the array.

LIST may be empty.

Returns a list of the original *length* elements at *offset*.

In our example, we'll use a little shortcut if there is a *LIST*:

```
sub SPLICE {
  my $self   = shift;
  my $offset = shift || 0;
  my $length = shift || $self->FETCHSIZE() - $offset;
  my @list   = ();
  if ( @_ ) {
    tie @list, __PACKAGE__, $self->{ELEMSIZE};
    @list   = @_;
  }
  return splice @{$self->{ARRAY}}, $offset, $length, @list;
}
```

UNTIE this

Will be called when `untie` happens. (See Section 76.3.6 [The untie Gotcha], page 1305 below.)

DESTROY this

This method will be triggered when the tied variable needs to be destructed. As with the scalar tie class, this is almost never needed in a language that does its own garbage collection, so this time we'll just leave it out.

76.3.3 Tying Hashes

Hashes were the first Perl data type to be tied (see dbmopen()). A class implementing a tied hash should define the following methods: TIEHASH is the constructor. FETCH and STORE access the key and value pairs. EXISTS reports whether a key is present in the hash, and DELETE deletes one. CLEAR empties the hash by deleting all the key and value pairs. FIRSTKEY and NEXTKEY implement the keys() and each() functions to iterate over all the keys. SCALAR is triggered when the tied hash is evaluated in scalar context. UNTIE is called when `untie` happens, and DESTROY is called when the tied variable is garbage collected.

If this seems like a lot, then feel free to inherit from merely the standard Tie::StdHash module for most of your methods, redefining only the interesting ones. See `Tie-Hash` for details.

Remember that Perl distinguishes between a key not existing in the hash, and the key existing in the hash but having a corresponding value of `undef`. The two possibilities can be tested with the `exists()` and `defined()` functions.

Here's an example of a somewhat interesting tied hash class: it gives you a hash representing a particular user's dot files. You index into the hash with the name of the file (minus the dot) and you get back that dot file's contents. For example:

```
use DotFiles;
tie %dot, 'DotFiles';
if ( $dot{profile} =~ /MANPATH/ ||
     $dot{login}   =~ /MANPATH/ ||
     $dot{cshrc}   =~ /MANPATH/    )
{
    print "you seem to set your MANPATH\n";
}
```

Or here's another sample of using our tied class:

```
tie %him, 'DotFiles', 'daemon';
foreach $f ( keys %him ) {
    printf "daemon dot file %s is size %d\n",
        $f, length $him{$f};
}
```

In our tied hash DotFiles example, we use a regular hash for the object containing several important fields, of which only the `{LIST}` field will be what the user thinks of as the real hash.

USER

whose dot files this object represents

HOME

where those dot files live

CLOBBER

whether we should try to change or remove those dot files

LIST

the hash of dot file names and content mappings

Here’s the start of `Dotfiles.pm`:

```
package DotFiles;
use Carp;
sub whowasi { (caller(1))[3] . ’()’ }
my $DEBUG = 0;
sub debug { $DEBUG = @_ ? shift : 1 }
```

For our example, we want to be able to emit debugging info to help in tracing during development. We keep also one convenience function around internally to help print out warnings; whowasi() returns the function name that calls it.

Here are the methods for the DotFiles tied hash.

TIEHASH classname, LIST

This is the constructor for the class. That means it is expected to return a blessed reference through which the new object (probably but not necessarily an anonymous hash) will be accessed.

Here’s the constructor:

```
sub TIEHASH {
    my $self = shift;
    my $user = shift || $>;
    my $dotdir = shift || ’’;
    croak "usage: @{[&whowasi]} [USER [DOTDIR]]" if @_;
    $user = getpwuid($user) if $user =˜ /ˆ\d+$/;
    my $dir = (getpwnam($user))[7]
            || croak "@{[&whowasi]}: no user $user";
    $dir .= "/$dotdir" if $dotdir;

    my $node = {
        USER    => $user,
        HOME    => $dir,
        LIST    => {},
        CLOBBER => 0,
    };

    opendir(DIR, $dir)
            || croak "@{[&whowasi]}: can’t opendir $dir: $!";
    foreach $dot ( grep /ˆ\./ && -f "$dir/$_", readdir(DIR)) {
        $dot =˜ s/ˆ\.//;
        $node->{LIST}{$dot} = undef;
    }
    closedir DIR;
    return bless $node, $self;
}
```

It’s probably worth mentioning that if you’re going to filetest the return values out of a readdir, you’d better prepend the directory in question. Otherwise, because we didn’t chdir() there, it would have been testing the wrong file.

FETCH this, key

This method will be triggered every time an element in the tied hash is accessed (read). It takes one argument beyond its self reference: the key whose value we're trying to fetch.

Here's the fetch for our DotFiles example.

```
sub FETCH {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $dir = $self->{HOME};
    my $file = "$dir/.$dot";

    unless (exists $self->{LIST}->{$dot} || -f $file) {
        carp "@{[&whowasi]}: no $dot file" if $DEBUG;
        return undef;
    }

    if (defined $self->{LIST}->{$dot}) {
        return $self->{LIST}->{$dot};
    } else {
        return $self->{LIST}->{$dot} = `cat $dir/.$dot`;
    }
}
```

It was easy to write by having it call the Unix cat(1) command, but it would probably be more portable to open the file manually (and somewhat more efficient). Of course, because dot files are a Unixy concept, we're not that concerned.

STORE this, key, value

This method will be triggered every time an element in the tied hash is set (written). It takes two arguments beyond its self reference: the index at which we're trying to store something, and the value we're trying to put there.

Here in our DotFiles example, we'll be careful not to let them try to overwrite the file unless they've called the clobber() method on the original object reference returned by tie().

```
sub STORE {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $value = shift;
    my $file = $self->{HOME} . "/.$dot";
    my $user = $self->{USER};

    croak "@{[&whowasi]}: $file not clobberable"
        unless $self->{CLOBBER};
    open(my $f, '>', $file) || croak "can't open $file: $!";
    print $f $value;
    close($f);
}
```

If they wanted to clobber something, they might say:

```
    $ob = tie %daemon_dots, 'daemon';
    $ob->clobber(1);
    $daemon_dots{signature} = "A true daemon\n";
```

Another way to lay hands on a reference to the underlying object is to use the tied() function, so they might alternately have set clobber using:

```
    tie %daemon_dots, 'daemon';
    tied(%daemon_dots)->clobber(1);
```

The clobber method is simply:

```
    sub clobber {
        my $self = shift;
        $self->{CLOBBER} = @_ ? shift : 1;
    }
```

DELETE this, key

This method is triggered when we remove an element from the hash, typically by using the delete() function. Again, we'll be careful to check whether they really want to clobber files.

```
    sub DELETE   {
        carp &whowasi if $DEBUG;

        my $self = shift;
        my $dot = shift;
        my $file = $self->{HOME} . "/.$dot";
        croak "@{[&whowasi]}: won't remove file $file"
            unless $self->{CLOBBER};
        delete $self->{LIST}->{$dot};
        my $success = unlink($file);
        carp "@{[&whowasi]}: can't unlink $file: $!" unless $success;
        $success;
    }
```

The value returned by DELETE becomes the return value of the call to delete(). If you want to emulate the normal behavior of delete(), you should return whatever FETCH would have returned for this key. In this example, we have chosen instead to return a value which tells the caller whether the file was successfully deleted.

CLEAR this

This method is triggered when the whole hash is to be cleared, usually by assigning the empty list to it.

In our example, that would remove all the user's dot files! It's such a dangerous thing that they'll have to set CLOBBER to something higher than 1 to make it happen.

```
    sub CLEAR    {
        carp &whowasi if $DEBUG;
        my $self = shift;
        croak "@{[&whowasi]}: won't remove all dot files for $self->{USER}"
            unless $self->{CLOBBER} > 1;
        my $dot;
        foreach $dot ( keys %{$self->{LIST}}) {
            $self->DELETE($dot);
        }
    }
```

EXISTS this, key

This method is triggered when the user uses the exists() function on a particular hash. In our example, we'll look at the {LIST} hash element for this:

```
    sub EXISTS   {
        carp &whowasi if $DEBUG;
        my $self = shift;
        my $dot = shift;
        return exists $self->{LIST}->{$dot};
    }
```

FIRSTKEY this

This method will be triggered when the user is going to iterate through the hash, such as via a keys() or each() call.

```
    sub FIRSTKEY {
        carp &whowasi if $DEBUG;
        my $self = shift;
        my $a = keys %{$self->{LIST}};          # reset each() iterator
        each %{$self->{LIST}}
    }
```

NEXTKEY this, lastkey

This method gets triggered during a keys() or each() iteration. It has a second argument which is the last key that had been accessed. This is useful if you're carrying about ordering or calling the iterator from more than one sequence, or not really storing things in a hash anywhere.

For our example, we're using a real hash so we'll do just the simple thing, but we'll have to go through the LIST field indirectly.

```
    sub NEXTKEY  {
        carp &whowasi if $DEBUG;
        my $self = shift;
        return each %{ $self->{LIST} }
    }
```

SCALAR this

This is called when the hash is evaluated in scalar context. In order to mimic the behaviour of untied hashes, this method should return a false value when the tied hash is considered empty. If this method does not exist, perl will make some educated guesses and return true when the hash is inside an iteration. If this isn't the case, FIRSTKEY is called, and the result will be a false value if FIRSTKEY returns the empty list, true otherwise.

However, you should **not** blindly rely on perl always doing the right thing. Particularly, perl will mistakenly return true when you clear the hash by repeatedly calling DELETE until it is empty. You are therefore advised to supply your own SCALAR method when you want to be absolutely sure that your hash behaves nicely in scalar context.

In our example we can just call `scalar` on the underlying hash referenced by `$self->{LIST}`:

```
sub SCALAR {
    carp &whowasi if $DEBUG;
    my $self = shift;
    return scalar %{ $self->{LIST} }
}
```

UNTIE this

This is called when `untie` occurs. See Section 76.3.6 [The untie Gotcha], page 1305 below.

DESTROY this

This method is triggered when a tied hash is about to go out of scope. You don't really need it unless you're trying to add debugging or have auxiliary state to clean up. Here's a very simple function:

```
sub DESTROY  {
    carp &whowasi if $DEBUG;
}
```

Note that functions such as keys() and values() may return huge lists when used on large objects, like DBM files. You may prefer to use the each() function to iterate over such. Example:

```
# print out history file offsets
use NDBM_File;
tie(%HIST, 'NDBM_File', '/usr/lib/news/history', 1, 0);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
untie(%HIST);
```

76.3.4 Tying FileHandles

This is partially implemented now.

A class implementing a tied filehandle should define the following methods: TIEHANDLE, at least one of PRINT, PRINTF, WRITE, READLINE, GETC, READ, and possibly CLOSE, UNTIE and DESTROY. The class can also provide: BINMODE, OPEN, EOF, FILENO, SEEK, TELL - if the corresponding perl operators are used on the handle.

When STDERR is tied, its PRINT method will be called to issue warnings and error messages. This feature is temporarily disabled during the call, which means you can use warn() inside PRINT without starting a recursive loop. And just like __WARN__ and __DIE__ handlers, STDERR's PRINT method may be called to report parser errors, so the caveats mentioned under [perlvar %SIG], page 1383 apply.

All of this is especially useful when perl is embedded in some other program, where output to STDOUT and STDERR may have to be redirected in some special way. See nvi and the Apache module for examples.

When tying a handle, the first argument to tie should begin with an asterisk. So, if you are tying STDOUT, use *STDOUT. If you have assigned it to a scalar variable, say $handle, use *$handle. tie $handle ties the scalar variable $handle, not the handle inside it.

In our example we're going to create a shouting handle.

```
package Shout;
```

TIEHANDLE classname, LIST

This is the constructor for the class. That means it is expected to return a blessed reference of some sort. The reference can be used to hold some internal information.

```
sub TIEHANDLE { print "<shout>\n"; my $i; bless \$i, shift }
```

WRITE this, LIST

This method will be called when the handle is written to via the syswrite function.

```
sub WRITE {
    $r = shift;
    my($buf,$len,$offset) = @_;
    print "WRITE called, \$buf=$buf, \$len=$len, \$offset=$offset";
}
```

PRINT this, LIST

This method will be triggered every time the tied handle is printed to with the print() or say() functions. Beyond its self reference it also expects the list that was passed to the print function.

```
sub PRINT { $r = shift; $$r++; print join($,,map(uc($_),@_)),$\ }
```

say() acts just like print() except $\ will be localized to \n so you need do nothing special to handle say() in PRINT().

PRINTF this, LIST

This method will be triggered every time the tied handle is printed to with the printf() function. Beyond its self reference it also expects the format and list that was passed to the printf function.

```
sub PRINTF {
    shift;
    my $fmt = shift;
    print sprintf($fmt, @_);
}
```

READ this, LIST

This method will be called when the handle is read from via the read or sysread functions.

```
sub READ {
    my $self = shift;
    my $bufref = \$_[0];
    my(undef,$len,$offset) = @_;
    print "READ called, \$buf=$bufref, \$len=$len, \$offset=$offset";
    # add to $$bufref, set $len to number of characters read
    $len;
}
```

READLINE this

This method is called when the handle is read via `<HANDLE>` or `readline HANDLE`.

As per [`readline`], page 433, in scalar context it should return the next line, or `undef` for no more data. In list context it should return all remaining lines, or an empty list for no more data. The strings returned should include the input record separator `$/` (see Section 86.1 [perlvar NAME], page 1375), unless it is `undef` (which means "slurp" mode).

```
sub READLINE {
  my $r = shift;
  if (wantarray) {
    return ("all remaining\n",
            "lines up\n",
            "to eof\n");
  } else {
    return "READLINE called " . ++$$r . " times\n";
  }
}
```

GETC this

This method will be called when the `getc` function is called.

```
sub GETC { print "Don't GETC, Get Perl"; return "a"; }
```

EOF this

This method will be called when the `eof` function is called.

Starting with Perl 5.12, an additional integer parameter will be passed. It will be zero if `eof` is called without parameter; `1` if `eof` is given a filehandle as a parameter, e.g. `eof(FH)`; and `2` in the very special case that the tied filehandle is `ARGV` and `eof` is called with an empty parameter list, e.g. `eof()`.

```
sub EOF { not length $stringbuf }
```

CLOSE this

This method will be called when the handle is closed via the `close` function.

```
sub CLOSE { print "CLOSE called.\n" }
```

UNTIE this

As with the other types of ties, this method will be called when `untie` happens. It may be appropriate to "auto CLOSE" when this occurs. See Section 76.3.6 [The untie Gotcha], page 1305 below.

DESTROY this

As with the other types of ties, this method will be called when the tied handle is about to be destroyed. This is useful for debugging and possibly cleaning up.

```
sub DESTROY { print "</shout>\n" }
```

Here's how to use our little example:

```
tie(*FOO,'Shout');
print FOO "hello\n";
$a = 4; $b = 6;
print FOO $a, " plus ", $b, " equals ", $a + $b, "\n";
print <FOO>;
```

76.3.5 UNTIE this

You can define for all tie types an UNTIE method that will be called at untie(). See Section 76.3.6 [The untie Gotcha], page 1305 below.

76.3.6 The `untie` Gotcha

If you intend making use of the object returned from either tie() or tied(), and if the tie's target class defines a destructor, there is a subtle gotcha you *must* guard against.

As setup, consider this (admittedly rather contrived) example of a tie; all it does is use a file to keep a log of the values assigned to a scalar.

```
package Remember;

use strict;
use warnings;
use IO::File;

sub TIESCALAR {
    my $class = shift;
    my $filename = shift;
    my $handle = IO::File->new( "> $filename" )
                     or die "Cannot open $filename: $!\n";

    print $handle "The Start\n";
    bless {FH => $handle, Value => 0}, $class;
}

sub FETCH {
    my $self = shift;
    return $self->{Value};
}

sub STORE {
    my $self = shift;
    my $value = shift;
    my $handle = $self->{FH};
```

```
        print $handle "$value\n";
        $self->{Value} = $value;
    }

    sub DESTROY {
        my $self = shift;
        my $handle = $self->{FH};
        print $handle "The End\n";
        close $handle;
    }

    1;
```

Here is an example that makes use of this tie:

```
    use strict;
    use Remember;

    my $fred;
    tie $fred, 'Remember', 'myfile.txt';
    $fred = 1;
    $fred = 4;
    $fred = 5;
    untie $fred;
    system "cat myfile.txt";
```

This is the output when it is executed:

```
    The Start
    1
    4
    5
    The End
```

So far so good. Those of you who have been paying attention will have spotted that the tied object hasn't been used so far. So lets add an extra method to the Remember class to allow comments to be included in the file; say, something like this:

```
    sub comment {
        my $self = shift;
        my $text = shift;
        my $handle = $self->{FH};
        print $handle $text, "\n";
    }
```

And here is the previous example modified to use the `comment` method (which requires the tied object):

```
    use strict;
    use Remember;

    my ($fred, $x);
    $x = tie $fred, 'Remember', 'myfile.txt';
```

```
$fred = 1;
$fred = 4;
comment $x "changing...";
$fred = 5;
untie $fred;
system "cat myfile.txt";
```

When this code is executed there is no output. Here's why:

When a variable is tied, it is associated with the object which is the return value of the TIESCALAR, TIEARRAY, or TIEHASH function. This object normally has only one reference, namely, the implicit reference from the tied variable. When untie() is called, that reference is destroyed. Then, as in the first example above, the object's destructor (DESTROY) is called, which is normal for objects that have no more valid references; and thus the file is closed.

In the second example, however, we have stored another reference to the tied object in $x. That means that when untie() gets called there will still be a valid reference to the object in existence, so the destructor is not called at that time, and thus the file is not closed. The reason there is no output is because the file buffers have not been flushed to disk.

Now that you know what the problem is, what can you do to avoid it? Prior to the introduction of the optional UNTIE method the only way was the good old `-w` flag. Which will spot any instances where you call untie() and there are still valid references to the tied object. If the second script above this near the top `use warnings 'untie'` or was run with the `-w` flag, Perl prints this warning message:

```
untie attempted while 1 inner references still exist
```

To get the script to work properly and silence the warning make sure there are no valid references to the tied object *before* untie() is called:

```
undef $x;
untie $fred;
```

Now that UNTIE exists the class designer can decide which parts of the class functionality are really associated with `untie` and which with the object being destroyed. What makes sense for a given class depends on whether the inner references are being kept so that non-tie-related methods can be called on the object. But in most cases it probably makes sense to move the functionality that would have been in DESTROY to the UNTIE method.

If the UNTIE method exists then the warning above does not occur. Instead the UNTIE method is passed the count of "extra" references and can issue its own warning if appropriate. e.g. to replicate the no UNTIE case this method can be used:

```
sub UNTIE
{
 my ($obj,$count) = @_;
 carp "untie attempted while $count inner references still exist" if $count;
}
```

76.4 SEE ALSO

See DB_File or Config for some interesting tie() implementations. A good starting point for many tie() implementations is with one of the modules Tie-Scalar, Tie-Array, Tie-Hash, or Tie-Handle.

76.5 BUGS

The bucket usage information provided by scalar(%hash) is not available. What this means is that using %tied_hash in boolean context doesn't work right (currently this always tests false, regardless of whether the hash is empty or hash elements).

Localizing tied arrays or hashes does not work. After exiting the scope the arrays or the hashes are not restored.

Counting the number of entries in a hash via scalar(keys(%hash)) or scalar(values(%hash)) is inefficient since it needs to iterate through all the entries with FIRSTKEY/NEXTKEY.

Tied hash/array slices cause multiple FETCH/STORE pairs, there are no tie methods for slice operations.

You cannot easily tie a multilevel data structure (such as a hash of hashes) to a dbm file. The first problem is that all but GDBM and Berkeley DB have size limitations, but beyond that, you also have problems with how references are to be represented on disk. One module that does attempt to address this need is DBM::Deep. Check your nearest CPAN site as described in perlmodlib for source code. Note that despite its name, DBM::Deep does not use dbm. Another earlier attempt at solving the problem is MLDBM, which is also available on the CPAN, but which has some fairly serious limitations.

Tied filehandles are still incomplete. sysopen(), truncate(), flock(), fcntl(), stat() and -X can't currently be trapped.

76.6 AUTHOR

Tom Christiansen

TIEHANDLE by Sven Verdoolaege <skimo@dns.ufsia.ac.be> and Doug MacEachern <dougm@osf.org>

UNTIE by Nick Ing-Simmons <nick@ing-simmons.net>

SCALAR by Tassilo von Parseval <tassilo.von.parseval@rwth-aachen.de>

Tying Arrays by Casey West <casey@geeknest.com>

77 perltodo

77.1 NAME

perltodo - Link to the Perl to-do list

77.2 DESCRIPTION

The Perl 5 to-do list is maintained in the git repository, and can be viewed at `http://perl5.git.perl.org/perl.git/blob/HEAD:/Porting/todo.pod`

(The to-do list used to be here in perltodo. That has stopped, as installing a snapshot that becomes increasingly out of date isn't that useful to anyone.)

78 perltooc

78.1 NAME

perltooc - Links to information on object-oriented programming in Perl

78.2 DESCRIPTION

For information on OO programming with Perl, please see Section 47.1 [perlootut NAME], page 786 and Section 46.1 [perlobj NAME], page 769.

(The above documents supersede the tutorial that was formerly here in perltooc.)

79 perltoot

79.1 NAME

perltoot - Links to information on object-oriented programming in Perl

79.2 DESCRIPTION

For information on OO programming with Perl, please see Section 47.1 [perlootut NAME], page 786 and Section 46.1 [perlobj NAME], page 769.

(The above documents supersede the tutorial that was formerly here in perltoot.)

80 perltrap

80.1 NAME

perltrap - Perl traps for the unwary

80.2 DESCRIPTION

The biggest trap of all is forgetting to `use warnings` or use the **-w** switch; see `warnings` and Section 69.1 [perlrun NAME], page 1176. The second biggest trap is not making your entire program runnable under `use strict`. The third biggest trap is not reading the list of changes in this version of Perl; see `perldelta`.

80.2.1 Awk Traps

Accustomed **awk** users should take special note of the following:

- A Perl program executes only once, not once for each input line. You can do an implicit loop with `-n` or `-p`.
- The English module, loaded via

  ```
  use English;
  ```

  allows you to refer to special variables (like `$/`) with names (like $RS), as though they were in **awk**; see Section 86.1 [perlvar NAME], page 1375 for details.
- Semicolons are required after all simple statements in Perl (except at the end of a block). Newline is not a statement delimiter.
- Curly brackets are required on `if`s and `while`s.
- Variables begin with "$", "@" or "%" in Perl.
- Arrays index from 0. Likewise string positions in substr() and index().
- You have to decide whether your array has numeric or string indices.
- Hash values do not spring into existence upon mere reference.
- You have to decide whether you want to use string or numeric comparisons.
- Reading an input line does not split it for you. You get to split it to an array yourself. And the split() operator has different arguments than **awk**'s.
- The current input line is normally in $_, not $0. It generally does not have the newline stripped. ($0 is the name of the program executed.) See Section 86.1 [perlvar NAME], page 1375.
- $<*digit*> does not refer to fields–it refers to substrings matched by the last match pattern.
- The print() statement does not add field and record separators unless you set `$,` and `$\`. You can set $OFS and $ORS if you're using the English module.
- You must open your files before you print to them.
- The range operator is "..", not comma. The comma operator works as in C.
- The match operator is "=~", not "~". ("~" is the one's complement operator, as in C.)

- The exponentiation operator is `"**"`, not `"^"`. `"^"` is the XOR operator, as in C. (You know, one could get the feeling that **awk** is basically incompatible with C.)
- The concatenation operator is `"."`, not the null string. (Using the null string would render `/pat/ /pat/` unparsable, because the third slash would be interpreted as a division operator–the tokenizer is in fact slightly context sensitive for operators like `"/"`, `"?"`, and `">"`. And in fact, `"."` itself can be the beginning of a number.)
- The `next`, `exit`, and `continue` keywords work differently.
- The following variables work differently:

Awk	Perl
`ARGC`	`scalar @ARGV (compare with $#ARGV)`
`ARGV[0]`	`$0`
`FILENAME`	`$ARGV`
`FNR`	`$. - something`
`FS`	`(whatever you like)`
`NF`	`$#Fld, or some such`
`NR`	`$.`
`OFMT`	`$#`
`OFS`	`$,`
`ORS`	`$\`
`RLENGTH`	`length($&)`
`RS`	`$/`
`RSTART`	``length($`)``
`SUBSEP`	`$;`

- You cannot set $RS to a pattern, only a string.
- When in doubt, run the **awk** construct through **a2p** and see what it gives you.

80.2.2 C/C++ Traps

Cerebral C and C++ programmers should take note of the following:

- Curly brackets are required on `if`'s and `while`'s.
- You must use `elsif` rather than `else if`.
- The `break` and `continue` keywords from C become in Perl `last` and `next`, respectively. Unlike in C, these do *not* work within a `do { } while` construct. See Section 74.2.7 [perlsyn Loop Control], page 1254.
- The switch statement is called `given/when` and only available in perl 5.10 or newer. See Section 74.2.11 [perlsyn Switch Statements], page 1258.
- Variables begin with "$", "@" or "%" in Perl.
- Comments begin with "#", not "/*" or "//". Perl may interpret C/C++ comments as division operators, unterminated regular expressions or the defined-or operator.
- You can't take the address of anything, although a similar operator in Perl is the backslash, which creates a reference.
- `ARGV` must be capitalized. `$ARGV[0]` is C's `argv[1]`, and `argv[0]` ends up in `$0`.
- System calls such as link(), unlink(), rename(), etc. return nonzero for success, not 0. (system(), however, returns zero for success.)

- Signal handlers deal with signal names, not numbers. Use `kill -l` to find their names on your system.

80.2.3 JavaScript Traps

Judicious JavaScript programmers should take note of the following:

- In Perl, binary `+` is always addition. `$string1 + $string2` converts both strings to numbers and then adds them. To concatenate two strings, use the `.` operator.
- The `+` unary operator doesn't do anything in Perl. It exists to avoid syntactic ambiguities.
- Unlike `for...in`, Perl's `for` (also spelled `foreach`) does not allow the left-hand side to be an arbitrary expression. It must be a variable:

  ```
  for my $variable (keys %hash) {
       ...
  }
  ```

  Furthermore, don't forget the `keys` in there, as `foreach my $kv (%hash) {}` iterates over the keys and values, and is generally not useful ($kv would be a key, then a value, and so on).
- To iterate over the indices of an array, use `foreach my $i (0 .. $#array) {}`. `foreach my $v (@array) {}` iterates over the values.
- Perl requires braces following `if`, `while`, `foreach`, etc.
- In Perl, `else if` is spelled `elsif`.
- `? :` has higher precedence than assignment. In JavaScript, one can write:

  ```
  condition ? do_something() : variable = 3
  ```

  and the variable is only assigned if the condition is false. In Perl, you need parentheses:

  ```
  $condition ? do_something() : ($variable = 3);
  ```

  Or just use `if`.
- Perl requires semicolons to separate statements.
- Variables declared with `my` only affect code *after* the declaration. You cannot write `$x = 1; my $x;` and expect the first assignment to affect the same variable. It will instead assign to an `$x` declared previously in an outer scope, or to a global variable.

  Note also that the variable is not visible until the following *statement*. This means that in `my $x = 1 + $x` the second $x refers to one declared previously.
- `my` variables are scoped to the current block, not to the current function. If you write `{my $x;} $x;`, the second `$x` does not refer to the one declared inside the block.
- An object's members cannot be made accessible as variables. The closest Perl equivalent to `with(object) { method() }` is `for`, which can alias `$_` to the object:

  ```
  for ($object) {
      $_->method;
  }
  ```
- The object or class on which a method is called is passed as one of the method's arguments, not as a separate `this` value.

80.2.4 Sed Traps

Seasoned **sed** programmers should take note of the following:

- A Perl program executes only once, not once for each input line. You can do an implicit loop with `-n` or `-p`.
- Backreferences in substitutions use "$" rather than "\".
- The pattern matching metacharacters "(", ")", and "|" do not have backslashes in front.
- The range operator is `...`, rather than comma.

80.2.5 Shell Traps

Sharp shell programmers should take note of the following:

- The backtick operator does variable interpolation without regard to the presence of single quotes in the command.
- The backtick operator does no translation of the return value, unlike **csh**.
- Shells (especially **csh**) do several levels of substitution on each command line. Perl does substitution in only certain constructs such as double quotes, backticks, angle brackets, and search patterns.
- Shells interpret scripts a little bit at a time. Perl compiles the entire program before executing it (except for `BEGIN` blocks, which execute at compile time).
- The arguments are available via @ARGV, not $1, $2, etc.
- The environment is not automatically made available as separate scalar variables.
- The shell's `test` uses "=", "!=", "<" etc for string comparisons and "-eq", "-ne", "-lt" etc for numeric comparisons. This is the reverse of Perl, which uses `eq`, `ne`, `lt` for string comparisons, and `==`, `!= <` etc for numeric comparisons.

80.2.6 Perl Traps

Practicing Perl Programmers should take note of the following:

- Remember that many operations behave differently in a list context than they do in a scalar one. See Section 11.1 [perldata NAME], page 70 for details.
- Avoid barewords if you can, especially all lowercase ones. You can't tell by just looking at it whether a bareword is a function or a string. By using quotes on strings and parentheses on function calls, you won't ever get them confused.
- You cannot discern from mere inspection which builtins are unary operators (like chop() and chdir()) and which are list operators (like print() and unlink()). (Unless prototyped, user-defined subroutines can **only** be list operators, never unary ones.) See Section 48.1 [perlop NAME], page 798 and Section 73.1 [perlsub NAME], page 1216.
- People have a hard time remembering that some functions default to $_, or @ARGV, or whatever, but that others which you might expect to do not.
- The <FH> construct is not the name of the filehandle, it is a readline operation on that handle. The data read is assigned to $_ only if the file read is the sole condition in a while loop:

```
    while (<FH>)      { }
```

```
while (defined($_ = <FH>)) { }..
<FH>;  # data discarded!
```

- Remember not to use `=` when you need `=~`; these two constructs are quite different:

  ```
  $x =  /foo/;
  $x =~ /foo/;
  ```

- The `do {}` construct isn't a real loop that you can use loop control on.
- Use `my()` for local variables whenever you can get away with it (but see Section 24.1 [perlform NAME], page 343 for where you can't). Using `local()` actually gives a local value to a global variable, which leaves you open to unforeseen side-effects of dynamic scoping.
- If you localize an exported variable in a module, its exported value will not change. The local name becomes an alias to a new value but the external name is still an alias for the original.

As always, if any of these are ever officially declared as bugs, they'll be fixed and removed.

81 perlunicode

81.1 NAME

perlunicode - Unicode support in Perl

81.2 DESCRIPTION

If you haven't already, before reading this document, you should become familiar with both Section 84.1 [perlunitut NAME], page 1367 and Section 83.1 [perluniintro NAME], page 1352.

Unicode aims to **UNI**-fy the en-**CODE**-ings of all the world's character sets into a single Standard. For quite a few of the various coding standards that existed when Unicode was first created, converting from each to Unicode essentially meant adding a constant to each code point in the original standard, and converting back meant just subtracting that same constant. For ASCII and ISO-8859-1, the constant is 0. For ISO-8859-5, (Cyrillic) the constant is 864; for Hebrew (ISO-8859-8), it's 1488; Thai (ISO-8859-11), 3424; and so forth. This made it easy to do the conversions, and facilitated the adoption of Unicode.

And it worked; nowadays, those legacy standards are rarely used. Most everyone uses Unicode.

Unicode is a comprehensive standard. It specifies many things outside the scope of Perl, such as how to display sequences of characters. For a full discussion of all aspects of Unicode, see `http://www.unicode.org`.

81.2.1 Important Caveats

Even though some of this section may not be understandable to you on first reading, we think it's important enough to highlight some of the gotchas before delving further, so here goes:

Unicode support is an extensive requirement. While Perl does not implement the Unicode standard or the accompanying technical reports from cover to cover, Perl does support many Unicode features.

Also, the use of Unicode may present security issues that aren't obvious. Read Unicode Security Considerations (`http://www.unicode.org/reports/tr36`).

Safest if you `use feature 'unicode_strings'`

> In order to preserve backward compatibility, Perl does not turn on full internal Unicode support unless the pragma Section "The 'unicode_strings' feature" in `feature` is specified. (This is automatically selected if you `use 5.012` or higher.) Failure to do this can trigger unexpected surprises. See Section 81.2.17 [The "Unicode Bug"], page 1340 below.
>
> This pragma doesn't affect I/O. Nor does it change the internal representation of strings, only their interpretation. There are still several places where Unicode isn't fully supported, such as in filenames.

Input and Output Layers

> Use the `:encoding(...)` layer to read from and write to filehandles using the specified encoding. (See `open`.)

You should convert your non-ASCII, non-UTF-8 Perl scripts to be UTF-8.

See `encoding`.

`use utf8` still needed to enable Section 81.2.10 [UTF-8], page 1334 in scripts

If your Perl script is itself encoded in Section 81.2.10 [UTF-8], page 1334, the `use utf8` pragma must be explicitly included to enable recognition of that (in string or regular expression literals, or in identifier names). **This is the only time when an explicit `use utf8` is needed.** (See `utf8`).

`BOM`-marked scripts and Section 81.2.10 [UTF-16], page 1334 scripts autodetected

However, if a Perl script begins with the Unicode `BOM` (UTF-16LE, UTF16-BE, or UTF-8), or if the script looks like non-`BOM`-marked UTF-16 of either endianness, Perl will correctly read in the script as the appropriate Unicode encoding. (`BOM`-less UTF-8 cannot be effectively recognized or differentiated from ISO 8859-1 or other eight-bit encodings.)

81.2.2 Byte and Character Semantics

Before Unicode, most encodings used 8 bits (a single byte) to encode each character. Thus a character was a byte, and a byte was a character, and there could be only 256 or fewer possible characters. "Byte Semantics" in the title of this section refers to this behavior. There was no need to distinguish between "Byte" and "Character".

Then along comes Unicode which has room for over a million characters (and Perl allows for even more). This means that a character may require more than a single byte to represent it, and so the two terms are no longer equivalent. What matter are the characters as whole entities, and not usually the bytes that comprise them. That's what the term "Character Semantics" in the title of this section refers to.

Perl had to change internally to decouple "bytes" from "characters". It is important that you too change your ideas, if you haven't already, so that "byte" and "character" no longer mean the same thing in your mind.

The basic building block of Perl strings has always been a "character". The changes basically come down to that the implementation no longer thinks that a character is always just a single byte.

There are various things to note:

- String handling functions, for the most part, continue to operate in terms of characters. `length()`, for example, returns the number of characters in a string, just as before. But that number no longer is necessarily the same as the number of bytes in the string (there may be more bytes than characters). The other such functions include `chop()`, `chomp()`, `substr()`, `pos()`, `index()`, `rindex()`, `sort()`, `sprintf()`, and `write()`.

  The exceptions are:

  - the bit-oriented `vec`

  - the byte-oriented `pack`/`unpack` `"C"` format

    However, the `W` specifier does operate on whole characters, as does the `U` specifier.
  - some operators that interact with the platform's operating system

    Operators dealing with filenames are examples.

- when the functions are called from within the scope of the `bytes` pragma

  Likely, you should use this only for debugging anyway.

- Strings–including hash keys–and regular expression patterns may contain characters that have ordinal values larger than 255.

  If you use a Unicode editor to edit your program, Unicode characters may occur directly within the literal strings in UTF-8 encoding, or UTF-16. (The former requires a `BOM` or `use utf8`, the latter requires a `BOM`.)

  Section 83.2.5 [perluniintro Creating Unicode], page 1355 gives other ways to place non-ASCII characters in your strings.

- The `chr()` and `ord()` functions work on whole characters.
- Regular expressions match whole characters. For example, `"."` matches a whole character instead of only a single byte.
- The `tr///` operator translates whole characters. (Note that the `tr///CU` functionality has been removed. For similar functionality to that, see `pack('U0', ...)` and `pack('C0', ...)`).
- `scalar reverse()` reverses by character rather than by byte.
- The bit string operators, `& | ^ ~` and (starting in v5.22) `&. |. ^. ~.` can operate on characters that don't fit into a byte. However, the current behavior is likely to change. You should not use these operators on strings that are encoded in UTF-8. If you're not sure about the encoding of a string, downgrade it before using any of these operators; you can use Section "Utility functions" in `utf8`.

The bottom line is that Perl has always practiced "Character Semantics", but with the advent of Unicode, that is now different than "Byte Semantics".

81.2.3 ASCII Rules versus Unicode Rules

Before Unicode, when a character was a byte was a character, Perl knew only about the 128 characters defined by ASCII, code points 0 through 127 (except for under `use locale`). That left the code points 128 to 255 as unassigned, and available for whatever use a program might want. The only semantics they have is their ordinal numbers, and that they are members of none of the non-negative character classes. None are considered to match `\w` for example, but all match `\W`.

Unicode, of course, assigns each of those code points a particular meaning (along with ones above 255). To preserve backward compatibility, Perl only uses the Unicode meanings when there is some indication that Unicode is what is intended; otherwise the non-ASCII code points remain treated as if they are unassigned.

Here are the ways that Perl knows that a string should be treated as Unicode:

- Within the scope of `use utf8`

  If the whole program is Unicode (signified by using 8-bit **U**nicode **T**ransformation **F**ormat), then all strings within it must be Unicode.

- Within the scope of Section "The 'unicode_strings' feature" in `feature`

  This pragma was created so you can explicitly tell Perl that operations executed within its scope are to use Unicode rules. More operations are affected with newer perls. See Section 81.2.17 [The "Unicode Bug"], page 1340.

- Within the scope of `use 5.012` or higher

  This implicitly turns on `use feature 'unicode_strings'`.
- Within the scope of Section 38.10 [`use locale 'not_characters'`], page 721, or Section 38.1 [`use locale`], page 701 and the current locale is a UTF-8 locale.

  The former is defined to imply Unicode handling; and the latter indicates a Unicode locale, hence a Unicode interpretation of all strings within it.
- When the string contains a Unicode-only code point

  Perl has never accepted code points above 255 without them being Unicode, so their use implies Unicode for the whole string.
- When the string contains a Unicode named code point `\N{...}`

  The `\N{...}` construct explicitly refers to a Unicode code point, even if it is one that is also in ASCII. Therefore the string containing it must be Unicode.
- When the string has come from an external source marked as Unicode

  The [`-C`], page 1179 command line option can specify that certain inputs to the program are Unicode, and the values of this can be read by your Perl code, see [perlvar ${^UNICODE}], page 1407.
- When the string has been upgraded to UTF-8

  The function Section "Utility functions" in `utf8` can be explicitly used to permanently (unless a subsequent `utf8::utf8_downgrade()` is called) cause a string to be treated as Unicode.
- There are additional methods for regular expression patterns

  A pattern that is compiled with the `/u` or `/a` modifiers is treated as Unicode (though there are some restrictions with `/a`). Under the `/d` and `/l` modifiers, there are several other indications for Unicode; see Section 58.2.1.2 [perlre Character set modifiers], page 992.

Note that all of the above are overridden within the scope of `bytes`; but you should be using this pragma only for debugging.

Note also that some interactions with the platform's operating system never use Unicode rules.

When Unicode rules are in effect:

- Case translation operators use the Unicode case translation tables.

  Note that `uc()`, or `\U` in interpolated strings, translates to uppercase, while `ucfirst`, or `\u` in interpolated strings, translates to titlecase in languages that make the distinction (which is equivalent to uppercase in languages without the distinction).

  There is a CPAN module, `Unicode-Casing`, which allows you to define your own mappings to be used in `lc()`, `lcfirst()`, `uc()`, `ucfirst()`, and `fc` (or their double-quoted string inlined versions such as `\U`). (Prior to Perl 5.16, this functionality was partially provided in the Perl core, but suffered from a number of insurmountable drawbacks, so the CPAN module was written instead.)
- Character classes in regular expressions match based on the character properties specified in the Unicode properties database.

  `\w` can be used to match a Japanese ideograph, for instance; and `[[:digit:]]` a Bengali number.

- Named Unicode properties, scripts, and block ranges may be used (like bracketed character classes) by using the `\p{}` "matches property" construct and the `\P{}` negation, "doesn't match property".

  See Section 81.2.5 [Unicode Character Properties], page 1321 for more details.

  You can define your own character properties and use them in the regular expression with the `\p{}` or `\P{}` construct. See Section 81.2.6 [User-Defined Character Properties], page 1329 for more details.

81.2.4 Extended Grapheme Clusters (Logical characters)

Consider a character, say `H`. It could appear with various marks around it, such as an acute accent, or a circumflex, or various hooks, circles, arrows, *etc.*, above, below, to one side or the other, *etc.* There are many possibilities among the world's languages. The number of combinations is astronomical, and if there were a character for each combination, it would soon exhaust Unicode's more than a million possible characters. So Unicode took a different approach: there is a character for the base `H`, and a character for each of the possible marks, and these can be variously combined to get a final logical character. So a logical character–what appears to be a single character–can be a sequence of more than one individual characters. The Unicode standard calls these "extended grapheme clusters" (which is an improved version of the no-longer much used "grapheme cluster"); Perl furnishes the `\X` regular expression construct to match such sequences in their entirety.

But Unicode's intent is to unify the existing character set standards and practices, and several pre-existing standards have single characters that mean the same thing as some of these combinations, like ISO-8859-1, which has quite a few of them. For example, `"LATIN CAPITAL LETTER E WITH ACUTE"` was already in this standard when Unicode came along. Unicode therefore added it to its repertoire as that single character. But this character is considered by Unicode to be equivalent to the sequence consisting of the character `"LATIN CAPITAL LETTER E"` followed by the character `"COMBINING ACUTE ACCENT"`.

`"LATIN CAPITAL LETTER E WITH ACUTE"` is called a "pre-composed" character, and its equivalence with the "E" and the "COMBINING ACCENT" sequence is called canonical equivalence. All pre-composed characters are said to have a decomposition (into the equivalent sequence), and the decomposition type is also called canonical. A string may be comprised as much as possible of precomposed characters, or it may be comprised of entirely decomposed characters. Unicode calls these respectively, "Normalization Form Composed" (NFC) and "Normalization Form Decomposed". The `Unicode-Normalize` module contains functions that convert between the two. A string may also have both composed characters and decomposed characters; this module can be used to make it all one or the other.

You may be presented with strings in any of these equivalent forms. There is currently nothing in Perl 5 that ignores the differences. So you'll have to specially hanlde it. The usual advice is to convert your inputs to `NFD` before processing further.

For more detailed information, see `http://unicode.org/reports/tr15/`.

81.2.5 Unicode Character Properties

(The only time that Perl considers a sequence of individual code points as a single logical character is in the `\X` construct, already mentioned above. Therefore "character" in this discussion means a single Unicode code point.)

Very nearly all Unicode character properties are accessible through regular expressions by using the `\p{}` "matches property" construct and the `\P{}` "doesn't match property" for its negation.

For instance, `\p{Uppercase}` matches any single character with the Unicode "Uppercase" property, while `\p{L}` matches any character with a `General_Category` of "L" (letter) property (see Section 81.2.5.1 [General_Category], page 1323 below). Brackets are not required for single letter property names, so `\p{L}` is equivalent to `\pL`.

More formally, `\p{Uppercase}` matches any single character whose Unicode `Uppercase` property value is `True`, and `\P{Uppercase}` matches any character whose `Uppercase` property value is `False`, and they could have been written as `\p{Uppercase=True}` and `\p{Uppercase=False}`, respectively.

This formality is needed when properties are not binary; that is, if they can take on more values than just `True` and `False`. For example, the `Bidi_Class` property (see Section 81.2.5.2 [Bidirectional Character Types], page 1324 below), can take on several different values, such as `Left`, `Right`, `Whitespace`, and others. To match these, one needs to specify both the property name (`Bidi_Class`), AND the value being matched against (`Left`, `Right`, *etc.*). This is done, as in the examples above, by having the two components separated by an equal sign (or interchangeably, a colon), like `\p{Bidi_Class: Left}`.

All Unicode-defined character properties may be written in these compound forms of `\p{`*`property`*`=`*`value`*`}` or `\p{`*`property`*`:`*`value`*`}`, but Perl provides some additional properties that are written only in the single form, as well as single-form short-cuts for all binary properties and certain others described below, in which you may omit the property name and the equals or colon separator.

Most Unicode character properties have at least two synonyms (or aliases if you prefer): a short one that is easier to type and a longer one that is more descriptive and hence easier to understand. Thus the "L" and "Letter" properties above are equivalent and can be used interchangeably. Likewise, "Upper" is a synonym for "Uppercase", and we could have written `\p{Uppercase}` equivalently as `\p{Upper}`. Also, there are typically various synonyms for the values the property can be. For binary properties, "True" has 3 synonyms: "T", "Yes", and "Y"; and "False" has correspondingly "F", "No", and "N". But be careful. A short form of a value for one property may not mean the same thing as the same short form for another. Thus, for the `Section 81.2.5.1 [General_Category], page 1323` property, "L" means "Letter", but for the Section 81.2.5.2 [`Bidi_Class`], page 1324 property, "L" means "Left". A complete list of properties and synonyms is in `perluniprops`.

Upper/lower case differences in property names and values are irrelevant; thus `\p{Upper}` means the same thing as `\p{upper}` or even `\p{UpPeR}`. Similarly, you can add or subtract underscores anywhere in the middle of a word, so that these are also equivalent to `\p{U_p_p_e_r}`. And white space is irrelevant adjacent to non-word characters, such as the braces and the equals or colon separators, so `\p{ Upper }` and `\p{ Upper_case : Y }` are equivalent to these as well. In fact, white space and even hyphens can usually be added or deleted anywhere. So even `\p{ Up-per case = Yes}` is equivalent. All this is called "loose-matching" by Unicode. The few places where stricter matching is used is in the middle of numbers, and in the Perl extension properties that begin or end with an underscore. Stricter matching cares about white space (except adjacent to non-word characters), hyphens, and non-interior underscores.

You can also use negation in both `\p{}` and `\P{}` by introducing a caret (^) between the first brace and the property name: `\p{^Tamil}` is equal to `\P{Tamil}`.

Almost all properties are immune to case-insensitive matching. That is, adding a `/i` regular expression modifier does not change what they match. There are two sets that are affected. The first set is `Uppercase_Letter`, `Lowercase_Letter`, and `Titlecase_Letter`, all of which match `Cased_Letter` under `/i` matching. And the second set is `Uppercase`, `Lowercase`, and `Titlecase`, all of which match `Cased` under `/i` matching. This set also includes its subsets `PosixUpper` and `PosixLower` both of which under `/i` match `PosixAlpha`. (The difference between these sets is that some things, such as Roman numerals, come in both upper and lower case so they are `Cased`, but aren't considered letters, so they aren't `Cased_Letter`'s.)

See Section 81.2.12 [Beyond Unicode code points], page 1337 for special considerations when matching Unicode properties against non-Unicode code points.

81.2.5.1 General_Category

Every Unicode character is assigned a general category, which is the "most usual categorization of a character" (from `http://www.unicode.org/reports/tr44`).

The compound way of writing these is like `\p{General_Category=Number}` (short: `\p{gc:n}`). But Perl furnishes shortcuts in which everything up through the equal or colon separator is omitted. So you can instead just write `\pN`.

Here are the short and long forms of the values the `General Category` property can have:

Short	Long
L	Letter
LC, L&	Cased_Letter (that is: [\p{Ll}\p{Lu}\p{Lt}])
Lu	Uppercase_Letter
Ll	Lowercase_Letter
Lt	Titlecase_Letter
Lm	Modifier_Letter
Lo	Other_Letter
M	Mark
Mn	Nonspacing_Mark
Mc	Spacing_Mark
Me	Enclosing_Mark
N	Number
Nd	Decimal_Number (also Digit)
Nl	Letter_Number
No	Other_Number
P	Punctuation (also Punct)
Pc	Connector_Punctuation
Pd	Dash_Punctuation
Ps	Open_Punctuation

Pe	Close_Punctuation
Pi	Initial_Punctuation (may behave like Ps or Pe depending on usage)
Pf	Final_Punctuation (may behave like Ps or Pe depending on usage)
Po	Other_Punctuation
S	Symbol
Sm	Math_Symbol
Sc	Currency_Symbol
Sk	Modifier_Symbol
So	Other_Symbol
Z	Separator
Zs	Space_Separator
Zl	Line_Separator
Zp	Paragraph_Separator
C	Other
Cc	Control (also Cntrl)
Cf	Format
Cs	Surrogate
Co	Private_Use
Cn	Unassigned

Single-letter properties match all characters in any of the two-letter sub-properties starting with the same letter. `LC` and `L&` are special: both are aliases for the set consisting of everything matched by `Ll`, `Lu`, and `Lt`.

81.2.5.2 Bidirectional Character Types

Because scripts differ in their directionality (Hebrew and Arabic are written right to left, for example) Unicode supplies a `Bidi_Class` property. Some of the values this property can have are:

Value	Meaning
L	Left-to-Right
LRE	Left-to-Right Embedding
LRO	Left-to-Right Override
R	Right-to-Left
AL	Arabic Letter
RLE	Right-to-Left Embedding
RLO	Right-to-Left Override
PDF	Pop Directional Format
EN	European Number
ES	European Separator
ET	European Terminator
AN	Arabic Number

CS	Common Separator
NSM	Non-Spacing Mark
BN	Boundary Neutral
B	Paragraph Separator
S	Segment Separator
WS	Whitespace
ON	Other Neutrals

This property is always written in the compound form. For example, `\p{Bidi_Class:R}` matches characters that are normally written right to left. Unlike the `Section 81.2.5.1 [General_Category], page 1323` property, this property can have more values added in a future Unicode release. Those listed above comprised the complete set for many Unicode releases, but others were added in Unicode 6.3; you can always find what the current ones are in in `perluniprops`. And `http://www.unicode.org/reports/tr9/` describes how to use them.

81.2.5.3 Scripts

The world's languages are written in many different scripts. This sentence (unless you're reading it in translation) is written in Latin, while Russian is written in Cyrillic, and Greek is written in, well, Greek; Japanese mainly in Hiragana or Katakana. There are many more.

The Unicode `Script` and `Script_Extensions` properties give what script a given character is in. Either property can be specified with the compound form like `\p{Script=Hebrew}` (short: `\p{sc=hebr}`), or `\p{Script_Extensions=Javanese}` (short: `\p{scx=java}`). In addition, Perl furnishes shortcuts for all `Script` property names. You can omit everything up through the equals (or colon), and simply write `\p{Latin}` or `\P{Cyrillic}`. (This is not true for `Script_Extensions`, which is required to be written in the compound form.)

The difference between these two properties involves characters that are used in multiple scripts. For example the digits '0' through '9' are used in many parts of the world. These are placed in a script named `Common`. Other characters are used in just a few scripts. For example, the `"KATAKANA-HIRAGANA DOUBLE HYPHEN"` is used in both Japanese scripts, Katakana and Hiragana, but nowhere else. The `Script` property places all characters that are used in multiple scripts in the `Common` script, while the `Script_Extensions` property places those that are used in only a few scripts into each of those scripts; while still using `Common` for those used in many scripts. Thus both these match:

```
"0" =~ /\p{sc=Common}/     # Matches
"0" =~ /\p{scx=Common}/    # Matches
```

and only the first of these match:

```
"\N{KATAKANA-HIRAGANA DOUBLE HYPHEN}" =~ /\p{sc=Common}  # Matches
"\N{KATAKANA-HIRAGANA DOUBLE HYPHEN}" =~ /\p{scx=Common} # No match
```

And only the last two of these match:

```
"\N{KATAKANA-HIRAGANA DOUBLE HYPHEN}" =~ /\p{sc=Hiragana}  # No match
"\N{KATAKANA-HIRAGANA DOUBLE HYPHEN}" =~ /\p{sc=Katakana}  # No match
"\N{KATAKANA-HIRAGANA DOUBLE HYPHEN}" =~ /\p{scx=Hiragana} # Matches
"\N{KATAKANA-HIRAGANA DOUBLE HYPHEN}" =~ /\p{scx=Katakana} # Matches
```

`Script_Extensions` is thus an improved `Script`, in which there are fewer characters in the `Common` script, and correspondingly more in other scripts. It is new in Unicode version

6.0, and its data are likely to change significantly in later releases, as things get sorted out. New code should probably be using Script_Extensions and not plain Script.

(Actually, besides Common, the Inherited script, contains characters that are used in multiple scripts. These are modifier characters which inherit the script value of the controlling character. Some of these are used in many scripts, and so go into Inherited in both Script and Script_Extensions. Others are used in just a few scripts, so are in Inherited in Script, but not in Script_Extensions.)

It is worth stressing that there are several different sets of digits in Unicode that are equivalent to 0-9 and are matchable by \d in a regular expression. If they are used in a single language only, they are in that language's Script and Script_Extension. If they are used in more than one script, they will be in sc=Common, but only if they are used in many scripts should they be in scx=Common.

A complete list of scripts and their shortcuts is in perluniprops.

81.2.5.4 Use of the "Is" Prefix

For backward compatibility (with Perl 5.6), all properties writable without using the compound form mentioned so far may have Is or Is_ prepended to their name, so \P{Is_Lu}, for example, is equal to \P{Lu}, and \p{IsScript:Arabic} is equal to \p{Arabic}.

81.2.5.5 Blocks

In addition to **scripts**, Unicode also defines **blocks** of characters. The difference between scripts and blocks is that the concept of scripts is closer to natural languages, while the concept of blocks is more of an artificial grouping based on groups of Unicode characters with consecutive ordinal values. For example, the "Basic Latin" block is all the characters whose ordinals are between 0 and 127, inclusive; in other words, the ASCII characters. The "Latin" script contains some letters from this as well as several other blocks, like "Latin-1 Supplement", "Latin Extended-A", *etc.*, but it does not contain all the characters from those blocks. It does not, for example, contain the digits 0-9, because those digits are shared across many scripts, and hence are in the Common script.

For more about scripts versus blocks, see UAX#24 "Unicode Script Property": http://www.unicode.org/reports/tr24

The Script or Script_Extensions properties are likely to be the ones you want to use when processing natural language; the Block property may occasionally be useful in working with the nuts and bolts of Unicode.

Block names are matched in the compound form, like \p{Block: Arrows} or \p{Blk=Hebrew}. Unlike most other properties, only a few block names have a Unicode-defined short name. But Perl does provide a (slight, no longer recommended) shortcut: You can say, for example \p{In_Arrows} or \p{In_Hebrew}.

For backwards compatibility, the In prefix may be omitted if there is no naming conflict with a script or any other property, and you can even use an Is prefix instead in those cases. But don't do this for new code because your code could break in new releases, and this has already happened: There was a time in very early Unicode releases when \p{Hebrew} would have matched the *block* Hebrew; now it doesn't.

Using the In prefix avoids this ambiguity, so far. But new versions of Unicode continue to add new properties whose names begin with In. There is a possibility that one of them someday will conflict with your usage. Since this is just a Perl extension, Unicode's name will take precedence and your code will become broken. Also, Unicode is free to add a script whose name begins with `In`; that would cause problems.

So it's clearer and best to use the compound form when specifying blocks. And be sure that is what you really really want to do. In most cases scripts are what you want instead.

A complete list of blocks and their shortcuts is in `perluniprops`.

81.2.5.6 Other Properties

There are many more properties than the very basic ones described here. A complete list is in `perluniprops`.

Unicode defines all its properties in the compound form, so all single-form properties are Perl extensions. Most of these are just synonyms for the Unicode ones, but some are genuine extensions, including several that are in the compound form. And quite a few of these are actually recommended by Unicode (in `http://www.unicode.org/reports/tr18`).

This section gives some details on all extensions that aren't just synonyms for compound-form Unicode properties (for those properties, you'll have to refer to the Unicode Standard (`http://www.unicode.org/reports/tr44`).

`\p{All}`

This matches every possible code point. It is equivalent to `qr/./s`. Unlike all the other non-user-defined `\p{}` property matches, no warning is ever generated if this is property is matched against a non-Unicode code point (see Section 81.2.12 [Beyond Unicode code points], page 1337 below).

`\p{Alnum}`

This matches any `\p{Alphabetic}` or `\p{Decimal_Number}` character.

`\p{Any}`

This matches any of the 1_114_112 Unicode code points. It is a synonym for `\p{Unicode}`.

`\p{ASCII}`

This matches any of the 128 characters in the US-ASCII character set, which is a subset of Unicode.

`\p{Assigned}`

This matches any assigned code point; that is, any code point whose Section 81.2.5.1 [general category], page 1323 is not `Unassigned` (or equivalently, not `Cn`).

`\p{Blank}`

This is the same as `\h` and `\p{HorizSpace}`: A character that changes the spacing horizontally.

`\p{Decomposition_Type: Non_Canonical}` (Short: `\p{Dt=NonCanon}`)

Matches a character that has a non-canonical decomposition.

The Section 81.2.4 [Extended Grapheme Clusters (Logical characters)], page 1321 section above talked about canonical decompositions. However, many more characters have a different type of decomposition, a "compatible" or "non-canonical" decomposition. The sequences that form these decompositions are not considered canonically equivalent to the pre-composed character. An example is the `"SUPERSCRIPT ONE"`. It is somewhat like a regular digit 1, but not exactly; its decomposition into the digit 1 is called a "compatible" decomposition, specifically a "super" decomposition. There are several such compatibility decompositions (see `http://www.unicode.org/reports/tr44`), including one called "compat", which means some miscellaneous type of decomposition that doesn't fit into the other decomposition categories that Unicode has chosen.

Note that most Unicode characters don't have a decomposition, so their decomposition type is `"None"`.

For your convenience, Perl has added the `Non_Canonical` decomposition type to mean any of the several compatibility decompositions.

`\p{Graph}`

Matches any character that is graphic. Theoretically, this means a character that on a printer would cause ink to be used.

`\p{HorizSpace}`

This is the same as `\h` and `\p{Blank}`: a character that changes the spacing horizontally.

`\p{In=*}`

This is a synonym for `\p{Present_In=*}`

`\p{PerlSpace}`

This is the same as `\s`, restricted to ASCII, namely `[ \f\n\r\t]` and starting in Perl v5.18, a vertical tab.

Mnemonic: Perl's (original) space

`\p{PerlWord}`

This is the same as `\w`, restricted to ASCII, namely `[A-Za-z0-9_]`

Mnemonic: Perl's (original) word.

`\p{Posix...}`

There are several of these, which are equivalents, using the `\p{}` notation, for Posix classes and are described in Section 61.2.3.5 [perlrecharclass POSIX Character Classes], page 1069.

`\p{Present_In: *}` (Short: `\p{In=*}`)

This property is used when you need to know in what Unicode version(s) a character is.

The `"*"` above stands for some two digit Unicode version number, such as `1.1` or `4.0`; or the `"*"` can also be `Unassigned`. This property will match the code points whose final disposition has been settled as of the Unicode release given by the version number; `\p{Present_In: Unassigned}` will match those code points whose meaning has yet to be assigned.

For example, `U+0041 "LATIN CAPITAL LETTER A"` was present in the very first Unicode release available, which is `1.1`, so this property is true for all valid `"*"` versions. On the other hand, `U+1EFF` was not assigned until version 5.1 when it became `"LATIN SMALL LETTER Y WITH LOOP"`, so the only `"*"` that would match it are 5.1, 5.2, and later.

Unicode furnishes the `Age` property from which this is derived. The problem with Age is that a strict interpretation of it (which Perl takes) has it matching the precise release a code point's meaning is introduced in. Thus `U+0041` would match only 1.1; and `U+1EFF` only 5.1. This is not usually what you want.

Some non-Perl implementations of the Age property may change its meaning to be the same as the Perl `Present_In` property; just be aware of that.

Another confusion with both these properties is that the definition is not that the code point has been *assigned*, but that the meaning of the code point has been *determined*. This is because 66 code points will always be unassigned, and so the `Age` for them is the Unicode version in which the decision to make them so was made. For example, `U+FDD0` is to be permanently unassigned to a character, and the decision to do that was made in version 3.1, so `\p{Age=3.1}` matches this character, as also does `\p{Present_In: 3.1}` and up.

`\p{Print}`

This matches any character that is graphical or blank, except controls.

`\p{SpacePerl}`

This is the same as `\s`, including beyond ASCII.

Mnemonic: Space, as modified by Perl. (It doesn't include the vertical tab until v5.18, which both the Posix standard and Unicode consider white space.)

`\p{Title}` and `\p{Titlecase}`

Under case-sensitive matching, these both match the same code points as `\p{General Category=Titlecase_Letter}` (`\p{gc=lt}`). The difference is that under `/i` caseless matching, these match the same as `\p{Cased}`, whereas `\p{gc=lt}` matches `\p{Cased_Letter`).

`\p{Unicode}`

This matches any of the 1_114_112 Unicode code points. `\p{Any}`.

`\p{VertSpace}`

This is the same as `\v`: A character that changes the spacing vertically.

`\p{Word}`

This is the same as `\w`, including over 100_000 characters beyond ASCII.

`\p{XPosix...}`

There are several of these, which are the standard Posix classes extended to the full Unicode range. They are described in Section 61.2.3.5 [perlrecharclass POSIX Character Classes], page 1069.

81.2.6 User-Defined Character Properties

You can define your own binary character properties by defining subroutines whose names begin with `"In"` or `"Is"`. (The experimental feature [perlre (?[ ])], page 1017 provides an alternative which allows more complex definitions.) The subroutines can be defined in any package. The user-defined properties can be used in the regular expression `\p{}` and `\P{}`

constructs; if you are using a user-defined property from a package other than the one you are in, you must specify its package in the \p{} or \P{} construct.

```
# assuming property Is_Foreign defined in Lang::
package main;  # property package name required
if ($txt =~ /\p{Lang::IsForeign}+/) { ... }

package Lang;  # property package name not required
if ($txt =~ /\p{IsForeign}+/) { ... }
```

Note that the effect is compile-time and immutable once defined. However, the subroutines are passed a single parameter, which is 0 if case-sensitive matching is in effect and non-zero if caseless matching is in effect. The subroutine may return different values depending on the value of the flag, and one set of values will immutably be in effect for all case-sensitive matches, and the other set for all case-insensitive matches.

Note that if the regular expression is tainted, then Perl will die rather than calling the subroutine when the name of the subroutine is determined by the tainted data.

The subroutines must return a specially-formatted string, with one or more newline-separated lines. Each line must be one of the following:

- A single hexadecimal number denoting a code point to include.
- Two hexadecimal numbers separated by horizontal whitespace (space or tabular characters) denoting a range of code points to include.
- Something to include, prefixed by "+": a built-in character property (prefixed by "utf8::") or a fully qualified (including package name) user-defined character property, to represent all the characters in that property; two hexadecimal code points for a range; or a single hexadecimal code point.
- Something to exclude, prefixed by "-": an existing character property (prefixed by "utf8::") or a fully qualified (including package name) user-defined character property, to represent all the characters in that property; two hexadecimal code points for a range; or a single hexadecimal code point.
- Something to negate, prefixed "!": an existing character property (prefixed by "utf8::") or a fully qualified (including package name) user-defined character property, to represent all the characters in that property; two hexadecimal code points for a range; or a single hexadecimal code point.
- Something to intersect with, prefixed by "&": an existing character property (prefixed by "utf8::") or a fully qualified (including package name) user-defined character property, for all the characters except the characters in the property; two hexadecimal code points for a range; or a single hexadecimal code point.

For example, to define a property that covers both the Japanese syllabaries (hiragana and katakana), you can define

```
sub InKana {
    return <<END;
3040\t309F
30A0\t30FF
END
}
```

Imagine that the here-doc end marker is at the beginning of the line. Now you can use `\p{InKana}` and `\P{InKana}`.

You could also have used the existing block property names:

```
sub InKana {
    return <<'END';
+utf8::InHiragana
+utf8::InKatakana
END
}
```

Suppose you wanted to match only the allocated characters, not the raw block ranges: in other words, you want to remove the unassigned characters:

```
sub InKana {
    return <<'END';
+utf8::InHiragana
+utf8::InKatakana
-utf8::IsCn
END
}
```

The negation is useful for defining (surprise!) negated classes.

```
sub InNotKana {
    return <<'END';
!utf8::InHiragana
-utf8::InKatakana
+utf8::IsCn
END
}
```

This will match all non-Unicode code points, since every one of them is not in Kana. You can use intersection to exclude these, if desired, as this modified example shows:

```
sub InNotKana {
    return <<'END';
!utf8::InHiragana
-utf8::InKatakana
+utf8::IsCn
&utf8::Any
END
}
```

`&utf8::Any` must be the last line in the definition.

Intersection is used generally for getting the common characters matched by two (or more) classes. It's important to remember not to use `"&"` for the first set; that would be intersecting with nothing, resulting in an empty set.

Unlike non-user-defined `\p{}` property matches, no warning is ever generated if these properties are matched against a non-Unicode code point (see Section 81.2.12 [Beyond Unicode code points], page 1337 below).

81.2.7 User-Defined Case Mappings (for serious hackers only)

This feature has been removed as of Perl 5.16. The CPAN module `Unicode-Casing` provides better functionality without the drawbacks that this feature had. If you are using a Perl earlier than 5.16, this feature was most fully documented in the 5.14 version of this pod: `http://perldoc.perl.org/5.14.0/perlunicode.html#User-Defined-Case-Mappings-%28for-serious-hackers-only%29`

81.2.8 Character Encodings for Input and Output

See `Encode`.

81.2.9 Unicode Regular Expression Support Level

The following list of Unicode supported features for regular expressions describes all features currently directly supported by core Perl. The references to "Level N" and the section numbers refer to the Unicode Technical Standard #18, "Unicode Regular Expressions", version 13, from August 2008.

- Level 1 - Basic Unicode Support

  ```
  RL1.1   Hex Notation                     - done          [1]
  RL1.2   Properties                       - done          [2][3]
  RL1.2a  Compatibility Properties         - done          [4]
  RL1.3   Subtraction and Intersection     - experimental  [5]
  RL1.4   Simple Word Boundaries           - done          [6]
  RL1.5   Simple Loose Matches             - done          [7]
  RL1.6   Line Boundaries                  - MISSING       [8][9]
  RL1.7   Supplementary Code Points        - done          [10]
  ```

  [1] `\N{U+...}` and `\x{...}`
  [2] `\p{...}` `\P{...}`
  [3] supports not only minimal list, but all Unicode character properties (see Unicode Character Properties above)
  [4] `\d \D \s \S \w \W \X [:`*prop*`:] [:^`*prop*`:]`
  [5] The experimental feature starting in v5.18 `"(?[...])"` accomplishes this.

  See [perlre (?[ ])], page 1017. If you don't want to use an experimental feature, you can use one of the following:

  - Regular expression look-ahead

    You can mimic class subtraction using lookahead. For example, what UTS#18 might write as

    ```
    [{Block=Greek}-[{UNASSIGNED}]]
    ```

    in Perl can be written as:

    ```
    (?!\p{Unassigned})\p{Block=Greek}
    (?=\p{Assigned})\p{Block=Greek}
    ```

    But in this particular example, you probably really want

    ```
    \p{Greek}
    ```

    which will match assigned characters known to be part of the Greek script.

- CPAN module `Unicode-Regex-Set`

  It does implement the full UTS#18 grouping, intersection, union, and removal (subtraction) syntax.
- Section 81.2.6 [User-Defined Character Properties], page 1329

  `"+"` for union, `"-"` for removal (set-difference), `"&"` for intersection

[6] `\b \B`

[7] Note that Perl does Full case-folding in matching, not Simple:

For example `U+1F88` is equivalent to `U+1F00 U+03B9`, instead of just `U+1F80`. This difference matters mainly for certain Greek capital letters with certain modifiers: the Full case-folding decomposes the letter, while the Simple case-folding would map it to a single character.

[8] Perl treats `\n` as the start- and end-line delimiter. Unicode specifies more characters that should be so-interpreted.

These are:

```
VT   U+000B  (\v in C)
FF   U+000C  (\f)
CR   U+000D  (\r)
NEL  U+0085
LS   U+2028
PS   U+2029
```

`^` and `$` in regular expression patterns are supposed to match all these, but don't. These characters also don't, but should, affect `<>` `$.`, and script line numbers.

Also, lines should not be split within `CRLF` (i.e. there is no empty line between `\r` and `\n`). For `CRLF`, try the `:crlf` layer (see `PerlIO`).

[9] But `Unicode-LineBreak` is available.

This module supplies line breaking conformant with UAX#14 "Unicode Line Breaking Algorithm" (`http://www.unicode.org/reports/tr14`).

[10] UTF-8/UTF-EBDDIC used in Perl allows not only `U+10000` to `U+10FFFF` but also beyond `U+10FFFF`

- Level 2 - Extended Unicode Support

```
RL2.1   Canonical Equivalents           - MISSING       [10][11]
RL2.2   Default Grapheme Clusters       - MISSING       [12]
RL2.3   Default Word Boundaries         - DONE          [14]
RL2.4   Default Loose Matches           - MISSING       [15]
RL2.5   Name Properties                 - DONE
RL2.6   Wildcard Properties             - MISSING

[10] see UAX#15 "Unicode Normalization Forms"
[11] have Unicode::Normalize but not integrated to regexes
[12] have \X and \b{gcb} but we don't have a "Grapheme Cluster
     Mode"
[14] see UAX#29, Word Boundaries
[15] This is covered in Chapter 3.13 (in Unicode 6.0)
```

- Level 3 - Tailored Support

```
RL3.1   Tailored Punctuation            - MISSING
RL3.2   Tailored Grapheme Clusters      - MISSING       [17][18]
RL3.3   Tailored Word Boundaries        - MISSING
RL3.4   Tailored Loose Matches          - MISSING
RL3.5   Tailored Ranges                 - MISSING
RL3.6   Context Matching                - MISSING       [19]
RL3.7   Incremental Matches             - MISSING
     ( RL3.8   Unicode Set Sharing )
RL3.9   Possible Match Sets             - MISSING
RL3.10  Folded Matching                 - MISSING       [20]
RL3.11  Submatchers                     - MISSING

[17] see UAX#10 "Unicode Collation Algorithms"
[18] have Unicode::Collate but not integrated to regexes
[19] have (?<=x) and (?=x), but look-aheads or look-behinds
     should see outside of the target substring
[20] need insensitive matching for linguistic features other
     than case; for example, hiragana to katakana, wide and
     narrow, simplified Han to traditional Han (see UTR#30
     "Character Foldings")
```

81.2.10 Unicode Encodings

Unicode characters are assigned to *code points*, which are abstract numbers. To use these numbers, various encodings are needed.

- UTF-8

  UTF-8 is a variable-length (1 to 4 bytes), byte-order independent encoding. In most of Perl's documentation, including elsewhere in this document, the term "UTF-8" means also "UTF-EBCDIC". But in this section, "UTF-8" refers only to the encoding used on ASCII platforms. It is a superset of 7-bit US-ASCII, so anything encoded in ASCII has the identical representation when encoded in UTF-8.

  The following table is from Unicode 3.2.

```
Code Points            1st Byte  2nd Byte  3rd Byte 4th Byte

  U+0000..U+007F       00..7F
  U+0080..U+07FF     * C2..DF    80..BF
  U+0800..U+0FFF       E0      * A0..BF    80..BF
  U+1000..U+CFFF       E1..EC    80..BF    80..BF
  U+D000..U+D7FF       ED        80..9F    80..BF
  U+D800..U+DFFF       +++++ utf16 surrogates, not legal utf8 +++++
  U+E000..U+FFFF       EE..EF    80..BF    80..BF
 U+10000..U+3FFFF      F0      * 90..BF    80..BF    80..BF
 U+40000..U+FFFFF      F1..F3    80..BF    80..BF    80..BF
U+100000..U+10FFFF     F4        80..8F    80..BF    80..BF
```

  Note the gaps marked by "*" before several of the byte entries above. These are caused by legal UTF-8 avoiding non-shortest encodings: it is technically possible to UTF-8-

encode a single code point in different ways, but that is explicitly forbidden, and the shortest possible encoding should always be used (and that is what Perl does).

Another way to look at it is via bits:

```
                Code Points  1st Byte  2nd Byte  3rd Byte  4th Byte

                   0aaaaaaa  0aaaaaaa
           00000bbbbbaaaaaa  110bbbbb  10aaaaaa
           ccccbbbbbbaaaaaa  1110cccc  10bbbbbb  10aaaaaa
 00000dddccccccbbbbbbaaaaaa  11110ddd  10cccccc  10bbbbbb  10aaaaaa
```

As you can see, the continuation bytes all begin with "10", and the leading bits of the start byte tell how many bytes there are in the encoded character.

The original UTF-8 specification allowed up to 6 bytes, to allow encoding of numbers up to 0x7FFF_FFFF. Perl continues to allow those, and has extended that up to 13 bytes to encode code points up to what can fit in a 64-bit word. However, Perl will warn if you output any of these as being non-portable; and under strict UTF-8 input protocols, they are forbidden.

- UTF-EBCDIC

  Like UTF-8, but EBCDIC-safe, in the way that UTF-8 is ASCII-safe. This means that all the basic characters (which includes all those that have ASCII equivalents (like "A", "0", "%", *etc.*) are the same in both EBCDIC and UTF-EBCDIC.)

  UTF-EBCDIC is used on EBCDIC platforms. The largest Unicode code points take 5 bytes to represent (instead of 4 in UTF-8), and Perl extends it to a maximum of 7 bytes to encode pode points up to what can fit in a 32-bit word (instead of 13 bytes and a 64-bit word in UTF-8).

- UTF-16, UTF-16BE, UTF-16LE, Surrogates, and BOM's (Byte Order Marks)

  The followings items are mostly for reference and general Unicode knowledge, Perl doesn't use these constructs internally.

  Like UTF-8, UTF-16 is a variable-width encoding, but where UTF-8 uses 8-bit code units, UTF-16 uses 16-bit code units. All code points occupy either 2 or 4 bytes in UTF-16: code points U+0000..U+FFFF are stored in a single 16-bit unit, and code points U+10000..U+10FFFF in two 16-bit units. The latter case is using *surrogates*, the first 16-bit unit being the *high surrogate*, and the second being the *low surrogate*.

  Surrogates are code points set aside to encode the U+10000..U+10FFFF range of Unicode code points in pairs of 16-bit units. The *high surrogates* are the range U+D800..U+DBFF and the *low surrogates* are the range U+DC00..U+DFFF. The surrogate encoding is

```
    $hi = ($uni - 0x10000) / 0x400 + 0xD800;
    $lo = ($uni - 0x10000) % 0x400 + 0xDC00;
```

  and the decoding is

```
    $uni = 0x10000 + ($hi - 0xD800) * 0x400 + ($lo - 0xDC00);
```

  Because of the 16-bitness, UTF-16 is byte-order dependent. UTF-16 itself can be used for in-memory computations, but if storage or transfer is required either UTF-16BE (big-endian) or UTF-16LE (little-endian) encodings must be chosen.

  This introduces another problem: what if you just know that your data is UTF-16, but you don't know which endianness? Byte Order Marks, or BOM's, are a solution to this.

A special character has been reserved in Unicode to function as a byte order marker: the character with the code point U+FEFF is the BOM.

The trick is that if you read a BOM, you will know the byte order, since if it was written on a big-endian platform, you will read the bytes 0xFE 0xFF, but if it was written on a little-endian platform, you will read the bytes 0xFF 0xFE. (And if the originating platform was writing in ASCII platform UTF-8, you will read the bytes 0xEF 0xBB 0xBF.)

The way this trick works is that the character with the code point U+FFFE is not supposed to be in input streams, so the sequence of bytes 0xFF 0xFE is unambiguously "BOM, represented in little-endian format" and cannot be U+FFFE, represented in big-endian format".

Surrogates have no meaning in Unicode outside their use in pairs to represent other code points. However, Perl allows them to be represented individually internally, for example by saying chr(0xD801), so that all code points, not just those valid for open interchange, are representable. Unicode does define semantics for them, such as their Section 81.2.5.1 [General_Category], page 1323 is "Cs". But because their use is somewhat dangerous, Perl will warn (using the warning category "surrogate", which is a sub-category of "utf8") if an attempt is made to do things like take the lower case of one, or match case-insensitively, or to output them. (But don't try this on Perls before 5.14.)

- UTF-32, UTF-32BE, UTF-32LE

  The UTF-32 family is pretty much like the UTF-16 family, except that the units are 32-bit, and therefore the surrogate scheme is not needed. UTF-32 is a fixed-width encoding. The BOM signatures are 0x00 0x00 0xFE 0xFF for BE and 0xFF 0xFE 0x00 0x00 for LE.

- UCS-2, UCS-4

  Legacy, fixed-width encodings defined by the ISO 10646 standard. UCS-2 is a 16-bit encoding. Unlike UTF-16, UCS-2 is not extensible beyond U+FFFF, because it does not use surrogates. UCS-4 is a 32-bit encoding, functionally identical to UTF-32 (the difference being that UCS-4 forbids neither surrogates nor code points larger than 0x10_FFFF).

- UTF-7

  A seven-bit safe (non-eight-bit) encoding, which is useful if the transport or storage is not eight-bit safe. Defined by RFC 2152.

81.2.11 Noncharacter code points

66 code points are set aside in Unicode as "noncharacter code points". These all have the Unassigned (Cn) Section 81.2.5.1 [General_Category], page 1323, and no character will ever be assigned to any of them. They are the 32 code points between U+FDD0 and U+FDEF inclusive, and the 34 code points:

```
U+FFFE   U+FFFF
U+1FFFE  U+1FFFF
U+2FFFE  U+2FFFF
...
U+EFFFE  U+EFFFF
```

```
U+FFFFE  U+FFFFF
U+10FFFE U+10FFFF
```

Until Unicode 7.0, the noncharacters were "**forbidden** for use in open interchange of Unicode text data", so that code that processed those streams could use these code points as sentinels that could be mixed in with character data, and would always be distinguishable from that data. (Emphasis above and in the next paragraph are added in this document.)

Unicode 7.0 changed the wording so that they are "**not recommended** for use in open interchange of Unicode text data". The 7.0 Standard goes on to say:

> "If a noncharacter is received in open interchange, an application is not required to interpret it in any way. It is good practice, however, to recognize it as a noncharacter and to take appropriate action, such as replacing it with `U+FFFD` replacement character, to indicate the problem in the text. It is not recommended to simply delete noncharacter code points from such text, because of the potential security issues caused by deleting uninterpreted characters. (See conformance clause C7 in Section 3.2, Conformance Requirements, and Unicode Technical Report #36, "Unicode Security Considerations" (`http://www.unicode.org/reports/tr36/#Substituting_for_Ill_Formed_Subsequences`))."

This change was made because it was found that various commercial tools like editors, or for things like source code control, had been written so that they would not handle program files that used these code points, effectively precluding their use almost entirely! And that was never the intent. They've always been meant to be usable within an application, or cooperating set of applications, at will.

If you're writing code, such as an editor, that is supposed to be able to handle any Unicode text data, then you shouldn't be using these code points yourself, and instead allow them in the input. If you need sentinels, they should instead be something that isn't legal Unicode. For UTF-8 data, you can use the bytes 0xC1 and 0xC2 as sentinels, as they never appear in well-formed UTF-8. (There are equivalents for UTF-EBCDIC). You can also store your Unicode code points in integer variables and use negative values as sentinels.

If you're not writing such a tool, then whether you accept noncharacters as input is up to you (though the Standard recommends that you not). If you do strict input stream checking with Perl, these code points continue to be forbidden. This is to maintain backward compatibility (otherwise potential security holes could open up, as an unsuspecting application that was written assuming the noncharacters would be filtered out before getting to it, could now, without warning, start getting them). To do strict checking, you can use the layer `:encoding('UTF-8')`.

Perl continues to warn (using the warning category `"nonchar"`, which is a sub-category of `"utf8"`) if an attempt is made to output noncharacters.

81.2.12 Beyond Unicode code points

The maximum Unicode code point is `U+10FFFF`, and Unicode only defines operations on code points up through that. But Perl works on code points up to the maximum permissible unsigned number available on the platform. However, Perl will not accept these from input streams unless lax rules are being used, and will warn (using the warning category `"non_unicode"`, which is a sub-category of `"utf8"`) if any are output.

Since Unicode rules are not defined on these code points, if a Unicode-defined operation is done on them, Perl uses what we believe are sensible rules, while generally warning, using the "non_unicode" category. For example, uc("\x{11_0000}") will generate such a warning, returning the input parameter as its result, since Perl defines the uppercase of every non-Unicode code point to be the code point itself. (All the case changing operations, not just uppercasing, work this way.)

The situation with matching Unicode properties in regular expressions, the \p{} and \P{} constructs, against these code points is not as clear cut, and how these are handled has changed as we've gained experience.

One possibility is to treat any match against these code points as undefined. But since Perl doesn't have the concept of a match being undefined, it converts this to failing or FALSE. This is almost, but not quite, what Perl did from v5.14 (when use of these code points became generally reliable) through v5.18. The difference is that Perl treated all \p{} matches as failing, but all \P{} matches as succeeding.

One problem with this is that it leads to unexpected, and confusting results in some cases:

```
chr(0x110000) =~ \p{ASCII_Hex_Digit=True}      # Failed on <= v5.18
chr(0x110000) =~ \p{ASCII_Hex_Digit=False}     # Failed! on <= v5.18
```

That is, it treated both matches as undefined, and converted that to false (raising a warning on each). The first case is the expected result, but the second is likely counterintuitive: "How could both be false when they are complements?" Another problem was that the implementation optimized many Unicode property matches down to already existing simpler, faster operations, which don't raise the warning. We chose to not forgo those optimizations, which help the vast majority of matches, just to generate a warning for the unlikely event that an above-Unicode code point is being matched against.

As a result of these problems, starting in v5.20, what Perl does is to treat non-Unicode code points as just typical unassigned Unicode characters, and matches accordingly. (Note: Unicode has atypical unassigned code points. For example, it has noncharacter code points, and ones that, when they do get assigned, are destined to be written Right-to-left, as Arabic and Hebrew are. Perl assumes that no non-Unicode code point has any atypical properties.)

Perl, in most cases, will raise a warning when matching an above-Unicode code point against a Unicode property when the result is TRUE for \p{}, and FALSE for \P{}. For example:

```
chr(0x110000) =~ \p{ASCII_Hex_Digit=True}      # Fails, no warning
chr(0x110000) =~ \p{ASCII_Hex_Digit=False}     # Succeeds, with warning
```

In both these examples, the character being matched is non-Unicode, so Unicode doesn't define how it should match. It clearly isn't an ASCII hex digit, so the first example clearly should fail, and so it does, with no warning. But it is arguable that the second example should have an undefined, hence FALSE, result. So a warning is raised for it.

Thus the warning is raised for many fewer cases than in earlier Perls, and only when what the result is could be arguable. It turns out that none of the optimizations made by Perl (or are ever likely to be made) cause the warning to be skipped, so it solves both problems of Perl's earlier approach. The most commonly used property that is affected by this change is \p{Unassigned} which is a short form for \p{General_Category=Unassigned}. Starting in v5.20, all non-Unicode code points are considered `Unassigned`. In earlier releases the matches failed because the result was considered undefined.

The only place where the warning is not raised when it might ought to have been is if optimizations cause the whole pattern match to not even be attempted. For example, Perl may figure out that for a string to match a certain regular expression pattern, the string has to contain the substring `"foobar"`. Before attempting the match, Perl may look for that substring, and if not found, immediately fail the match without actually trying it; so no warning gets generated even if the string contains an above-Unicode code point.

This behavior is more "Do what I mean" than in earlier Perls for most applications. But it catches fewer issues for code that needs to be strictly Unicode compliant. Therefore there is an additional mode of operation available to accommodate such code. This mode is enabled if a regular expression pattern is compiled within the lexical scope where the `"non_unicode"` warning class has been made fatal, say by:

```
 use warnings FATAL => "non_unicode"
```

(see `warnings`). In this mode of operation, Perl will raise the warning for all matches against a non-Unicode code point (not just the arguable ones), and it skips the optimizations that might cause the warning to not be output. (It currently still won't warn if the match isn't even attempted, like in the `"foobar"` example above.)

In summary, Perl now normally treats non-Unicode code points as typical Unicode unassigned code points for regular expression matches, raising a warning only when it is arguable what the result should be. However, if this warning has been made fatal, it isn't skipped.

There is one exception to all this. `\p{All}` looks like a Unicode property, but it is a Perl extension that is defined to be true for all possible code points, Unicode or not, so no warning is ever generated when matching this against a non-Unicode code point. (Prior to v5.20, it was an exact synonym for `\p{Any}`, matching code points `0` through `0x10FFFF`.)

81.2.13 Security Implications of Unicode

First, read Unicode Security Considerations (`http://www.unicode.org/reports/tr36`).

Also, note the following:

- Malformed UTF-8

  Unfortunately, the original specification of UTF-8 leaves some room for interpretation of how many bytes of encoded output one should generate from one input Unicode character. Strictly speaking, the shortest possible sequence of UTF-8 bytes should be generated, because otherwise there is potential for an input buffer overflow at the receiving end of a UTF-8 connection. Perl always generates the shortest length UTF-8, and with warnings on, Perl will warn about non-shortest length UTF-8 along with other malformations, such as the surrogates, which are not Unicode code points valid for interchange.
- Regular expression pattern matching may surprise you if you're not accustomed to Unicode. Starting in Perl 5.14, several pattern modifiers are available to control this, called the character set modifiers. Details are given in Section 58.2.1.2 [perlre Character set modifiers], page 992.

As discussed elsewhere, Perl has one foot (two hooves?) planted in each of two worlds: the old world of ASCII and single-byte locales, and the new world of Unicode, upgrading when necessary. If your legacy code does not explicitly use Unicode, no automatic switch-over to Unicode should happen.

81.2.14 Unicode in Perl on EBCDIC

Unicode is supported on EBCDIC platforms. See Section 19.1 [perlebcdic NAME], page 266.

Unless ASCII vs. EBCDIC issues are specifically being discussed, references to UTF-8 encoding in this document and elsewhere should be read as meaning UTF-EBCDIC on EBCDIC platforms. See Section 19.3.6 [perlebcdic Unicode and UTF], page 269.

Because UTF-EBCDIC is so similar to UTF-8, the differences are mostly hidden from you; `use utf8` (and NOT something like `use utfebcdic`) declares the the script is in the platform's "native" 8-bit encoding of Unicode. (Similarly for the `":utf8"` layer.)

81.2.15 Locales

See Section 38.10 [perllocale Unicode and UTF-8], page 721

81.2.16 When Unicode Does Not Happen

There are still many places where Unicode (in some encoding or another) could be given as arguments or received as results, or both in Perl, but it is not, in spite of Perl having extensive ways to input and output in Unicode, and a few other "entry points" like the `@ARGV` array (which can sometimes be interpreted as UTF-8).

The following are such interfaces. Also, see Section 81.2.17 [The "Unicode Bug"], page 1340. For all of these interfaces Perl currently (as of v5.16.0) simply assumes byte strings both as arguments and results, or UTF-8 strings if the (deprecated) `encoding` pragma has been used.

One reason that Perl does not attempt to resolve the role of Unicode in these situations is that the answers are highly dependent on the operating system and the file system(s). For example, whether filenames can be in Unicode and in exactly what kind of encoding, is not exactly a portable concept. Similarly for `qx` and `system`: how well will the "command-line interface" (and which of them?) handle Unicode?

- `chdir`, `chmod`, `chown`, `chroot`, `exec`, `link`, `lstat`, `mkdir`, `rename`, `rmdir`, `stat`, `symlink`, `truncate`, `unlink`, `utime`, `-X`
- `%ENV`
- `glob` (aka the `<*>`)
- `open`, `opendir`, `sysopen`
- `qx` (aka the backtick operator), `system`
- `readdir`, `readlink`

81.2.17 The "Unicode Bug"

The term, "Unicode bug" has been applied to an inconsistency with the code points in the `Latin-1 Supplement` block, that is, between 128 and 255. Without a locale specified, unlike all other characters or code points, these characters can have very different semantics depending on the rules in effect. (Characters whose code points are above 255 force Unicode rules; whereas the rules for ASCII characters are the same under both ASCII and Unicode rules.)

Under Unicode rules, these upper-Latin1 characters are interpreted as Unicode code points, which means they have the same semantics as Latin-1 (ISO-8859-1) and C1 controls.

As explained in Section 81.2.3 [ASCII Rules versus Unicode Rules], page 1319, under ASCII rules, they are considered to be unassigned characters.

This can lead to unexpected results. For example, a string's semantics can suddenly change if a code point above 255 is appended to it, which changes the rules from ASCII to Unicode. As an example, consider the following program and its output:

```
$ perl -le'
    no feature 'unicode_strings';
    $s1 = "\xC2";
    $s2 = "\x{2660}";
    for ($s1, $s2, $s1.$s2) {
        print /\w/ || 0;
    }
'
0
0
1
```

If there's no `\w` in `s1` nor in `s2`, why does their concatenation have one?

This anomaly stems from Perl's attempt to not disturb older programs that didn't use Unicode, along with Perl's desire to add Unicode support seamlessly. But the result turned out to not be seamless. (By the way, you can choose to be warned when things like this happen. See `encoding-warnings`.)

Section "The 'unicode_strings' feature" in `feature` was added, starting in Perl v5.12, to address this problem. It affects these things:

- Changing the case of a scalar, that is, using `uc()`, `ucfirst()`, `lc()`, and `lcfirst()`, or `\L`, `\U`, `\u` and `\l` in double-quotish contexts, such as regular expression substitutions.

  Under `unicode_strings` starting in Perl 5.12.0, Unicode rules are generally used. See [perlfunc lc], page 399 for details on how this works in combination with various other pragmas.

- Using caseless (`/i`) regular expression matching.

  Starting in Perl 5.14.0, regular expressions compiled within the scope of `unicode_strings` use Unicode rules even when executed or compiled into larger regular expressions outside the scope.

- Matching any of several properties in regular expressions.

  These properties are `\b` (without braces), `\B` (without braces), `\s`, `\S`, `\w`, `\W`, and all the Posix character classes *except* `[[:ascii:]]`.

  Starting in Perl 5.14.0, regular expressions compiled within the scope of `unicode_strings` use Unicode rules even when executed or compiled into larger regular expressions outside the scope.

- In `quotemeta` or its inline equivalent `\Q`.

  Starting in Perl 5.16.0, consistent quoting rules are used within the scope of `unicode_strings`, as described in [perlfunc quotemeta], page 430. Prior to that, or outside its scope, no code points above 127 are quoted in UTF-8 encoded strings, but in byte encoded strings, code points between 128-255 are always quoted.

You can see from the above that the effect of `unicode_strings` increased over several Perl releases. (And Perl's support for Unicode continues to improve; it's best to use the latest available release in order to get the most complete and accurate results possible.) Note that `unicode_strings` is automatically chosen if you `use 5.012` or higher.

For Perls earlier than those described above, or when a string is passed to a function outside the scope of `unicode_strings`, see the next section.

81.2.18 Forcing Unicode in Perl (Or Unforcing Unicode in Perl)

Sometimes (see Section 81.2.16 [When Unicode Does Not Happen], page 1340 or Section 81.2.17 [The "Unicode Bug"], page 1340) there are situations where you simply need to force a byte string into UTF-8, or vice versa. The standard module `Encode` can be used for this, or the low-level calls Section "Utility functions" in `utf8` and Section "Utility functions" in `utf8`.

Note that `utf8::downgrade()` can fail if the string contains characters that don't fit into a byte.

Calling either function on a string that already is in the desired state is a no-op.

Section 81.2.3 [ASCII Rules versus Unicode Rules], page 1319 gives all the ways that a string is made to use Unicode rules.

81.2.19 Using Unicode in XS

See Section 28.11 [perlguts Unicode Support], page 555 for an introduction to Unicode at the XS level, and Section "Unicode Support" in `perlapi` for the API details.

81.2.20 Hacking Perl to work on earlier Unicode versions (for very serious hackers only)

Perl by default comes with the latest supported Unicode version built-in, but the goal is to allow you to change to use any earlier one. In Perls v5.20 and v5.22, however, the earliest usable version is Unicode 5.1. Perl v5.18 is able to handle all earlier versions.

Download the files in the desired version of Unicode from the Unicode web site `http://www.unicode.org`). These should replace the existing files in `lib/unicore` in the Perl source tree. Follow the instructions in `README.perl` in that directory to change some of their names, and then build perl (see `INSTALL`).

81.2.21 Porting code from perl-5.6.X

Perls starting in 5.8 have a different Unicode model from 5.6. In 5.6 the programmer was required to use the `utf8` pragma to declare that a given scope expected to deal with Unicode data and had to make sure that only Unicode data were reaching that scope. If you have code that is working with 5.6, you will need some of the following adjustments to your code. The examples are written such that the code will continue to work under 5.6, so you should be safe to try them out.

- A filehandle that should read or write UTF-8

```
    if ($] > 5.008) {
```

```
  binmode $fh, ":encoding(utf8)";
}
```

- A scalar that is going to be passed to some extension

  Be it `Compress::Zlib`, `Apache::Request` or any extension that has no mention of Unicode in the manpage, you need to make sure that the UTF8 flag is stripped off. Note that at the time of this writing (January 2012) the mentioned modules are not UTF-8-aware. Please check the documentation to verify if this is still true.

  ```
  if ($] > 5.008) {
    require Encode;
    $val = Encode::encode_utf8($val); # make octets
  }
  ```

- A scalar we got back from an extension

  If you believe the scalar comes back as UTF-8, you will most likely want the UTF8 flag restored:

  ```
  if ($] > 5.008) {
    require Encode;
    $val = Encode::decode_utf8($val);
  }
  ```

- Same thing, if you are really sure it is UTF-8

  ```
  if ($] > 5.008) {
    require Encode;
    Encode::_utf8_on($val);
  }
  ```

- A wrapper for `DBI` `fetchrow_array` and `fetchrow_hashref`

  When the database contains only UTF-8, a wrapper function or method is a convenient way to replace all your `fetchrow_array` and `fetchrow_hashref` calls. A wrapper function will also make it easier to adapt to future enhancements in your database driver. Note that at the time of this writing (January 2012), the DBI has no standardized way to deal with UTF-8 data. Please check the `DBI` to verify if that is still true.

  ```
  sub fetchrow {
    # $what is one of fetchrow_{array,hashref}
    my($self, $sth, $what) = @_;
    if ($] < 5.008) {
      return $sth->$what;
    } else {
      require Encode;
      if (wantarray) {
        my @arr = $sth->$what;
        for (@arr) {
          defined && /[^\000-\177]/ && Encode::_utf8_on($_);
        }
        return @arr;
      } else {
        my $ret = $sth->$what;
        if (ref $ret) {
  ```

```
            for my $k (keys %$ret) {
              defined
              && /[^\000-\177]/
              && Encode::_utf8_on($_) for $ret->{$k};
            }
            return $ret;
          } else {
            defined && /[^\000-\177]/ && Encode::_utf8_on($_) for $ret;
            return $ret;
          }
        }
      }
    }
```

- A large scalar that you know can only contain ASCII

  Scalars that contain only ASCII and are marked as UTF-8 are sometimes a drag to your program. If you recognize such a situation, just remove the UTF8 flag:

  ```
    utf8::downgrade($val) if $] > 5.008;
  ```

81.3 BUGS

See also Section 81.2.17 [The "Unicode Bug"], page 1340 above.

81.3.1 Interaction with Extensions

When Perl exchanges data with an extension, the extension should be able to understand the UTF8 flag and act accordingly. If the extension doesn't recognize that flag, it's likely that the extension will return incorrectly-flagged data.

So if you're working with Unicode data, consult the documentation of every module you're using if there are any issues with Unicode data exchange. If the documentation does not talk about Unicode at all, suspect the worst and probably look at the source to learn how the module is implemented. Modules written completely in Perl shouldn't cause problems. Modules that directly or indirectly access code written in other programming languages are at risk.

For affected functions, the simple strategy to avoid data corruption is to always make the encoding of the exchanged data explicit. Choose an encoding that you know the extension can handle. Convert arguments passed to the extensions to that encoding and convert results back from that encoding. Write wrapper functions that do the conversions for you, so you can later change the functions when the extension catches up.

To provide an example, let's say the popular `Foo::Bar::escape_html` function doesn't deal with Unicode data yet. The wrapper function would convert the argument to raw UTF-8 and convert the result back to Perl's internal representation like so:

```
sub my_escape_html ($) {
    my($what) = shift;
    return unless defined $what;
    Encode::decode_utf8(Foo::Bar::escape_html(
                                     Encode::encode_utf8($what)));
}
```

Sometimes, when the extension does not convert data but just stores and retrieves it, you will be able to use the otherwise dangerous Section "_utf8_on" in `Encode` function. Let's say the popular `Foo::Bar` extension, written in C, provides a `param` method that lets you store and retrieve data according to these prototypes:

```
$self->param($name, $value);            # set a scalar
$value = $self->param($name);           # retrieve a scalar
```

If it does not yet provide support for any encoding, one could write a derived class with such a `param` method:

```
sub param {
  my($self,$name,$value) = @_;
  utf8::upgrade($name);     # make sure it is UTF-8 encoded
  if (defined $value) {
    utf8::upgrade($value);  # make sure it is UTF-8 encoded
    return $self->SUPER::param($name,$value);
  } else {
    my $ret = $self->SUPER::param($name);
    Encode::_utf8_on($ret); # we know, it is UTF-8 encoded
    return $ret;
  }
}
```

Some extensions provide filters on data entry/exit points, such as `DB_File::filter_store_key` and family. Look out for such filters in the documentation of your extensions; they can make the transition to Unicode data much easier.

81.3.2 Speed

Some functions are slower when working on UTF-8 encoded strings than on byte encoded strings. All functions that need to hop over characters such as `length()`, `substr()` or `index()`, or matching regular expressions can work **much** faster when the underlying data are byte-encoded.

In Perl 5.8.0 the slowness was often quite spectacular; in Perl 5.8.1 a caching scheme was introduced which improved the situation. In general, operations with UTF-8 encoded strings are still slower. As an example, the Unicode properties (character classes) like `\p{Nd}` are known to be quite a bit slower (5-20 times) than their simpler counterparts like `[0-9]` (then again, there are hundreds of Unicode characters matching `Nd` compared with the 10 ASCII characters matching `[0-9]`).

81.4 SEE ALSO

Section 84.1 [perlunitut NAME], page 1367, Section 83.1 [perluniintro NAME], page 1352, `perluniprops`, `Encode`, `open`, `utf8`, `bytes`, Section 68.1 [perlretut NAME], page 1131, [perlvar ${^UNICODE}], page 1407, `http://www.unicode.org/reports/tr44`).

82 perlunifaq

82.1 NAME

perlunifaq - Perl Unicode FAQ

82.2 Q and A

This is a list of questions and answers about Unicode in Perl, intended to be read after Section 84.1 [perlunitut NAME], page 1367.

82.2.1 perlunitut isn't really a Unicode tutorial, is it?

No, and this isn't really a Unicode FAQ.

Perl has an abstracted interface for all supported character encodings, so this is actually a generic `Encode` tutorial and `Encode` FAQ. But many people think that Unicode is special and magical, and I didn't want to disappoint them, so I decided to call the document a Unicode tutorial.

82.2.2 What character encodings does Perl support?

To find out which character encodings your Perl supports, run:

```
    perl -MEncode -le "print for Encode->encodings(':all')"
```

82.2.3 Which version of perl should I use?

Well, if you can, upgrade to the most recent, but certainly `5.8.1` or newer. The tutorial and FAQ assume the latest release.

You should also check your modules, and upgrade them if necessary. For example, HTML::Entities requires version >= 1.32 to function correctly, even though the changelog is silent about this.

82.2.4 What about binary data, like images?

Well, apart from a bare `binmode $fh`, you shouldn't treat them specially. (The binmode is needed because otherwise Perl may convert line endings on Win32 systems.)

Be careful, though, to never combine text strings with binary strings. If you need text in a binary stream, encode your text strings first using the appropriate encoding, then join them with binary strings. See also: "What if I don't encode?".

82.2.5 When should I decode or encode?

Whenever you're communicating text with anything that is external to your perl process, like a database, a text file, a socket, or another program. Even if the thing you're communicating with is also written in Perl.

82.2.6 What if I don't decode?

Whenever your encoded, binary string is used together with a text string, Perl will assume that your binary string was encoded with ISO-8859-1, also known as latin-1. If it wasn't latin-1, then your data is unpleasantly converted. For example, if it was UTF-8, the individual bytes of multibyte characters are seen as separate characters, and then again converted to UTF-8. Such double encoding can be compared to double HTML encoding (`&amp;gt;`), or double URI encoding (`%253E`).

This silent implicit decoding is known as "upgrading". That may sound positive, but it's best to avoid it.

82.2.7 What if I don't encode?

Your text string will be sent using the bytes in Perl's internal format. In some cases, Perl will warn you that you're doing something wrong, with a friendly warning:

```
  Wide character in print at example.pl line 2.
```

Because the internal format is often UTF-8, these bugs are hard to spot, because UTF-8 is usually the encoding you wanted! But don't be lazy, and don't use the fact that Perl's internal format is UTF-8 to your advantage. Encode explicitly to avoid weird bugs, and to show to maintenance programmers that you thought this through.

82.2.8 Is there a way to automatically decode or encode?

If all data that comes from a certain handle is encoded in exactly the same way, you can tell the PerlIO system to automatically decode everything, with the `encoding` layer. If you do this, you can't accidentally forget to decode or encode anymore, on things that use the layered handle.

You can provide this layer when `open`ing the file:

```
open my $fh, '>:encoding(UTF-8)', $filename;  # auto encoding on write
open my $fh, '<:encoding(UTF-8)', $filename;  # auto decoding on read
```

Or if you already have an open filehandle:

```
binmode $fh, ':encoding(UTF-8)';
```

Some database drivers for DBI can also automatically encode and decode, but that is sometimes limited to the UTF-8 encoding.

82.2.9 What if I don't know which encoding was used?

Do whatever you can to find out, and if you have to: guess. (Don't forget to document your guess with a comment.)

You could open the document in a web browser, and change the character set or character encoding until you can visually confirm that all characters look the way they should.

There is no way to reliably detect the encoding automatically, so if people keep sending you data without charset indication, you may have to educate them.

82.2.10 Can I use Unicode in my Perl sources?

Yes, you can! If your sources are UTF-8 encoded, you can indicate that with the `use utf8` pragma.

```
  use utf8;
```

This doesn't do anything to your input, or to your output. It only influences the way your sources are read. You can use Unicode in string literals, in identifiers (but they still have to be "word characters" according to `\w`), and even in custom delimiters.

82.2.11 Data::Dumper doesn't restore the UTF8 flag; is it broken?

No, Data::Dumper's Unicode abilities are as they should be. There have been some complaints that it should restore the UTF8 flag when the data is read again with `eval`. However, you should really not look at the flag, and nothing indicates that Data::Dumper should break this rule.

Here's what happens: when Perl reads in a string literal, it sticks to 8 bit encoding as long as it can. (But perhaps originally it was internally encoded as UTF-8, when you dumped it.) When it has to give that up because other characters are added to the text string, it silently upgrades the string to UTF-8.

If you properly encode your strings for output, none of this is of your concern, and you can just `eval` dumped data as always.

82.2.12 Why do regex character classes sometimes match only in the ASCII range?

Starting in Perl 5.14 (and partially in Perl 5.12), just put a `use feature 'unicode_strings'` near the beginning of your program. Within its lexical scope you shouldn't have this problem. It also is automatically enabled under `use feature ':5.12'` or `use v5.12` or using `-E` on the command line for Perl 5.12 or higher.

The rationale for requiring this is to not break older programs that rely on the way things worked before Unicode came along. Those older programs knew only about the ASCII character set, and so may not work properly for additional characters. When a string is encoded in UTF-8, Perl assumes that the program is prepared to deal with Unicode, but when the string isn't, Perl assumes that only ASCII is wanted, and so those characters that are not ASCII characters aren't recognized as to what they would be in Unicode. `use feature 'unicode_strings'` tells Perl to treat all characters as Unicode, whether the string is encoded in UTF-8 or not, thus avoiding the problem.

However, on earlier Perls, or if you pass strings to subroutines outside the feature's scope, you can force Unicode rules by changing the encoding to UTF-8 by doing `utf8::upgrade($string)`. This can be used safely on any string, as it checks and does not change strings that have already been upgraded.

For a more detailed discussion, see `Unicode-Semantics` on CPAN.

82.2.13 Why do some characters not uppercase or lowercase correctly?

See the answer to the previous question.

82.2.14 How can I determine if a string is a text string or a binary string?

You can't. Some use the UTF8 flag for this, but that's misuse, and makes well behaved modules like Data::Dumper look bad. The flag is useless for this purpose, because it's off when an 8 bit encoding (by default ISO-8859-1) is used to store the string.

This is something you, the programmer, has to keep track of; sorry. You could consider adopting a kind of "Hungarian notation" to help with this.

82.2.15 How do I convert from encoding FOO to encoding BAR?

By first converting the FOO-encoded byte string to a text string, and then the text string to a BAR-encoded byte string:

```
my $text_string = decode('FOO', $foo_string);
my $bar_string  = encode('BAR', $text_string);
```

or by skipping the text string part, and going directly from one binary encoding to the other:

```
use Encode qw(from_to);
from_to($string, 'FOO', 'BAR');  # changes contents of $string
```

or by letting automatic decoding and encoding do all the work:

```
open my $foofh, '<:encoding(FOO)', 'example.foo.txt';
open my $barfh, '>:encoding(BAR)', 'example.bar.txt';
print { $barfh } $_ while <$foofh>;
```

82.2.16 What are `decode_utf8` and `encode_utf8`?

These are alternate syntaxes for `decode('utf8', ...)` and `encode('utf8', ...)`.

82.2.17 What is a "wide character"?

This is a term used for characters occupying more than one byte.

The Perl warning "Wide character in ..." is caused by such a character. With no specified encoding layer, Perl tries to fit things into a single byte. When it can't, it emits this warning (if warnings are enabled), and uses UTF-8 encoded data instead.

To avoid this warning and to avoid having different output encodings in a single stream, always specify an encoding explicitly, for example with a PerlIO layer:

```
binmode STDOUT, ":encoding(UTF-8)";
```

82.3 INTERNALS

82.3.1 What is "the UTF8 flag"?

Please, unless you're hacking the internals, or debugging weirdness, don't think about the UTF8 flag at all. That means that you very probably shouldn't use `is_utf8`, `_utf8_on` or `_utf8_off` at all.

The UTF8 flag, also called SvUTF8, is an internal flag that indicates that the current internal representation is UTF-8. Without the flag, it is assumed to be ISO-8859-1. Perl converts between these automatically. (Actually Perl usually assumes the representation is ASCII; see Section 82.2.12 [Why do regex character classes sometimes match only in the ASCII range?], page 1348 above.)

One of Perl's internal formats happens to be UTF-8. Unfortunately, Perl can't keep a secret, so everyone knows about this. That is the source of much confusion. It's better to pretend that the internal format is some unknown encoding, and that you always have to encode and decode explicitly.

82.3.2 What about the `use bytes` pragma?

Don't use it. It makes no sense to deal with bytes in a text string, and it makes no sense to deal with characters in a byte string. Do the proper conversions (by decoding/encoding), and things will work out well: you get character counts for decoded data, and byte counts for encoded data.

`use bytes` is usually a failed attempt to do something useful. Just forget about it.

82.3.3 What about the `use encoding` pragma?

Don't use it. Unfortunately, it assumes that the programmer's environment and that of the user will use the same encoding. It will use the same encoding for the source code and for STDIN and STDOUT. When a program is copied to another machine, the source code does not change, but the STDIO environment might.

If you need non-ASCII characters in your source code, make it a UTF-8 encoded file and `use utf8`.

If you need to set the encoding for STDIN, STDOUT, and STDERR, for example based on the user's locale, `use open`.

82.3.4 What is the difference between `:encoding` and `:utf8`?

Because UTF-8 is one of Perl's internal formats, you can often just skip the encoding or decoding step, and manipulate the UTF8 flag directly.

Instead of `:encoding(UTF-8)`, you can simply use `:utf8`, which skips the encoding step if the data was already represented as UTF8 internally. This is widely accepted as good behavior when you're writing, but it can be dangerous when reading, because it causes internal inconsistency when you have invalid byte sequences. Using `:utf8` for input can sometimes result in security breaches, so please use `:encoding(UTF-8)` instead.

Instead of `decode` and `encode`, you could use `_utf8_on` and `_utf8_off`, but this is considered bad style. Especially `_utf8_on` can be dangerous, for the same reason that `:utf8` can.

There are some shortcuts for oneliners; see [-C], page 1179 in Section 69.1 [perlrun NAME], page 1176.

82.3.5 What's the difference between `UTF-8` and `utf8`?

`UTF-8` is the official standard. `utf8` is Perl's way of being liberal in what it accepts. If you have to communicate with things that aren't so liberal, you may want to consider using `UTF-8`. If you have to communicate with things that are too liberal, you may have to use `utf8`. The full explanation is in `Encode`.

`UTF-8` is internally known as `utf-8-strict`. The tutorial uses UTF-8 consistently, even where utf8 is actually used internally, because the distinction can be hard to make, and is mostly irrelevant.

For example, utf8 can be used for code points that don't exist in Unicode, like 9999999, but if you encode that to UTF-8, you get a substitution character (by default; see Section "Handling Malformed Data" in `Encode` for more ways of dealing with this.)

Okay, if you insist: the "internal format" is utf8, not UTF-8. (When it's not some other encoding.)

82.3.6 I lost track; what encoding is the internal format really?

It's good that you lost track, because you shouldn't depend on the internal format being any specific encoding. But since you asked: by default, the internal format is either ISO-8859-1 (latin-1), or utf8, depending on the history of the string. On EBCDIC platforms, this may be different even.

Perl knows how it stored the string internally, and will use that knowledge when you `encode`. In other words: don't try to find out what the internal encoding for a certain string is, but instead just encode it into the encoding that you want.

82.4 AUTHOR

Juerd Waalboer <#####@juerd.nl>

82.5 SEE ALSO

Section 81.1 [perlunicode NAME], page 1317, Section 83.1 [perluniintro NAME], page 1352, `Encode`

83 perluniintro

83.1 NAME

perluniintro - Perl Unicode introduction

83.2 DESCRIPTION

This document gives a general idea of Unicode and how to use Unicode in Perl. See Section 83.2.15 [Further Resources], page 1365 for references to more in-depth treatments of Unicode.

83.2.1 Unicode

Unicode is a character set standard which plans to codify all of the writing systems of the world, plus many other symbols.

Unicode and ISO/IEC 10646 are coordinated standards that unify almost all other modern character set standards, covering more than 80 writing systems and hundreds of languages, including all commercially-important modern languages. All characters in the largest Chinese, Japanese, and Korean dictionaries are also encoded. The standards will eventually cover almost all characters in more than 250 writing systems and thousands of languages. Unicode 1.0 was released in October 1991, and 6.0 in October 2010.

A Unicode *character* is an abstract entity. It is not bound to any particular integer width, especially not to the C language `char`. Unicode is language-neutral and display-neutral: it does not encode the language of the text, and it does not generally define fonts or other graphical layout details. Unicode operates on characters and on text built from those characters.

Unicode defines characters like `LATIN CAPITAL LETTER A` or `GREEK SMALL LETTER ALPHA` and unique numbers for the characters, in this case 0x0041 and 0x03B1, respectively. These unique numbers are called *code points*. A code point is essentially the position of the character within the set of all possible Unicode characters, and thus in Perl, the term *ordinal* is often used interchangeably with it.

The Unicode standard prefers using hexadecimal notation for the code points. If numbers like `0x0041` are unfamiliar to you, take a peek at a later section, Section 83.2.14 [Hexadecimal Notation], page 1364. The Unicode standard uses the notation `U+0041 LATIN CAPITAL LETTER A`, to give the hexadecimal code point and the normative name of the character.

Unicode also defines various *properties* for the characters, like "uppercase" or "lowercase", "decimal digit", or "punctuation"; these properties are independent of the names of the characters. Furthermore, various operations on the characters like uppercasing, lowercasing, and collating (sorting) are defined.

A Unicode *logical* "character" can actually consist of more than one internal *actual* "character" or code point. For Western languages, this is adequately modelled by a *base character* (like `LATIN CAPITAL LETTER A`) followed by one or more *modifiers* (like `COMBINING ACUTE ACCENT`). This sequence of base character and modifiers is called a *combining character sequence*. Some non-western languages require more complicated models, so Unicode created the *grapheme cluster* concept, which was later further refined into the *extended* *grapheme cluster*. For example, a Korean Hangul syllable is considered a single logical character, but most often consists of three actual Unicode characters: a leading consonant followed by an interior vowel followed by a trailing consonant.

Whether to call these extended grapheme clusters "characters" depends on your point of view. If you are a programmer, you probably would tend towards seeing each element in the sequences as one unit, or "character". However from the user's point of view, the whole sequence could be seen as one "character" since that's probably what it looks like in the context of the user's language. In this document, we take the programmer's point of view: one "character" is one Unicode code point.

For some combinations of base character and modifiers, there are *precomposed* characters. There is a single character equivalent, for example, for the sequence `LATIN CAPITAL LETTER A` followed by `COMBINING ACUTE ACCENT`. It is called `LATIN CAPITAL LETTER A WITH ACUTE`. These precomposed characters are, however, only available for some combinations, and are mainly meant to support round-trip conversions between Unicode and legacy standards (like ISO 8859). Using sequences, as Unicode does, allows for needing fewer basic building blocks (code points) to express many more potential grapheme clusters. To support conversion between equivalent forms, various *normalization forms* are also defined. Thus, `LATIN CAPITAL LETTER A WITH ACUTE` is in *Normalization Form Composed*, (abbreviated NFC), and the sequence `LATIN CAPITAL LETTER A` followed by `COMBINING ACUTE ACCENT` represents the same character in *Normalization Form Decomposed* (NFD).

Because of backward compatibility with legacy encodings, the "a unique number for every character" idea breaks down a bit: instead, there is "at least one number for every character". The same character could be represented differently in several legacy encodings. The converse is not true: some code points do not have an assigned character. Firstly, there are unallocated code points within otherwise used blocks. Secondly, there are special Unicode control characters that do not represent true characters.

When Unicode was first conceived, it was thought that all the world's characters could be represented using a 16-bit word; that is a maximum of `0x10000` (or 65,536) characters would be needed, from `0x0000` to `0xFFFF`. This soon proved to be wrong, and since Unicode 2.0 (July 1996), Unicode has been defined all the way up to 21 bits (`0x10FFFF`), and Unicode 3.1 (March 2001) defined the first characters above `0xFFFF`. The first `0x10000` characters are called the *Plane 0*, or the *Basic Multilingual Plane* (BMP). With Unicode 3.1, 17 (yes, seventeen) planes in all were defined–but they are nowhere near full of defined characters, yet.

When a new language is being encoded, Unicode generally will choose a `block` of consecutive unallocated code points for its characters. So far, the number of code points in these blocks has always been evenly divisible by 16. Extras in a block, not currently needed, are left unallocated, for future growth. But there have been occasions when a later release needed more code points than the available extras, and a new block had to allocated somewhere else, not contiguous to the initial one, to handle the overflow. Thus, it became apparent early on that "block" wasn't an adequate organizing principal, and so the `Script` property was created. (Later an improved script property was added as well, the `Script_Extensions` property.) Those code points that are in overflow blocks can still have the same script as the original ones. The script concept fits more closely with natural language: there is `Latin` script, `Greek` script, and so on; and there are several artificial scripts, like `Common` for characters that are used in multiple scripts, such as mathematical symbols.

Scripts usually span varied parts of several blocks. For more information about scripts, see Section 81.2.5.3 [perlunicode Scripts], page 1325. The division into blocks exists, but it is almost completely accidental–an artifact of how the characters have been and still are allocated. (Note that this paragraph has oversimplified things for the sake of this being an introduction. Unicode doesn't really encode languages, but the writing systems for them–their scripts; and one script can be used by many languages. Unicode also encodes things that aren't really about languages, such as symbols like `BAGGAGE CLAIM`.)

The Unicode code points are just abstract numbers. To input and output these abstract numbers, the numbers must be *encoded* or *serialised* somehow. Unicode defines several *character encoding forms*, of which *UTF-8* is the most popular. UTF-8 is a variable length encoding that encodes Unicode characters as 1 to 4 bytes. Other encodings include UTF-16 and UTF-32 and their big- and little-endian variants (UTF-8 is byte-order independent). The ISO/IEC 10646 defines the UCS-2 and UCS-4 encoding forms.

For more information about encodings–for instance, to learn what *surrogates* and *byte order marks* (BOMs) are–see Section 81.1 [perlunicode NAME], page 1317.

83.2.2 Perl's Unicode Support

Starting from Perl v5.6.0, Perl has had the capacity to handle Unicode natively. Perl v5.8.0, however, is the first recommended release for serious Unicode work. The maintenance release 5.6.1 fixed many of the problems of the initial Unicode implementation, but for example regular expressions still do not work with Unicode in 5.6.1. Perl v5.14.0 is the first release where Unicode support is (almost) seamlessly integrable without some gotchas (the exception being some differences in [quotemeta], page 430, and that is fixed starting in Perl 5.16.0). To enable this seamless support, you should `use feature 'unicode_strings'` (which is automatically selected if you `use 5.012` or higher). See `feature`. (5.14 also fixes a number of bugs and departures from the Unicode standard.)

Before Perl v5.8.0, the use of `use utf8` was used to declare that operations in the current block or file would be Unicode-aware. This model was found to be wrong, or at least clumsy: the "Unicodeness" is now carried with the data, instead of being attached to the operations. Starting with Perl v5.8.0, only one case remains where an explicit `use utf8` is needed: if your Perl script itself is encoded in UTF-8, you can use UTF-8 in your identifier names, and in string and regular expression literals, by saying `use utf8`. This is not the default because scripts with legacy 8-bit data in them would break. See `utf8`.

83.2.3 Perl's Unicode Model

Perl supports both pre-5.6 strings of eight-bit native bytes, and strings of Unicode characters. The general principle is that Perl tries to keep its data as eight-bit bytes for as long as possible, but as soon as Unicodeness cannot be avoided, the data is transparently upgraded to Unicode. Prior to Perl v5.14.0, the upgrade was not completely transparent (see Section 81.2.17 [perlunicode The "Unicode Bug"], page 1340), and for backwards compatibility, full transparency is not gained unless `use feature 'unicode_strings'` (see `feature`) or `use 5.012` (or higher) is selected.

Internally, Perl currently uses either whatever the native eight-bit character set of the platform (for example Latin-1) is, defaulting to UTF-8, to encode Unicode strings. Specifically, if all code points in the string are `0xFF` or less, Perl uses the native eight-bit character set. Otherwise, it uses UTF-8.

A user of Perl does not normally need to know nor care how Perl happens to encode its internal strings, but it becomes relevant when outputting Unicode strings to a stream without a PerlIO layer (one with the "default" encoding). In such a case, the raw bytes used internally (the native character set or UTF-8, as appropriate for each string) will be used, and a "Wide character" warning will be issued if those strings contain a character beyond 0x00FF.

For example,

```
perl -e 'print "\x{DF}\n", "\x{0100}\x{DF}\n"'
```

produces a fairly useless mixture of native bytes and UTF-8, as well as a warning:

```
Wide character in print at ...
```

To output UTF-8, use the `:encoding` or `:utf8` output layer. Prepending

```
binmode(STDOUT, ":utf8");
```

to this sample program ensures that the output is completely UTF-8, and removes the program's warning.

You can enable automatic UTF-8-ification of your standard file handles, default `open()` layer, and `@ARGV` by using either the `-C` command line switch or the `PERL_UNICODE` environment variable, see Section 69.1 [perlrun NAME], page 1176 for the documentation of the `-C` switch.

Note that this means that Perl expects other software to work the same way: if Perl has been led to believe that STDIN should be UTF-8, but then STDIN coming in from another command is not UTF-8, Perl will likely complain about the malformed UTF-8.

All features that combine Unicode and I/O also require using the new PerlIO feature. Almost all Perl 5.8 platforms do use PerlIO, though: you can see whether yours is by running "perl -V" and looking for `useperlio=define`.

83.2.4 Unicode and EBCDIC

Perl 5.8.0 added support for Unicode on EBCDIC platforms. This support was allowed to lapse in later releases, but was revived in 5.22. Unicode support is somewhat more complex to implement since additional conversions are needed. See Section 19.1 [perlebcdic NAME], page 266 for more information.

On EBCDIC platforms, the internal Unicode encoding form is UTF-EBCDIC instead of UTF-8. The difference is that as UTF-8 is "ASCII-safe" in that ASCII characters encode to UTF-8 as-is, while UTF-EBCDIC is "EBCDIC-safe", in that all the basic characters (which includes all those that have ASCII equivalents (like "`A`", "`0`", "`%`", *etc.*) are the same in both EBCDIC and UTF-EBCDIC. Often, documentation will use the term "UTF-8" to mean UTF-EBCDIC as well. This is the case in this document.

83.2.5 Creating Unicode

This section applies fully to Perls starting with v5.22. Various caveats for earlier releases are in the Section 83.2.5.1 [Earlier releases caveats], page 1356 subsection below.

To create Unicode characters in literals, use the `\N{...}` notation in double-quoted strings:

```
my $smiley_from_name = "\N{WHITE SMILING FACE}";
my $smiley_from_code_point = "\N{U+263a}";
```

Similarly, they can be used in regular expression literals

```
$smiley =~ /\N{WHITE SMILING FACE}/;
$smiley =~ /\N{U+263a}/;
```

At run-time you can use:

```
use charnames ();
my $hebrew_alef_from_name
                    = charnames::string_vianame("HEBREW LETTER ALEF");
my $hebrew_alef_from_code_point = charnames::string_vianame("U+05D0");
```

Naturally, `ord()` will do the reverse: it turns a character into a code point.

There are other runtime options as well. You can use `pack()`:

```
my $hebrew_alef_from_code_point = pack("U", 0x05d0);
```

Or you can use `chr()`, though it is less convenient in the general case:

```
$hebrew_alef_from_code_point = chr(utf8::unicode_to_native(0x05d0));
utf8::upgrade($hebrew_alef_from_code_point);
```

The `utf8::unicode_to_native()` and `utf8::upgrade()` aren't needed if the argument is above 0xFF, so the above could have been written as

```
$hebrew_alef_from_code_point = chr(0x05d0);
```

since 0x5d0 is above 255.

`\x{}` and `\o{}` can also be used to specify code points at compile time in double-quotish strings, but, for backward compatibility with older Perls, the same rules apply as with `chr()` for code points less than 256.

`utf8::unicode_to_native()` is used so that the Perl code is portable to EBCDIC platforms. You can omit it if you're *really* sure no one will ever want to use your code on a non-ASCII platform. Starting in Perl v5.22, calls to it on ASCII platforms are optimized out, so there's no performance penalty at all in adding it. Or you can simply use the other constructs that don't require it.

See Section 83.2.15 [Further Resources], page 1365 for how to find all these names and numeric codes.

83.2.5.1 Earlier releases caveats

On EBCDIC platforms, prior to v5.22, using `\N{U+...}` doesn't work properly.

Prior to v5.16, using `\N{...}` with a character name (as opposed to a `U+...` code point) required a `use charnames :full`.

Prior to v5.14, there were some bugs in `\N{...}` with a character name (as opposed to a `U+...` code point).

`charnames::string_vianame()` was introduced in v5.14. Prior to that, `charnames::vianame()` should work, but only if the argument is of the form `"U+..."`. Your best bet there for runtime Unicode by character name is probably:

```
use charnames ();
my $hebrew_alef_from_name
                 = pack("U", charnames::vianame("HEBREW LETTER ALEF"));
```

83.2.6 Handling Unicode

Handling Unicode is for the most part transparent: just use the strings as usual. Functions like `index()`, `length()`, and `substr()` will work on the Unicode characters; regular expressions will work on the Unicode characters (see Section 81.1 [perlunicode NAME], page 1317 and Section 68.1 [perlretut NAME], page 1131).

Note that Perl considers grapheme clusters to be separate characters, so for example

```
print length("\N{LATIN CAPITAL LETTER A}\N{COMBINING ACUTE ACCENT}"),
      "\n";
```

will print 2, not 1. The only exception is that regular expressions have `\X` for matching an extended grapheme cluster. (Thus `\X` in a regular expression would match the entire sequence of both the example characters.)

Life is not quite so transparent, however, when working with legacy encodings, I/O, and certain special cases:

83.2.7 Legacy Encodings

When you combine legacy data and Unicode, the legacy data needs to be upgraded to Unicode. Normally the legacy data is assumed to be ISO 8859-1 (or EBCDIC, if applicable).

The `Encode` module knows about many encodings and has interfaces for doing conversions between those encodings:

```
use Encode 'decode';
$data = decode("iso-8859-3", $data); # convert from legacy to utf-8
```

83.2.8 Unicode I/O

Normally, writing out Unicode data

```
print FH $some_string_with_unicode, "\n";
```

produces raw bytes that Perl happens to use to internally encode the Unicode string. Perl's internal encoding depends on the system as well as what characters happen to be in the string at the time. If any of the characters are at code points `0x100` or above, you will get a warning. To ensure that the output is explicitly rendered in the encoding you desire–and to avoid the warning–open the stream with the desired encoding. Some examples:

```
open FH, ">:utf8", "file";

open FH, ">:encoding(ucs2)",      "file";
open FH, ">:encoding(UTF-8)",     "file";
open FH, ">:encoding(shift_jis)", "file";
```

and on already open streams, use `binmode()`:

```
binmode(STDOUT, ":utf8");

binmode(STDOUT, ":encoding(ucs2)");
binmode(STDOUT, ":encoding(UTF-8)");
binmode(STDOUT, ":encoding(shift_jis)");
```

The matching of encoding names is loose: case does not matter, and many encodings have several aliases. Note that the `:utf8` layer must always be specified exactly like that;

it is *not* subject to the loose matching of encoding names. Also note that currently `:utf8` is unsafe for input, because it accepts the data without validating that it is indeed valid UTF-8; you should instead use `:encoding(utf-8)` (with or without a hyphen).

See `PerlIO` for the `:utf8` layer, `PerlIO-encoding` and `Encode-PerlIO` for the `:encoding()` layer, and `Encode-Supported` for many encodings supported by the `Encode` module.

Reading in a file that you know happens to be encoded in one of the Unicode or legacy encodings does not magically turn the data into Unicode in Perl's eyes. To do that, specify the appropriate layer when opening files

```
open(my $fh,'<:encoding(utf8)', 'anything');
my $line_of_unicode = <$fh>;

open(my $fh,'<:encoding(Big5)', 'anything');
my $line_of_unicode = <$fh>;
```

The I/O layers can also be specified more flexibly with the `open` pragma. See `open`, or look at the following example.

```
use open ':encoding(utf8)'; # input/output default encoding will be
                            # UTF-8
open X, ">file";
print X chr(0x100), "\n";
close X;
open Y, "<file";
printf "%#x\n", ord(<Y>); # this should print 0x100
close Y;
```

With the `open` pragma you can use the `:locale` layer

```
BEGIN { $ENV{LC_ALL} = $ENV{LANG} = 'ru_RU.KOI8-R' }
# the :locale will probe the locale environment variables like
# LC_ALL
use open OUT => ':locale'; # russki parusski
open(O, ">koi8");
print O chr(0x430); # Unicode CYRILLIC SMALL LETTER A = KOI8-R 0xc1
close O;
open(I, "<koi8");
printf "%#x\n", ord(<I>), "\n"; # this should print 0xc1
close I;
```

These methods install a transparent filter on the I/O stream that converts data from the specified encoding when it is read in from the stream. The result is always Unicode.

The `open` pragma affects all the `open()` calls after the pragma by setting default layers. If you want to affect only certain streams, use explicit layers directly in the `open()` call.

You can switch encodings on an already opened stream by using `binmode()`; see ⟨undefined⟩ [perlfunc binmode], page ⟨undefined⟩.

The `:locale` does not currently work with `open()` and `binmode()`, only with the `open` pragma. The `:utf8` and `:encoding(...)` methods do work with all of `open()`, `binmode()`, and the `open` pragma.

Similarly, you may use these I/O layers on output streams to automatically convert Unicode to the specified encoding when it is written to the stream. For example, the following snippet copies the contents of the file "text.jis" (encoded as ISO-2022-JP, aka JIS) to the file "text.utf8", encoded as UTF-8:

```
open(my $nihongo, '<:encoding(iso-2022-jp)', 'text.jis');
open(my $unicode, '>:utf8',                  'text.utf8');
while (<$nihongo>) { print $unicode $_ }
```

The naming of encodings, both by the `open()` and by the `open` pragma allows for flexible names: `koi8-r` and `KOI8R` will both be understood.

Common encodings recognized by ISO, MIME, IANA, and various other standardisation organisations are recognised; for a more detailed list see `Encode-Supported`.

`read()` reads characters and returns the number of characters. `seek()` and `tell()` operate on byte counts, as do `sysread()` and `sysseek()`.

Notice that because of the default behaviour of not doing any conversion upon input if there is no default layer, it is easy to mistakenly write code that keeps on expanding a file by repeatedly encoding the data:

```
# BAD CODE WARNING
open F, "file";
local $/; ## read in the whole file of 8-bit characters
$t = <F>;
close F;
open F, ">:encoding(utf8)", "file";
print F $t; ## convert to UTF-8 on output
close F;
```

If you run this code twice, the contents of the `file` will be twice UTF-8 encoded. A `use open ':encoding(utf8)'` would have avoided the bug, or explicitly opening also the `file` for input as UTF-8.

NOTE: the `:utf8` and `:encoding` features work only if your Perl has been built with `PerlIO`, which is the default on most systems.

83.2.9 Displaying Unicode As Text

Sometimes you might want to display Perl scalars containing Unicode as simple ASCII (or EBCDIC) text. The following subroutine converts its argument so that Unicode characters with code points greater than 255 are displayed as `\x{...}`, control characters (like `\n`) are displayed as `\x..`, and the rest of the characters as themselves:

```
sub nice_string {
       join("",
       map { $_ > 255                    # if wide character...
             ? sprintf("\\x{%04X}", $_)  # \x{...}
             : chr($_) =~ /[[:cntrl:]]/  # else if control character...
               ? sprintf("\\x%02X", $_)  # \x..
               : quotemeta(chr($_))      # else quoted or as themselves
       } unpack("W*", $_[0]));           # unpack Unicode characters
  }
```

For example,

```
nice_string("foo\x{100}bar\n")
```

returns the string

```
'foo\x{0100}bar\x0A'
```

which is ready to be printed.

(`\\x{}` is used here instead of `\\N{}`, since it's most likely that you want to see what the native values are.)

83.2.10 Special Cases

- Bit Complement Operator ~ And vec()

  The bit complement operator ~ may produce surprising results if used on strings containing characters with ordinal values above 255. In such a case, the results are consistent with the internal encoding of the characters, but not with much else. So don't do that. Similarly for `vec()`: you will be operating on the internally-encoded bit patterns of the Unicode characters, not on the code point values, which is very probably not what you want.

- Peeking At Perl's Internal Encoding

  Normal users of Perl should never care how Perl encodes any particular Unicode string (because the normal ways to get at the contents of a string with Unicode–via input and output–should always be via explicitly-defined I/O layers). But if you must, there are two ways of looking behind the scenes.

  One way of peeking inside the internal encoding of Unicode characters is to use `unpack("C*", ...` to get the bytes of whatever the string encoding happens to be, or `unpack("U0..", ...)` to get the bytes of the UTF-8 encoding:

  ```
  # this prints  c4 80  for the UTF-8 bytes 0xc4 0x80
  print join(" ", unpack("U0(H2)*", pack("U", 0x100))), "\n";
  ```

  Yet another way would be to use the Devel::Peek module:

  ```
  perl -MDevel::Peek -e 'Dump(chr(0x100))'
  ```

  That shows the `UTF8` flag in FLAGS and both the UTF-8 bytes and Unicode characters in `PV`. See also later in this document the discussion about the `utf8::is_utf8()` function.

83.2.11 Advanced Topics

- String Equivalence

  The question of string equivalence turns somewhat complicated in Unicode: what do you mean by "equal"?

  (Is `LATIN CAPITAL LETTER A WITH ACUTE` equal to `LATIN CAPITAL LETTER A`?)

  The short answer is that by default Perl compares equivalence (`eq`, `ne`) based only on code points of the characters. In the above case, the answer is no (because 0x00C1 != 0x0041). But sometimes, any CAPITAL LETTER A's should be considered equal, or even A's of any case.

  The long answer is that you need to consider character normalization and casing issues: see `Unicode-Normalize`, Unicode Technical Report #15, Unicode Normalization Forms (`http://www.unicode.org/unicode/reports/tr15`) and sections on case mapping in the Unicode Standard (`http://www.unicode.org`).

As of Perl 5.8.0, the "Full" case-folding of *Case Mappings/SpecialCasing* is implemented, but bugs remain in `qr//i` with them, mostly fixed by 5.14, and essentially entirely by 5.18.

- String Collation

  People like to see their strings nicely sorted–or as Unicode parlance goes, collated. But again, what do you mean by collate?

  (Does `LATIN CAPITAL LETTER A WITH ACUTE` come before or after `LATIN CAPITAL LETTER A WITH GRAVE`?)

  The short answer is that by default, Perl compares strings (`lt`, `le`, `cmp`, `ge`, `gt`) based only on the code points of the characters. In the above case, the answer is "after", since `0x00C1 > 0x00C0`.

  The long answer is that "it depends", and a good answer cannot be given without knowing (at the very least) the language context. See `Unicode-Collate`, and *Unicode Collation Algorithm* `http://www.unicode.org/unicode/reports/tr10/`

83.2.12 Miscellaneous

- Character Ranges and Classes

  Character ranges in regular expression bracketed character classes (e.g., `/[a-z]/`) and in the `tr///` (also known as `y///`) operator are not magically Unicode-aware. What this means is that `[A-Za-z]` will not magically start to mean "all alphabetic letters" (not that it does mean that even for 8-bit characters; for those, if you are using locales (Section 38.1 [perllocale NAME], page 701), use `/[[:alpha:]]/`; and if not, use the 8-bit-aware property `\p{alpha}`).

  All the properties that begin with `\p` (and its inverse `\P`) are actually character classes that are Unicode-aware. There are dozens of them, see `perluniprops`.

  Starting in v5.22, you can use Unicode code points as the end points of regular expression pattern character ranges, and the range will include all Unicode code points that lie between those end points, inclusive.

  ```
   qr/ [\N{U+03]-\N{U+20}] /x
  ```

  includes the code points `\N{U+03}`, `\N{U+04}`, ..., `\N{U+20}`.

  (It is planned to extend this behavior to ranges in `tr///` in Perl v5.24.)

- String-To-Number Conversions

  Unicode does define several other decimal–and numeric–characters besides the familiar 0 to 9, such as the Arabic and Indic digits. Perl does not support string-to-number conversion for digits other than ASCII `0` to `9` (and ASCII `a` to `f` for hexadecimal). To get safe conversions from any Unicode string, use Section “num()” in `Unicode-UCD`.

83.2.13 Questions With Answers

- Will My Old Scripts Break?

  Very probably not. Unless you are generating Unicode characters somehow, old behaviour should be preserved. About the only behaviour that has changed and which could start generating Unicode is the old behaviour of `chr()` where supplying an argument more than 255 produced a character modulo 255. `chr(300)`, for example, was equal to `chr(45)` or `"-"` (in ASCII), now it is LATIN CAPITAL LETTER I WITH BREVE.

- How Do I Make My Scripts Work With Unicode?

  Very little work should be needed since nothing changes until you generate Unicode data. The most important thing is getting input as Unicode; for that, see the earlier I/O discussion. To get full seamless Unicode support, add `use feature 'unicode_strings'` (or `use 5.012` or higher) to your script.

- How Do I Know Whether My String Is In Unicode?

  You shouldn't have to care. But you may if your Perl is before 5.14.0 or you haven't specified `use feature 'unicode_strings'` or `use 5.012` (or higher) because otherwise the rules for the code points in the range 128 to 255 are different depending on whether the string they are contained within is in Unicode or not. (See Section 81.2.16 [perlunicode When Unicode Does Not Happen], page 1340.)

  To determine if a string is in Unicode, use:

  ```
  print utf8::is_utf8($string) ? 1 : 0, "\n";
  ```

  But note that this doesn't mean that any of the characters in the string are necessary UTF-8 encoded, or that any of the characters have code points greater than 0xFF (255) or even 0x80 (128), or that the string has any characters at all. All the `is_utf8()` does is to return the value of the internal "utf8ness" flag attached to the `$string`. If the flag is off, the bytes in the scalar are interpreted as a single byte encoding. If the flag is on, the bytes in the scalar are interpreted as the (variable-length, potentially multi-byte) UTF-8 encoded code points of the characters. Bytes added to a UTF-8 encoded string are automatically upgraded to UTF-8. If mixed non-UTF-8 and UTF-8 scalars are merged (double-quoted interpolation, explicit concatenation, or printf/sprintf parameter substitution), the result will be UTF-8 encoded as if copies of the byte strings were upgraded to UTF-8: for example,

  ```
  $a = "ab\x80c";
  $b = "\x{100}";
  print "$a = $b\n";
  ```

  the output string will be UTF-8-encoded `ab\x80c = \x{100}\n`, but `$a` will stay byte-encoded.

  Sometimes you might really need to know the byte length of a string instead of the character length. For that use either the `Encode::encode_utf8()` function or the `bytes` pragma and the `length()` function:

  ```
  my $unicode = chr(0x100);
  print length($unicode), "\n"; # will print 1
  require Encode;
  print length(Encode::encode_utf8($unicode)),"\n"; # will print 2
  use bytes;
  print length($unicode), "\n"; # will also print 2
                                # (the 0xC4 0x80 of the UTF-8)
  no bytes;
  ```

- How Do I Find Out What Encoding a File Has?

  You might try `Encode-Guess`, but it has a number of limitations.

- How Do I Detect Data That's Not Valid In a Particular Encoding?

  Use the `Encode` package to try converting it. For example,

  ```
  use Encode 'decode_utf8';

  if (eval { decode_utf8($string, Encode::FB_CROAK); 1 }) {
      # $string is valid utf8
  } else {
      # $string is not valid utf8
  }
  ```

  Or use `unpack` to try decoding it:

  ```
  use warnings;
  @chars = unpack("C0U*", $string_of_bytes_that_I_think_is_utf8);
  ```

  If invalid, a `Malformed UTF-8 character` warning is produced. The "C0" means "process the string character per character". Without that, the `unpack("U*", ...)` would work in `U0` mode (the default if the format string starts with `U`) and it would return the bytes making up the UTF-8 encoding of the target string, something that will always work.

- How Do I Convert Binary Data Into a Particular Encoding, Or Vice Versa?

  This probably isn't as useful as you might think. Normally, you shouldn't need to.

  In one sense, what you are asking doesn't make much sense: encodings are for characters, and binary data are not "characters", so converting "data" into some encoding isn't meaningful unless you know in what character set and encoding the binary data is in, in which case it's not just binary data, now is it?

  If you have a raw sequence of bytes that you know should be interpreted via a particular encoding, you can use `Encode`:

  ```
  use Encode 'from_to';
  from_to($data, "iso-8859-1", "utf-8"); # from latin-1 to utf-8
  ```

  The call to `from_to()` changes the bytes in `$data`, but nothing material about the nature of the string has changed as far as Perl is concerned. Both before and after the call, the string `$data` contains just a bunch of 8-bit bytes. As far as Perl is concerned, the encoding of the string remains as "system-native 8-bit bytes".

  You might relate this to a fictional 'Translate' module:

  ```
  use Translate;
  my $phrase = "Yes";
  Translate::from_to($phrase, 'english', 'deutsch');
  ## phrase now contains "Ja"
  ```

  The contents of the string changes, but not the nature of the string. Perl doesn't know any more after the call than before that the contents of the string indicates the affirmative.

  Back to converting data. If you have (or want) data in your system's native 8-bit encoding (e.g. Latin-1, EBCDIC, etc.), you can use pack/unpack to convert to/from Unicode.

```
    $native_string  = pack("W*", unpack("U*", $Unicode_string));
    $Unicode_string = pack("U*", unpack("W*", $native_string));
```

If you have a sequence of bytes you **know** is valid UTF-8, but Perl doesn't know it yet, you can make Perl a believer, too:

```
    use Encode 'decode_utf8';
    $Unicode = decode_utf8($bytes);
```

or:

```
    $Unicode = pack("U0a*", $bytes);
```

You can find the bytes that make up a UTF-8 sequence with

```
    @bytes = unpack("C*", $Unicode_string)
```

and you can create well-formed Unicode with

```
    $Unicode_string = pack("U*", 0xff, ...)
```

- How Do I Display Unicode? How Do I Input Unicode?

  See http://www.alanwood.net/unicode/ and http://www.cl.cam.ac.uk/~mgk25/unicode.html

- How Does Unicode Work With Traditional Locales?

  If your locale is a UTF-8 locale, starting in Perl v5.20, Perl works well for all categories except LC_COLLATE dealing with sorting and the cmp operator.

  For other locales, starting in Perl 5.16, you can specify

  ```
      use locale ':not_characters';
  ```

  to get Perl to work well with them. The catch is that you have to translate from the locale character set to/from Unicode yourself. See Section 83.2.8 [Unicode I/O], page 1357 above for how to

  ```
      use open ':locale';
  ```

  to accomplish this, but full details are in Section 38.10 [perllocale Unicode and UTF-8], page 721, including gotchas that happen if you don't specify :not_characters.

83.2.14 Hexadecimal Notation

The Unicode standard prefers using hexadecimal notation because that more clearly shows the division of Unicode into blocks of 256 characters. Hexadecimal is also simply shorter than decimal. You can use decimal notation, too, but learning to use hexadecimal just makes life easier with the Unicode standard. The U+HHHH notation uses hexadecimal, for example.

The 0x prefix means a hexadecimal number, the digits are 0-9 *and* a-f (or A-F, case doesn't matter). Each hexadecimal digit represents four bits, or half a byte. print 0x..., "\n" will show a hexadecimal number in decimal, and printf "%x\n", $decimal will show a decimal number in hexadecimal. If you have just the "hex digits" of a hexadecimal number, you can use the hex() function.

```
print 0x0009, "\n";    # 9
print 0x000a, "\n";    # 10
print 0x000f, "\n";    # 15
print 0x0010, "\n";    # 16
print 0x0011, "\n";    # 17
```

```
print 0x0100, "\n";    # 256

print 0x0041, "\n";    # 65

printf "%x\n",  65;    # 41
printf "%#x\n", 65;    # 0x41

print hex("41"), "\n"; # 65
```

83.2.15 Further Resources

- Unicode Consortium

  http://www.unicode.org/
- Unicode FAQ

  http://www.unicode.org/unicode/faq/
- Unicode Glossary

  http://www.unicode.org/glossary/
- Unicode Recommended Reading List

  The Unicode Consortium has a list of articles and books, some of which give a much more in depth treatment of Unicode: http://unicode.org/resources/readinglist.html
- Unicode Useful Resources

  http://www.unicode.org/unicode/onlinedat/resources.html
- Unicode and Multilingual Support in HTML, Fonts, Web Browsers and Other Applications

  http://www.alanwood.net/unicode/
- UTF-8 and Unicode FAQ for Unix/Linux

  http://www.cl.cam.ac.uk/~mgk25/unicode.html
- Legacy Character Sets

  http://www.czyborra.com/ http://www.eki.ee/letter/
- You can explore various information from the Unicode data files using the Unicode::UCD module.

83.3 UNICODE IN OLDER PERLS

If you cannot upgrade your Perl to 5.8.0 or later, you can still do some Unicode processing by using the modules Unicode::String, Unicode::Map8, and Unicode::Map, available from CPAN. If you have the GNU recode installed, you can also use the Perl front-end Convert::Recode for character conversions.

The following are fast conversions from ISO 8859-1 (Latin-1) bytes to UTF-8 bytes and back, the code works even with older Perl 5 versions.

```
# ISO 8859-1 to UTF-8
s/([\x80-\xFF])/chr(0xC0|ord($1)>>6).chr(0x80|ord($1)&0x3F)/eg;
```

```
# UTF-8 to ISO 8859-1
s/([\xC2\xC3])([\x80-\xBF])/chr(ord($1)<<6&0xC0|ord($2)&0x3F)/eg;
```

83.4 SEE ALSO

Section 84.1 [perlunitut NAME], page 1367, Section 81.1 [perlunicode NAME], page 1317, `Encode`, `open`, `utf8`, `bytes`, Section 68.1 [perlretut NAME], page 1131, Section 69.1 [perlrun NAME], page 1176, `Unicode-Collate`, `Unicode-Normalize`, `Unicode-UCD`

83.5 ACKNOWLEDGMENTS

Thanks to the kind readers of the perl5-porters@perl.org, perl-unicode@perl.org, linux-utf8@nl.linux.org, and unicore@unicode.org mailing lists for their valuable feedback.

83.6 AUTHOR, COPYRIGHT, AND LICENSE

Copyright 2001-2011 Jarkko Hietaniemi <jhi@iki.fi>. Now maintained by Perl 5 Porters.

This document may be distributed under the same terms as Perl itself.

84 perlunitut

84.1 NAME

perlunitut - Perl Unicode Tutorial

84.2 DESCRIPTION

The days of just flinging strings around are over. It's well established that modern programs need to be capable of communicating funny accented letters, and things like euro symbols. This means that programmers need new habits. It's easy to program Unicode capable software, but it does require discipline to do it right.

There's a lot to know about character sets, and text encodings. It's probably best to spend a full day learning all this, but the basics can be learned in minutes.

These are not the very basics, though. It is assumed that you already know the difference between bytes and characters, and realise (and accept!) that there are many different character sets and encodings, and that your program has to be explicit about them. Recommended reading is "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)" by Joel Spolsky, at `http://joelonsoftware.com/articles/Unicode.html`.

This tutorial speaks in rather absolute terms, and provides only a limited view of the wealth of character string related features that Perl has to offer. For most projects, this information will probably suffice.

84.2.1 Definitions

It's important to set a few things straight first. This is the most important part of this tutorial. This view may conflict with other information that you may have found on the web, but that's mostly because many sources are wrong.

You may have to re-read this entire section a few times...

84.2.1.1 Unicode

Unicode is a character set with room for lots of characters. The ordinal value of a character is called a **code point**. (But in practice, the distinction between code point and character is blurred, so the terms often are used interchangeably.)

There are many, many code points, but computers work with bytes, and a byte has room for only 256 values. Unicode has many more characters than that, so you need a method to make these accessible.

Unicode is encoded using several competing encodings, of which UTF-8 is the most used. In a Unicode encoding, multiple subsequent bytes can be used to store a single code point, or simply: character.

84.2.1.2 UTF-8

UTF-8 is a Unicode encoding. Many people think that Unicode and UTF-8 are the same thing, but they're not. There are more Unicode encodings, but much of the world has standardized on UTF-8.

UTF-8 treats the first 128 codepoints, 0..127, the same as ASCII. They take only one byte per character. All other characters are encoded as two to four bytes using a complex scheme. Fortunately, Perl handles this for us, so we don't have to worry about this.

84.2.1.3 Text strings (character strings)

Text strings, or **character strings** are made of characters. Bytes are irrelevant here, and so are encodings. Each character is just that: the character.

On a text string, you would do things like:

```
$text =~ s/foo/bar/;
if ($string =~ /^\d+$/) { ... }
$text = ucfirst $text;
my $character_count = length $text;
```

The value of a character (`ord`, `chr`) is the corresponding Unicode code point.

84.2.1.4 Binary strings (byte strings)

Binary strings, or **byte strings** are made of bytes. Here, you don't have characters, just bytes. All communication with the outside world (anything outside of your current Perl process) is done in binary.

On a binary string, you would do things like:

```
my (@length_content) = unpack "(V/a)*", $binary;
$binary =~ s/\x00\x0F/\xFF\xF0/;  # for the brave :)
print {$fh} $binary;
my $byte_count = length $binary;
```

84.2.1.5 Encoding

Encoding (as a verb) is the conversion from *text* to *binary*. To encode, you have to supply the target encoding, for example `iso-8859-1` or `UTF-8`. Some encodings, like the `iso-8859` ("latin") range, do not support the full Unicode standard; characters that can't be represented are lost in the conversion.

84.2.1.6 Decoding

Decoding is the conversion from *binary* to *text*. To decode, you have to know what encoding was used during the encoding phase. And most of all, it must be something decodable. It doesn't make much sense to decode a PNG image into a text string.

84.2.1.7 Internal format

Perl has an **internal format**, an encoding that it uses to encode text strings so it can store them in memory. All text strings are in this internal format. In fact, text strings are never in any other format!

You shouldn't worry about what this format is, because conversion is automatically done when you decode or encode.

84.2.2 Your new toolkit

Add to your standard heading the following line:

```
use Encode qw(encode decode);
```

Or, if you're lazy, just:

```
use Encode;
```

84.2.3 I/O flow (the actual 5 minute tutorial)

The typical input/output flow of a program is:

```
1. Receive and decode
2. Process
3. Encode and output
```

If your input is binary, and is supposed to remain binary, you shouldn't decode it to a text string, of course. But in all other cases, you should decode it.

Decoding can't happen reliably if you don't know how the data was encoded. If you get to choose, it's a good idea to standardize on UTF-8.

```
my $foo   = decode('UTF-8', get 'http://example.com/');
my $bar   = decode('ISO-8859-1', readline STDIN);
my $xyzzy = decode('Windows-1251', $cgi->param('foo'));
```

Processing happens as you knew before. The only difference is that you're now using characters instead of bytes. That's very useful if you use things like `substr`, or `length`.

It's important to realize that there are no bytes in a text string. Of course, Perl has its internal encoding to store the string in memory, but ignore that. If you have to do anything with the number of bytes, it's probably best to move that part to step 3, just after you've encoded the string. Then you know exactly how many bytes it will be in the destination string.

The syntax for encoding text strings to binary strings is as simple as decoding:

```
$body = encode('UTF-8', $body);
```

If you needed to know the length of the string in bytes, now's the perfect time for that. Because `$body` is now a byte string, `length` will report the number of bytes, instead of the number of characters. The number of characters is no longer known, because characters only exist in text strings.

```
my $byte_count = length $body;
```

And if the protocol you're using supports a way of letting the recipient know which character encoding you used, please help the receiving end by using that feature! For example, E-mail and HTTP support MIME headers, so you can use the `Content-Type` header. They can also have `Content-Length` to indicate the number of *bytes*, which is always a good idea to supply if the number is known.

```
"Content-Type: text/plain; charset=UTF-8",
"Content-Length: $byte_count"
```

84.3 SUMMARY

Decode everything you receive, encode everything you send out. (If it's text data.)

84.4 Q and A (or FAQ)

After reading this document, you ought to read Section 82.1 [perlunifaq NAME], page 1346 too, then Section 83.1 [perluniintro NAME], page 1352.

84.5 ACKNOWLEDGEMENTS

Thanks to Johan Vromans from Squirrel Consultancy. His UTF-8 rants during the Amsterdam Perl Mongers meetings got me interested and determined to find out how to use character encodings in Perl in ways that don't break easily.

Thanks to Gerard Goossen from TTY. His presentation "UTF-8 in the wild" (Dutch Perl Workshop 2006) inspired me to publish my thoughts and write this tutorial.

Thanks to the people who asked about this kind of stuff in several Perl IRC channels, and have constantly reminded me that a simpler explanation was needed.

Thanks to the people who reviewed this document for me, before it went public. They are: Benjamin Smith, Jan-Pieter Cornet, Johan Vromans, Lukas Mai, Nathan Gray.

84.6 AUTHOR

Juerd Waalboer <#####@juerd.nl>

84.7 SEE ALSO

Section 82.1 [perlunifaq NAME], page 1346, Section 81.1 [perlunicode NAME], page 1317, Section 83.1 [perluniintro NAME], page 1352, `Encode`

85 perlutil

85.1 NAME

perlutil - utilities packaged with the Perl distribution

85.2 DESCRIPTION

Along with the Perl interpreter itself, the Perl distribution installs a range of utilities on your system. There are also several utilities which are used by the Perl distribution itself as part of the install process. This document exists to list all of these utilities, explain what they are for and provide pointers to each module's documentation, if appropriate.

85.3 LIST OF UTILITIES

85.3.1 Documentation

`perldoc`

> The main interface to Perl's documentation is `perldoc`, although if you're reading this, it's more than likely that you've already found it. `perldoc` will extract and format the documentation from any file in the current directory, any Perl module installed on the system, or any of the standard documentation pages, such as this one. Use `perldoc <name>` to get information on any of the utilities described in this document.

`pod2man` and `pod2text`

> If it's run from a terminal, `perldoc` will usually call `pod2man` to translate POD (Plain Old Documentation - see Section 52.1 [perlpod NAME], page 900 for an explanation) into a manpage, and then run `man` to display it; if `man` isn't available, `pod2text` will be used instead and the output piped through your favourite pager.

`pod2html`

> As well as these two, there is another converter: `pod2html` will produce HTML pages from POD.

`pod2usage`

> If you just want to know how to use the utilities described here, `pod2usage` will just extract the "USAGE" section; some of the utilities will automatically call `pod2usage` on themselves when you call them with `-help`.

`podselect`

> `pod2usage` is a special case of `podselect`, a utility to extract named sections from documents written in POD. For instance, while utilities have "USAGE" sections, Perl modules usually have "SYNOPSIS" sections: `podselect -s "SYNOPSIS" ...` will extract this section for a given file.

`podchecker`

> If you're writing your own documentation in POD, the `podchecker` utility will look for errors in your markup.

splain

splain is an interface to Section 16.1 [perldiag NAME], page 137 - paste in your error message to it, and it'll explain it for you.

roffitall

The roffitall utility is not installed on your system but lives in the pod/ directory of your Perl source kit; it converts all the documentation from the distribution to *roff format, and produces a typeset PostScript or text file of the whole lot.

85.3.2 Converters

To help you convert legacy programs to more modern Perl, the pl2pm utility will help you convert old-style Perl 4 libraries to new-style Perl5 modules.

85.3.3 Administration

libnetcfg

To display and change the libnet configuration run the libnetcfg command.

perlivp

The perlivp program is set up at Perl source code build time to test the Perl version it was built under. It can be used after running make install (or your platform's equivalent procedure) to verify that perl and its libraries have been installed correctly.

85.3.4 Development

There are a set of utilities which help you in developing Perl programs, and in particular, extending Perl with C.

perlbug

perlbug is the recommended way to report bugs in the perl interpreter itself or any of the standard library modules back to the developers; please read through the documentation for perlbug thoroughly before using it to submit a bug report.

perlbug

This program provides an easy way to send a thank-you message back to the authors and maintainers of perl. It's just perlbug installed under another name.

h2ph

Back before Perl had the XS system for connecting with C libraries, programmers used to get library constants by reading through the C header files. You may still see require 'syscall.ph' or similar around - the .ph file should be created by running h2ph on the corresponding .h file. See the h2ph documentation for more on how to convert a whole bunch of header files at once.

c2ph and pstruct

c2ph and pstruct, which are actually the same program but behave differently depending on how they are called, provide another way of getting at C with

Perl - they'll convert C structures and union declarations to Perl code. This is deprecated in favour of `h2xs` these days.

`h2xs`

`h2xs` converts C header files into XS modules, and will try and write as much glue between C libraries and Perl modules as it can. It's also very useful for creating skeletons of pure Perl modules.

`enc2xs`

`enc2xs` builds a Perl extension for use by Encode from either Unicode Character Mapping files (.ucm) or Tcl Encoding Files (.enc). Besides being used internally during the build process of the Encode module, you can use `enc2xs` to add your own encoding to perl. No knowledge of XS is necessary.

`xsubpp`

`xsubpp` is a compiler to convert Perl XS code into C code. It is typically run by the makefiles created by `ExtUtils-MakeMaker`.

`xsubpp` will compile XS code into C code by embedding the constructs necessary to let C functions manipulate Perl values and creates the glue necessary to let Perl access those functions.

`prove`

`prove` is a command-line interface to the test-running functionality of `Test::Harness`. It's an alternative to `make test`.

`corelist`

A command-line front-end to `Module::CoreList`, to query what modules were shipped with given versions of perl.

85.3.5 General tools

A few general-purpose tools are shipped with perl, mostly because they came along modules included in the perl distribution.

`piconv`

piconv is a Perl version of **iconv**, a character encoding converter widely available for various Unixen today. This script was primarily a technology demonstrator for Perl v5.8.0, but you can use piconv in the place of iconv for virtually any case.

`ptar`

`ptar` is a tar-like program, written in pure Perl.

`ptardiff`

`ptardiff` is a small utility that produces a diff between an extracted archive and an unextracted one. (Note that this utility requires the `Text::Diff` module to function properly; this module isn't distributed with perl, but is available from the CPAN.)

`ptargrep`

`ptargrep` is a utility to apply pattern matching to the contents of files in a tar archive.

`shasum`

This utility, that comes with the `Digest::SHA` module, is used to print or verify SHA checksums.

`zipdetails`

`zipdetails` displays information about the internal record structure of the zip file. It is not concerned with displaying any details of the compressed data stored in the zip file.

85.3.6 Installation

These utilities help manage extra Perl modules that don't come with the perl distribution.

`cpan`

`cpan` is a command-line interface to CPAN.pm. It allows you to install modules or distributions from CPAN, or just get information about them, and a lot more. It is similar to the command line mode of the `CPAN` module,

```
perl -MCPAN -e shell
```

`instmodsh`

A little interface to ExtUtils::Installed to examine installed modules, validate your packlists and even create a tarball from an installed module.

85.4 SEE ALSO

`perldoc`, `pod2man`, Section 52.1 [perlpod NAME], page 900, `pod2html`, `pod2usage`, `podselect`, `podchecker`, `splain`, Section 16.1 [perldiag NAME], page 137, `roffitall|roffitall`, `File-Find`, `pl2pm`, `perlbug`, `h2ph`, `c2ph`, `h2xs`, `enc2xs`, `xsubpp`, `cpan`, `instmodsh`, `piconv`, `prove`, `corelist`, `ptar`, `ptardiff`, `shasum`, `zipdetails`

86 perlvar

86.1 NAME

perlvar - Perl predefined variables

86.2 DESCRIPTION

86.2.1 The Syntax of Variable Names

Variable names in Perl can have several formats. Usually, they must begin with a letter or underscore, in which case they can be arbitrarily long (up to an internal limit of 251 characters) and may contain letters, digits, underscores, or the special sequence `::` or `'`. In this case, the part before the last `::` or `'` is taken to be a *package qualifier*; see Section 40.1 [perlmod NAME], page 732.

Perl variable names may also be a sequence of digits or a single punctuation or control character (with the literal control character form deprecated). These names are all reserved for special uses by Perl; for example, the all-digits names are used to hold data captured by backreferences after a regular expression match. Perl has a special syntax for the single-control-character names: It understands `^X` (caret `X`) to mean the control-`X` character. For example, the notation `$^W` (dollar-sign caret `W`) is the scalar variable whose name is the single character control-`W`. This is better than typing a literal control-`W` into your program.

Since Perl v5.6.0, Perl variable names may be alphanumeric strings that begin with a caret (or a control character, but this form is deprecated). These variables must be written in the form `${^Foo}`; the braces are not optional. `${^Foo}` denotes the scalar variable whose name is a control-`F` followed by two `o`'s. These variables are reserved for future special uses by Perl, except for the ones that begin with `^_` (control-underscore or caret-underscore). No control-character name that begins with `^_` will acquire a special meaning in any future version of Perl; such names may therefore be used safely in programs. `$^_` itself, however, *is* reserved.

Perl identifiers that begin with digits, control characters, or punctuation characters are exempt from the effects of the `package` declaration and are always forced to be in package `main`; they are also exempt from `strict 'vars'` errors. A few other names are also exempt in these ways:

```
ENV      STDIN
INC      STDOUT
ARGV     STDERR
ARGVOUT
SIG
```

In particular, the special `${^_XYZ}` variables are always taken to be in package `main`, regardless of any `package` declarations presently in scope.

86.3 SPECIAL VARIABLES

The following names have special meaning to Perl. Most punctuation names have reasonable mnemonics, or analogs in the shells. Nevertheless, if you wish to use long variable names, you need only say:

```
use English;
```

at the top of your program. This aliases all the short names to the long names in the current package. Some even have medium names, generally borrowed from **awk**. For more info, please see `English`.

Before you continue, note the sort order for variables. In general, we first list the variables in case-insensitive, almost-lexigraphical order (ignoring the `{` or `^` preceding words, as in `${^UNICODE}` or `$^T`), although `$_` and `@_` move up to the top of the pile. For variables with the same identifier, we list it in order of scalar, array, hash, and bareword.

86.3.1 General Variables

$ARG

$_

The default input and pattern-searching space. The following pairs are equivalent:

```
while (<>) {...}    # equivalent only in while!
while (defined($_ = <>)) {...}

/^Subject:/
$_ =~ /^Subject:/

tr/a-z/A-Z/
$_ =~ tr/a-z/A-Z/

chomp
chomp($_)
```

Here are the places where Perl will assume `$_` even if you don't use it:

- The following functions use `$_` as a default argument:

  abs, alarm, chomp, chop, chr, chroot, cos, defined, eval, evalbytes, exp, fc, glob, hex, int, lc, lcfirst, length, log, lstat, mkdir, oct, ord, pos, print, printf, quotemeta, readlink, readpipe, ref, require, reverse (in scalar context only), rmdir, say, sin, split (for its second argument), sqrt, stat, study, uc, ucfirst, unlink, unpack.
- All file tests (`-f`, `-d`) except for `-t`, which defaults to STDIN. See [perlfunc -X], page 354
- The pattern matching operations `m//`, `s///` and `tr///` (aka `y///`) when used without an `=~` operator.
- The default iterator variable in a `foreach` loop if no other variable is supplied.
- The implicit iterator variable in the `grep()` and `map()` functions.
- The implicit variable of `given()`.
- The default place to put the next value or input record when a `<FH>`, `readline`, `readdir` or `each` operation's result is tested by itself as the sole criterion of a `while` test. Outside a `while` test, this will not happen.

`$_` is by default a global variable. However, as of perl v5.10.0, you can use a lexical version of `$_` by declaring it in a file or in a block with `my`. Moreover, declaring `our $_` restores the global `$_` in the current scope. Though this seemed like a good idea at the time it was introduced, lexical `$_` actually causes more problems than it solves. If you call a function that expects to be passed information via `$_`, it may or may not work, depending on how the function is written, there not being any easy way to solve this. Just avoid lexical `$_`, unless you are feeling particularly masochistic. For this reason lexical `$_` is still experimental and will produce a warning unless warnings have been disabled. As with other experimental features, the behavior of lexical `$_` is subject to change without notice, including change into a fatal error.

Mnemonic: underline is understood in certain operations.

@ARG

@_

Within a subroutine the array `@_` contains the parameters passed to that subroutine. Inside a subroutine, `@_` is the default array for the array operators `push`, `pop`, `shift`, and `unshift`.

See Section 73.1 [perlsub NAME], page 1216.

$LIST_SEPARATOR
$"

When an array or an array slice is interpolated into a double-quoted string or a similar context such as `/.../`, its elements are separated by this value. Default is a space. For example, this:

```
    print "The array is: @array\n";
```

is equivalent to this:

```
    print "The array is: " . join($", @array) . "\n";
```

Mnemonic: works in double-quoted context.

$PROCESS_ID
$PID

$$

The process number of the Perl running this script. Though you *can* set this variable, doing so is generally discouraged, although it can be invaluable for some testing purposes. It will be reset automatically across `fork()` calls.

Note for Linux and Debian GNU/kFreeBSD users: Before Perl v5.16.0 perl would emulate POSIX semantics on Linux systems using LinuxThreads, a partial implementation of POSIX Threads that has since been superseded by the Native POSIX Thread Library (NPTL).

LinuxThreads is now obsolete on Linux, and caching `getpid()` like this made embedding perl unnecessarily complex (since you'd have to manually update the value of $$), so now `$$` and `getppid()` will always return the same values as the underlying C library.

Debian GNU/kFreeBSD systems also used LinuxThreads up until and including the 6.0 release, but after that moved to FreeBSD thread semantics, which are POSIX-like.

To see if your system is affected by this discrepancy check if `getconf GNU_LIBPTHREAD_VERSION | grep -q NPTL` returns a false value. NTPL threads preserve the POSIX semantics.

Mnemonic: same as shells.

$PROGRAM_NAME
$0

Contains the name of the program being executed.

On some (but not all) operating systems assigning to `$0` modifies the argument area that the `ps` program sees. On some platforms you may have to use special `ps` options or a different `ps` to see the changes. Modifying the `$0` is more useful as a way of indicating the current program state than it is for hiding the program you're running.

Note that there are platform-specific limitations on the maximum length of `$0`. In the most extreme case it may be limited to the space occupied by the original `$0`.

In some platforms there may be arbitrary amount of padding, for example space characters, after the modified name as shown by `ps`. In some platforms this padding may extend all the way to the original length of the argument area, no matter what you do (this is the case for example with Linux 2.2).

Note for BSD users: setting `$0` does not completely remove "perl" from the ps(1) output. For example, setting `$0` to `"foobar"` may result in `"perl: foobar (perl)"` (whether both the `"perl: "` prefix and the " (perl)" suffix are shown depends on your exact BSD variant and version). This is an operating system feature, Perl cannot help it.

In multithreaded scripts Perl coordinates the threads so that any thread may modify its copy of the `$0` and the change becomes visible to ps(1) (assuming the operating system plays along). Note that the view of `$0` the other threads have will not change since they have their own copies of it.

If the program has been given to perl via the switches `-e` or `-E`, `$0` will contain the string `"-e"`.

On Linux as of perl v5.14.0 the legacy process name will be set with `prctl(2)`, in addition to altering the POSIX name via `argv[0]` as perl has done since version 4.000. Now system utilities that read the legacy process name such as ps, top and killall will recognize the name you set when assigning to `$0`. The string you supply will be cut off at 16 bytes, this is a limitation imposed by Linux.

Mnemonic: same as **sh** and **ksh**.

$REAL_GROUP_ID
$GID

$(

The real gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by `getgid()`, and the subsequent ones by `getgroups()`, one of which may be the same as the first number.

However, a value assigned to `$(` must be a single number used to set the real gid. So the value given by `$(` should *not* be assigned back to `$(` without being forced numeric, such as by adding zero. Note that this is different to the effective gid (`$)`) which does take a list.

You can change both the real gid and the effective gid at the same time by using `POSIX::setgid()`. Changes to `$(` require a check to `$!` to detect any possible errors after an attempted change.

Mnemonic: parentheses are used to *group* things. The real gid is the group you *left*, if you're running setgid.

$EFFECTIVE_GROUP_ID
$EGID

$)

The effective gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by `getegid()`, and the subsequent ones by `getgroups()`, one of which may be the same as the first number.

Similarly, a value assigned to `$)` must also be a space-separated list of numbers. The first number sets the effective gid, and the rest (if any) are passed to `setgroups()`. To get the effect of an empty list for `setgroups()`, just repeat the new effective gid; that is, to force an effective gid of 5 and an effectively empty `setgroups()` list, say `$) = "5 5"` .

You can change both the effective gid and the real gid at the same time by using `POSIX::setgid()` (use only a single numeric argument). Changes to `$)` require a check to `$!` to detect any possible errors after an attempted change.

`$<`, `$>`, `$(` and `$)` can be set only on machines that support the corresponding *set[re][ug]id()* routine. `$(` and `$)` can be swapped only on machines supporting `setregid()`.

Mnemonic: parentheses are used to *group* things. The effective gid is the group that's *right* for you, if you're running setgid.

$REAL_USER_ID
$UID

$<

The real uid of this process. You can change both the real uid and the effective uid at the same time by using `POSIX::setuid()`. Since changes to `$<` require a system call, check `$!` after a change attempt to detect any possible errors.

Mnemonic: it's the uid you came *from*, if you're running setuid.

$EFFECTIVE_USER_ID
$EUID

$> >>

The effective uid of this process. For example:

```
$< = $>;            # set real to effective uid
($<,$>) = ($>,$<);  # swap real and effective uids
```

You can change both the effective uid and the real uid at the same time by using `POSIX::setuid()`. Changes to `$>` require a check to `$!` to detect any possible errors after an attempted change.

`$<` and `$>` can be swapped only on machines supporting `setreuid()`.

Mnemonic: it's the uid you went *to*, if you're running setuid.

$SUBSCRIPT_SEPARATOR
$SUBSEP

$;

The subscript separator for multidimensional array emulation. If you refer to a hash element as

```
$foo{$x,$y,$z}
```

it really means

```
$foo{join($;, $x, $y, $z)}
```

But don't put

```
@foo{$x,$y,$z}      # a slice--note the @
```

which means

```
($foo{$x},$foo{$y},$foo{$z})
```

Default is `"\034"`, the same as SUBSEP in **awk**. If your keys contain binary data there might not be any safe value for `$;`.

Consider using `"real"` multidimensional arrays as described in Section 39.1 [perllol NAME], page 725.

Mnemonic: comma (the syntactic subscript separator) is a semi-semicolon.

$a

$b

Special package variables when using `sort()`, see ⟨undefined⟩ [perlfunc sort], page ⟨undefined⟩. Because of this specialness `$a` and `$b` don't need to be declared (using `use vars`, or `our()`) even when using the `strict 'vars'` pragma. Don't lexicalize them with `my $a` or `my $b` if you want to be able to use them in the `sort()` comparison block or function.

%ENV

The hash `%ENV` contains your current environment. Setting a value in `ENV` changes the environment for any child processes you subsequently `fork()` off.

As of v5.18.0, both keys and values stored in `%ENV` are stringified.

```
my $foo = 1;
$ENV{'bar'} = \$foo;
if( ref $ENV{'bar'} ) {
    say "Pre 5.18.0 Behaviour";
} else {
    say "Post 5.18.0 Behaviour";
}
```

Previously, only child processes received stringified values:

```
my $foo = 1;
$ENV{’bar’} = \$foo;

# Always printed ’non ref’
system($^X, ’-e’,
       q/print ( ref $ENV{’bar’}  ? ’ref’ : ’non ref’ ) /);
```

This happens because you can’t really share arbitrary data structures with foreign processes.

$OLD_PERL_VERSION
$]

The revision, version, and subversion of the Perl interpreter, represented as a decimal of the form 5.XXXYYY, where XXX is the version / 1e3 and YYY is the subversion / 1e6. For example, Perl v5.10.1 would be "5.010001".

This variable can be used to determine whether the Perl interpreter executing a script is in the right range of versions:

```
warn "No PerlIO!\n" if $] lt ’5.008’;
```

When comparing `$]`, string comparison operators are **highly recommended**. The inherent limitations of binary floating point representation can sometimes lead to incorrect comparisons for some numbers on some architectures.

See also the documentation of `use VERSION` and `require VERSION` for a convenient way to fail if the running Perl interpreter is too old.

See [$^V], page 1385 for a representation of the Perl version as a `version` object, which allows more flexible string comparisons.

The main advantage of `$]` over `$^V` is that it works the same on any version of Perl. The disadvantages are that it can’t easily be compared to versions in other formats (e.g. literal v-strings, "v1.2.3" or version objects) and numeric comparisons can occasionally fail; it’s good for string literal version checks and bad for comparing to a variable that hasn’t been sanity-checked.

Mnemonic: Is this version of perl in the right bracket?

$SYSTEM_FD_MAX
$^F

The maximum system file descriptor, ordinarily 2. System file descriptors are passed to `exec()`ed processes, while higher file descriptors are not. Also, during an `open()`, system file descriptors are preserved even if the `open()` fails (ordinary file descriptors are closed before the `open()` is attempted). The close-on-exec status of a file descriptor will be decided according to the value of `$^F` when the corresponding file, pipe, or socket was opened, not the time of the `exec()`.

@F

The array `@F` contains the fields of each line read in when autosplit mode is turned on. See Section 69.1 [perlrun NAME], page 1176 for the **-a** switch. This array is package-specific, and must be declared or given a full package name if not in package main when running under `strict 'vars'`.

@INC

The array `@INC` contains the list of places that the `do EXPR`, `require`, or `use` constructs look for their library files. It initially consists of the arguments to any **-I** command-line switches, followed by the default Perl library, probably `/usr/local/lib/perl`, followed by `"."`, to represent the current directory. (`"."` will not be appended if taint checks are enabled, either by `-T` or by `-t`.) If you need to modify this at runtime, you should use the `use lib` pragma to get the machine-dependent library properly loaded also:

```
    use lib '/mypath/libdir/';
    use SomeMod;
```

You can also insert hooks into the file inclusion system by putting Perl code directly into `@INC`. Those hooks may be subroutine references, array references or blessed objects. See [perlfunc require], page 437 for details.

%INC

The hash `%INC` contains entries for each filename included via the `do`, `require`, or `use` operators. The key is the filename you specified (with module names converted to pathnames), and the value is the location of the file found. The `require` operator uses this hash to determine whether a particular file has already been included.

If the file was loaded via a hook (e.g. a subroutine reference, see [perlfunc require], page 437 for a description of these hooks), this hook is by default inserted into `%INC` in place of a filename. Note, however, that the hook may have set the `%INC` entry by itself to provide some more specific info.

$INPLACE_EDIT
$^I

The current value of the inplace-edit extension. Use `undef` to disable inplace editing.

Mnemonic: value of **-i** switch.

$^M

By default, running out of memory is an untrappable, fatal error. However, if suitably built, Perl can use the contents of `$^M` as an emergency memory pool after `die()`ing. Suppose that your Perl were compiled with `-DPERL_EMERGENCY_SBRK` and used Perl's malloc. Then

```
    $^M = 'a' x (1 << 16);
```

would allocate a 64K buffer for use in an emergency. See the `INSTALL` file in the Perl distribution for information on how to add custom C compilation flags when compiling perl. To discourage casual use of this advanced feature, there is no `English` long name for this variable.

This variable was added in Perl 5.004.

$OSNAME
$^O

The name of the operating system under which this copy of Perl was built, as determined during the configuration process. For examples see Section 56.5 [perlport PLATFORMS], page 963.

The value is identical to `$Config{'osname'}`. See also `Config` and the **-V** command-line switch documented in Section 69.1 [perlrun NAME], page 1176.

In Windows platforms, `$^O` is not very helpful: since it is always `MSWin32`, it doesn't tell the difference between 95/98/ME/NT/2000/XP/CE/.NET. Use `Win32::GetOSName()` or Win32::GetOSVersion() (see `Win32` and Section 56.1 [perlport NAME], page 951) to distinguish between the variants.

This variable was added in Perl 5.003.

%SIG

The hash `%SIG` contains signal handlers for signals. For example:

```
sub handler {   # 1st argument is signal name
    my($sig) = @_;
    print "Caught a SIG$sig--shutting down\n";
    close(LOG);
    exit(0);
    }

$SIG{'INT'}  = \&handler;
$SIG{'QUIT'} = \&handler;
...
$SIG{'INT'}  = 'DEFAULT';   # restore default action
$SIG{'QUIT'} = 'IGNORE';    # ignore SIGQUIT
```

Using a value of `'IGNORE'` usually has the effect of ignoring the signal, except for the `CHLD` signal. See Section 36.1 [perlipc NAME], page 667 for more about this special case.

Here are some other examples:

```
$SIG{"PIPE"} = "Plumber";   # assumes main::Plumber (not
                            # recommended)
$SIG{"PIPE"} = \&Plumber;   # just fine; assume current
                            # Plumber
$SIG{"PIPE"} = *Plumber;    # somewhat esoteric
$SIG{"PIPE"} = Plumber();   # oops, what did Plumber()
                            # return??
```

Be sure not to use a bareword as the name of a signal handler, lest you inadvertently call it.

If your system has the `sigaction()` function then signal handlers are installed using it. This means you get reliable signal handling.

The default delivery policy of signals changed in Perl v5.8.0 from immediate (also known as "unsafe") to deferred, also known as "safe signals". See Section 36.1 [perlipc NAME], page 667 for more information.

Certain internal hooks can be also set using the %SIG hash. The routine indicated by $SIG{__WARN__} is called when a warning message is about to be printed. The warning message is passed as the first argument. The presence of a __WARN__ hook causes the ordinary printing of warnings to STDERR to be suppressed. You can use this to save warnings in a variable, or turn warnings into fatal errors, like this:

```
    local $SIG{__WARN__} = sub { die $_[0] };
    eval $proggie;
```

As the 'IGNORE' hook is not supported by __WARN__, you can disable warnings using the empty subroutine:

```
    local $SIG{__WARN__} = sub {};
```

The routine indicated by $SIG{__DIE__} is called when a fatal exception is about to be thrown. The error message is passed as the first argument. When a __DIE__ hook routine returns, the exception processing continues as it would have in the absence of the hook, unless the hook routine itself exits via a goto &sub, a loop exit, or a die(). The __DIE__ handler is explicitly disabled during the call, so that you can die from a __DIE__ handler. Similarly for __WARN__.

Due to an implementation glitch, the $SIG{__DIE__} hook is called even inside an eval(). Do not use this to rewrite a pending exception in $@, or as a bizarre substitute for overriding CORE::GLOBAL::die(). This strange action at a distance may be fixed in a future release so that $SIG{__DIE__} is only called if your program is about to exit, as was the original intent. Any other use is deprecated.

__DIE__/__WARN__ handlers are very special in one respect: they may be called to report (probable) errors found by the parser. In such a case the parser may be in inconsistent state, so any attempt to evaluate Perl code from such a handler will probably result in a segfault. This means that warnings or errors that result from parsing Perl should be used with extreme caution, like this:

```
    require Carp if defined $^S;
    Carp::confess("Something wrong") if defined &Carp::confess;
    die "Something wrong, but could not load Carp to give "
      . "backtrace...\n\t"
      . "To see backtrace try starting Perl with -MCarp switch";
```

Here the first line will load Carp *unless* it is the parser who called the handler. The second line will print backtrace and die if Carp was available. The third line will be executed only if Carp was not available.

Having to even think about the $^S variable in your exception handlers is simply wrong. $SIG{__DIE__} as currently implemented invites grievous and difficult to track down errors. Avoid it and use an END{} or CORE::GLOBAL::die override instead.

See ⟨undefined⟩ [perlfunc die], page ⟨undefined⟩, ⟨undefined⟩ [perlfunc warn], page ⟨undefined⟩, [perlfunc eval], page 377, and `warnings` for additional information.

$BASETIME
$^T

The time at which the program began running, in seconds since the epoch (beginning of 1970). The values returned by the **-M**, **-A**, and **-C** filetests are based on this value.

$PERL_VERSION
$^V

The revision, version, and subversion of the Perl interpreter, represented as a `version` object.

This variable first appeared in perl v5.6.0; earlier versions of perl will see an undefined value. Before perl v5.10.0 `$^V` was represented as a v-string rather than a `version` object.

`$^V` can be used to determine whether the Perl interpreter executing a script is in the right range of versions. For example:

```
warn "Hashes not randomized!\n" if !$^V or $^V lt v5.8.1
```

While version objects overload stringification, to portably convert `$^V` into its string representation, use `sprintf()`'s `"%vd"` conversion, which works for both v-strings or version objects:

```
printf "version is v%vd\n", $^V;  # Perl's version
```

See the documentation of `use VERSION` and `require VERSION` for a convenient way to fail if the running Perl interpreter is too old.

See also `$]` for a decimal representation of the Perl version.

The main advantage of `$^V` over `$]` is that, for Perl v5.10.0 or later, it overloads operators, allowing easy comparison against other version representations (e.g. decimal, literal v-string, `"v1.2.3"`, or objects). The disadvantage is that prior to v5.10.0, it was only a literal v-string, which can't be easily printed or compared.

Mnemonic: use ^V for a version object.

${^WIN32_SLOPPY_STAT}

If this variable is set to a true value, then `stat()` on Windows will not try to open the file. This means that the link count cannot be determined and file attributes may be out of date if additional hardlinks to the file exist. On the other hand, not opening the file is considerably faster, especially for files on network drives.

This variable could be set in the `sitecustomize.pl` file to configure the local Perl installation to use "sloppy" `stat()` by default. See the documentation for **-f** in Section 69.3.3 [perlrun], page 1179 for more information about site customization.

This variable was added in Perl v5.10.0.

$EXECUTABLE_NAME
$^X

The name used to execute the current copy of Perl, from C's argv[0] or (where supported) /proc/self/exe.

Depending on the host operating system, the value of $^X may be a relative or absolute pathname of the perl program file, or may be the string used to invoke perl but not the pathname of the perl program file. Also, most operating systems permit invoking programs that are not in the PATH environment variable, so there is no guarantee that the value of $^X is in PATH. For VMS, the value may or may not include a version number.

You usually can use the value of $^X to re-invoke an independent copy of the same perl that is currently running, e.g.,

```
@first_run = `$^X -le "print int rand 100 for 1..100"`;
```

But recall that not all operating systems support forking or capturing of the output of commands, so this complex statement may not be portable.

It is not safe to use the value of $^X as a path name of a file, as some operating systems that have a mandatory suffix on executable files do not require use of the suffix when invoking a command. To convert the value of $^X to a path name, use the following statements:

```
# Build up a set of file names (not command names).
use Config;
my $this_perl = $^X;
if ($^O ne 'VMS') {
    $this_perl .= $Config{_exe}
      unless $this_perl =~ m/$Config{_exe}$/i;
    }
```

Because many operating systems permit anyone with read access to the Perl program file to make a copy of it, patch the copy, and then execute the copy, the security-conscious Perl programmer should take care to invoke the installed copy of perl, not the copy referenced by $^X. The following statements accomplish this goal, and produce a pathname that can be invoked as a command or referenced as a file.

```
use Config;
my $secure_perl_path = $Config{perlpath};
if ($^O ne 'VMS') {
    $secure_perl_path .= $Config{_exe}
        unless $secure_perl_path =~ m/$Config{_exe}$/i;
    }
```

86.3.2 Variables related to regular expressions

Most of the special variables related to regular expressions are side effects. Perl sets these variables when it has a successful match, so you should check the match result before using them. For instance:

```
if( /P(A)TT(ER)N/ ) {
    print "I found $1 and $2\n";
    }
```

These variables are read-only and dynamically-scoped, unless we note otherwise.

The dynamic nature of the regular expression variables means that their value is limited to the block that they are in, as demonstrated by this bit of code:

```
my $outer = ’Wallace and Grommit’;
my $inner = ’Mutt and Jeff’;

my $pattern = qr/(\S+) and (\S+)/;

sub show_n { print "\$1 is $1; \$2 is $2\n" }

{
OUTER:
    show_n() if $outer =~ m/$pattern/;

    INNER: {
        show_n() if $inner =~ m/$pattern/;
        }

    show_n();
}
```

The output shows that while in the `OUTER` block, the values of `$1` and `$2` are from the match against `$outer`. Inside the `INNER` block, the values of `$1` and `$2` are from the match against `$inner`, but only until the end of the block (i.e. the dynamic scope). After the `INNER` block completes, the values of `$1` and `$2` return to the values for the match against `$outer` even though we have not made another match:

```
$1 is Wallace; $2 is Grommit
$1 is Mutt; $2 is Jeff
$1 is Wallace; $2 is Grommit
```

86.3.2.1 Performance issues

Traditionally in Perl, any use of any of the three variables `$‘`, `$&` or `$’` (or their `use English` equivalents) anywhere in the code, caused all subsequent successful pattern matches to make a copy of the matched string, in case the code might subsequently access one of those variables. This imposed a considerable performance penalty across the whole program, so generally the use of these variables has been discouraged.

In Perl 5.6.0 the `@-` and `@+` dynamic arrays were introduced that supply the indices of successful matches. So you could for example do this:

```
$str =~ /pattern/;

print $‘, $&, $’; # bad: perfomance hit

print             # good: no perfomance hit
    substr($str, 0,     $-[0]),
    substr($str, $-[0], $+[0]-$-[0]),
    substr($str, $+[0]);
```

In Perl 5.10.0 the `/p` match operator flag and the `${^PREMATCH}`, `${^MATCH}`, and `${^POSTMATCH}` variables were introduced, that allowed you to suffer the penalties only on patterns marked with `/p`.

In Perl 5.18.0 onwards, perl started noting the presence of each of the three variables separately, and only copied that part of the string required; so in

```
$‘; $&; "abcdefgh" =˜ /d/
```

perl would only copy the "abcd" part of the string. That could make a big difference in something like

```
$str = ’x’ x 1_000_000;
$&; # whoops
$str =˜ /x/g # one char copied a million times, not a million chars
```

In Perl 5.20.0 a new copy-on-write system was enabled by default, which finally fixes all performance issues with these three variables, and makes them safe to use anywhere.

The `Devel::NYTProf` and `Devel::FindAmpersand` modules can help you find uses of these problematic match variables in your code.

$<*digits*> ($1, $2, ...)

Contains the subpattern from the corresponding set of capturing parentheses from the last successful pattern match, not counting patterns matched in nested blocks that have been exited already.

These variables are read-only and dynamically-scoped.

Mnemonic: like \digits.

$MATCH

$&

The string matched by the last successful pattern match (not counting any matches hidden within a BLOCK or `eval()` enclosed by the current BLOCK).

See Section 86.3.2.1 [Performance issues], page 1387 above for the serious performance implications of using this variable (even once) in your code.

This variable is read-only and dynamically-scoped.

Mnemonic: like `&` in some editors.

${^MATCH}

This is similar to `$&` (`$MATCH`) except that it does not incur the performance penalty associated with that variable.

See Section 86.3.2.1 [Performance issues], page 1387 above.

In Perl v5.18 and earlier, it is only guaranteed to return a defined value when the pattern was compiled or executed with the `/p` modifier. In Perl v5.20, the `/p` modifier does nothing, so `${^MATCH}` does the same thing as `$MATCH`.

This variable was added in Perl v5.10.0.

This variable is read-only and dynamically-scoped.

$PREMATCH
$‘

The string preceding whatever was matched by the last successful pattern match, not counting any matches hidden within a BLOCK or `eval` enclosed by the current BLOCK.

See Section 86.3.2.1 [Performance issues], page 1387 above for the serious performance implications of using this variable (even once) in your code.

This variable is read-only and dynamically-scoped.

Mnemonic: `‘` often precedes a quoted string.

${^PREMATCH}

This is similar to `$‘` ($PREMATCH) except that it does not incur the performance penalty associated with that variable.

See Section 86.3.2.1 [Performance issues], page 1387 above.

In Perl v5.18 and earlier, it is only guaranteed to return a defined value when the pattern was compiled or executed with the `/p` modifier. In Perl v5.20, the `/p` modifier does nothing, so `${^PREMATCH}` does the same thing as `$PREMATCH`.

This variable was added in Perl v5.10.0.

This variable is read-only and dynamically-scoped.

$POSTMATCH
$’

The string following whatever was matched by the last successful pattern match (not counting any matches hidden within a BLOCK or `eval()` enclosed by the current BLOCK). Example:

```
local $_ = ’abcdefghi’;
/def/;
print "$‘:$&:$’\n";          # prints abc:def:ghi
```

See Section 86.3.2.1 [Performance issues], page 1387 above for the serious performance implications of using this variable (even once) in your code.

This variable is read-only and dynamically-scoped.

Mnemonic: `’` often follows a quoted string.

${^POSTMATCH}

This is similar to `$’` (`$POSTMATCH`) except that it does not incur the performance penalty associated with that variable.

See Section 86.3.2.1 [Performance issues], page 1387 above.

In Perl v5.18 and earlier, it is only guaranteed to return a defined value when the pattern was compiled or executed with the `/p` modifier. In Perl v5.20, the `/p` modifier does nothing, so `${^POSTMATCH}` does the same thing as `$POSTMATCH`.

This variable was added in Perl v5.10.0.

This variable is read-only and dynamically-scoped.

$LAST_PAREN_MATCH
$+

The text matched by the last bracket of the last successful search pattern. This is useful if you don’t know which one of a set of alternative patterns matched. For example:

```
    /Version: (.*)|Revision: (.*)/ && ($rev = $+);
```

This variable is read-only and dynamically-scoped.

Mnemonic: be positive and forward looking.

$LAST_SUBMATCH_RESULT
$^N

The text matched by the used group most-recently closed (i.e. the group with the rightmost closing parenthesis) of the last successful search pattern.

This is primarily used inside `(?{...})` blocks for examining text recently matched. For example, to effectively capture text to a variable (in addition to `$1`, `$2`, etc.), replace `(...)` with

```
    (?:(...)(?{ $var = $^N }))
```

By setting and then using `$var` in this way relieves you from having to worry about exactly which numbered set of parentheses they are.

This variable was added in Perl v5.8.0.

Mnemonic: the (possibly) Nested parenthesis that most recently closed.

@LAST_MATCH_END
@+

This array holds the offsets of the ends of the last successful submatches in the currently active dynamic scope. `$+[0]` is the offset into the string of the end of the entire match. This is the same value as what the `pos` function returns when called on the variable that was matched against. The *n*th element of this array holds the offset of the *n*th submatch, so `$+[1]` is the offset past where `$1` ends, `$+[2]` the offset past where `$2` ends, and so on. You can use `$#+` to determine how many subgroups were in the last successful match. See the examples given for the `@-` variable.

This variable was added in Perl v5.6.0.

%LAST_PAREN_MATCH
%+

Similar to `@+`, the `%+` hash allows access to the named capture buffers, should they exist, in the last successful match in the currently active dynamic scope.

For example, `$+{foo}` is equivalent to `$1` after the following match:

```
    'foo' =~ /(?<foo>foo)/;
```

The keys of the `%+` hash list only the names of buffers that have captured (and that are thus associated to defined values).

The underlying behaviour of `%+` is provided by the `Tie-Hash-NamedCapture` module.

Note: `%-` and `%+` are tied views into a common internal hash associated with the last successful regular expression. Therefore mixing iterative access to them via `each` may have unpredictable results. Likewise, if the last successful match changes, then the results may be surprising.

This variable was added in Perl v5.10.0.

This variable is read-only and dynamically-scoped.

@LAST_MATCH_START
@-

`$-[0]` is the offset of the start of the last successful match. `$-[`*n*`]` is the offset of the start of the substring matched by *n*-th subpattern, or undef if the subpattern did not match.

Thus, after a match against `$_`, `$&` coincides with `substr $_, $-[0], $+[0] - $-[0]`. Similarly, $*n* coincides with `substr $_, $-[n], $+[n] - $-[n]` if `$-[n]` is defined, and `$+` coincides with `substr $_, $-[$#-], $+[$#-] - $-[$#-]`. One can use `$#-` to find the last matched subgroup in the last successful match. Contrast with `$#+`, the number of subgroups in the regular expression. Compare with `@+`.

This array holds the offsets of the beginnings of the last successful submatches in the currently active dynamic scope. `$-[0]` is the offset into the string of the beginning of the entire match. The *n*th element of this array holds the offset of the *n*th submatch, so `$-[1]` is the offset where `$1` begins, `$-[2]` the offset where `$2` begins, and so on.

After a match against some variable `$var`:

`` $` `` is the same as `substr($var, 0, $-[0])`
`$&` is the same as `substr($var, $-[0], $+[0] - $-[0])`
`$'` is the same as `substr($var, $+[0])`
`$1` is the same as `substr($var, $-[1], $+[1] - $-[1])`
`$2` is the same as `substr($var, $-[2], $+[2] - $-[2])`
`$3` is the same as `substr($var, $-[3], $+[3] - $-[3])`

This variable was added in Perl v5.6.0.

%LAST_MATCH_START
%-

Similar to `%+`, this variable allows access to the named capture groups in the last successful match in the currently active dynamic scope. To each capture group name found in the regular expression, it associates a reference to an array containing the list of values captured by all buffers with that name (should there be several of them), in the order where they appear.

Here's an example:

```
if ('1234' =~ /(?<A>1)(?<B>2)(?<A>3)(?<B>4)/) {
    foreach my $bufname (sort keys %-) {
        my $ary = $-{$bufname};
        foreach my $idx (0..$#$ary) {
            print "\$-{$bufname}[$idx] : ",
                  (defined($ary->[$idx])
                      ? "'$ary->[$idx]'"
                      : "undef"),
                  "\n";
        }
    }
}
```

would print out:

```
$-{A}[0] : '1'
$-{A}[1] : '3'
$-{B}[0] : '2'
$-{B}[1] : '4'
```

The keys of the `%-` hash correspond to all buffer names found in the regular expression.

The behaviour of `%-` is implemented via the `Tie-Hash-NamedCapture` module.

Note: `%-` and `%+` are tied views into a common internal hash associated with the last successful regular expression. Therefore mixing iterative access to them via `each` may have unpredictable results. Likewise, if the last successful match changes, then the results may be surprising.

This variable was added in Perl v5.10.0.

This variable is read-only and dynamically-scoped.

$LAST_REGEXP_CODE_RESULT
$^R

The result of evaluation of the last successful `(?{ code })` regular expression assertion (see Section 58.1 [perlre NAME], page 989). May be written to.

This variable was added in Perl 5.005.

${^RE_DEBUG_FLAGS}

The current value of the regex debugging flags. Set to 0 for no debug output even when the `re 'debug'` module is loaded. See `re` for details.

This variable was added in Perl v5.10.0.

${^RE_TRIE_MAXBUF}

Controls how certain regex optimisations are applied and how much memory they utilize. This value by default is 65536 which corresponds to a 512kB temporary cache. Set this to a higher value to trade memory for speed when matching large alternations. Set it to a lower value if you want the optimisations to be as conservative of memory as possible but still occur, and set it to a negative value to prevent the optimisation and conserve the most memory. Under normal situations this variable should be of no interest to you.

This variable was added in Perl v5.10.0.

86.3.3 Variables related to filehandles

Variables that depend on the currently selected filehandle may be set by calling an appropriate object method on the `IO::Handle` object, although this is less efficient than using the regular built-in variables. (Summary lines below for this contain the word HANDLE.) First you must say

```
use IO::Handle;
```

after which you may use either

```
method HANDLE EXPR
```

or more safely,

```
HANDLE->method(EXPR)
```

Each method returns the old value of the `IO::Handle` attribute. The methods each take an optional EXPR, which, if supplied, specifies the new value for the `IO::Handle` attribute in question. If not supplied, most methods do nothing to the current value–except for `autoflush()`, which will assume a 1 for you, just to be different.

Because loading in the `IO::Handle` class is an expensive operation, you should learn how to use the regular built-in variables.

A few of these variables are considered "read-only". This means that if you try to assign to this variable, either directly or indirectly through a reference, you'll raise a run-time exception.

You should be very careful when modifying the default values of most special variables described in this document. In most cases you want to localize these variables before changing them, since if you don't, the change may affect other modules which rely on the default values of the special variables that you have changed. This is one of the correct ways to read the whole file at once:

```
open my $fh, "<", "foo" or die $!;
local $/; # enable localized slurp mode
my $content = <$fh>;
close $fh;
```

But the following code is quite bad:

```
open my $fh, "<", "foo" or die $!;
undef $/; # enable slurp mode
my $content = <$fh>;
close $fh;
```

since some other module, may want to read data from some file in the default "line mode", so if the code we have just presented has been executed, the global value of `$/` is now changed for any other code running inside the same Perl interpreter.

Usually when a variable is localized you want to make sure that this change affects the shortest scope possible. So unless you are already inside some short `{}` block, you should create one yourself. For example:

```
my $content = '';
open my $fh, "<", "foo" or die $!;
{
    local $/;
    $content = <$fh>;
}
close $fh;
```

Here is an example of how your own code can go broken:

```
for ( 1..3 ){
    $\ = "\r\n";
    nasty_break();
    print "$_";
}
```

```
sub nasty_break {
    $\ = "\f";
    # do something with $_
}
```

You probably expect this code to print the equivalent of

```
"1\r\n2\r\n3\r\n"
```

but instead you get:

```
"1\f2\f3\f"
```

Why? Because `nasty_break()` modifies `$\` without localizing it first. The value you set in `nasty_break()` is still there when you return. The fix is to add `local()` so the value doesn't leak out of `nasty_break()`:

```
local $\ = "\f";
```

It's easy to notice the problem in such a short example, but in more complicated code you are looking for trouble if you don't localize changes to the special variables.

$ARGV

Contains the name of the current file when reading from `<>`.

@ARGV

The array `@ARGV` contains the command-line arguments intended for the script. `$#ARGV` is generally the number of arguments minus one, because `$ARGV[0]` is the first argument, *not* the program's command name itself. See [$0], page 1378 for the command name.

ARGV

The special filehandle that iterates over command-line filenames in `@ARGV`. Usually written as the null filehandle in the angle operator `<>`. Note that currently `ARGV` only has its magical effect within the `<>` operator; elsewhere it is just a plain filehandle corresponding to the last file opened by `<>`. In particular, passing `\*ARGV` as a parameter to a function that expects a filehandle may not cause your function to automatically read the contents of all the files in `@ARGV`.

ARGVOUT

The special filehandle that points to the currently open output file when doing edit-in-place processing with **-i**. Useful when you have to do a lot of inserting and don't want to keep modifying `$_`. See Section 69.1 [perlrun NAME], page 1176 for the **-i** switch.

IO::Handle->output_field_separator(EXPR)
$OUTPUT_FIELD_SEPARATOR
$OFS

$,

The output field separator for the print operator. If defined, this value is printed between each of print's arguments. Default is `undef`.

You cannot call `output_field_separator()` on a handle, only as a static method. See `IO-Handle`.

Mnemonic: what is printed when there is a "," in your print statement.

HANDLE->input_line_number(EXPR)
$INPUT_LINE_NUMBER
$NR

$.

Current line number for the last filehandle accessed.

Each filehandle in Perl counts the number of lines that have been read from it. (Depending on the value of `$/`, Perl's idea of what constitutes a line may not match yours.) When a line is read from a filehandle (via `readline()` or `<>`), or when `tell()` or `seek()` is called on it, `$.` becomes an alias to the line counter for that filehandle.

You can adjust the counter by assigning to `$.`, but this will not actually move the seek pointer. *Localizing `$.` will not localize the filehandle's line count.* Instead, it will localize perl's notion of which filehandle `$.` is currently aliased to.

`$.` is reset when the filehandle is closed, but **not** when an open filehandle is reopened without an intervening `close()`. For more details, see Section 48.2.33 [perlop I/O Operators], page 844. Because `<>` never does an explicit close, line numbers increase across `ARGV` files (but see examples in [perlfunc eof], page 376).

You can also use `HANDLE->input_line_number(EXPR)` to access the line counter for a given filehandle without having to worry about which handle you last accessed.

Mnemonic: many programs use "." to mean the current line number.

IO::Handle->input_record_separator(EXPR)
$INPUT_RECORD_SEPARATOR
$RS

$/

The input record separator, newline by default. This influences Perl's idea of what a "line" is. Works like **awk**'s RS variable, including treating empty lines as a terminator if set to the null string (an empty line cannot contain any spaces or tabs). You may set it to a multi-character string to match a multi-character terminator, or to `undef` to read through the end of file. Setting it to `"\n\n"` means something slightly different than setting to `""`, if the file contains consecutive empty lines. Setting to `""` will treat two or more consecutive empty lines as a single empty line. Setting to `"\n\n"` will blindly assume that the next input character belongs to the next paragraph, even if it's a newline.

```
    local $/;           # enable "slurp" mode
    local $_ = <FH>;    # whole file now here
    s/\n[ \t]+/ /g;
```

Remember: the value of `$/` is a string, not a regex. **awk** has to be better for something. :-)

Setting `$/` to a reference to an integer, scalar containing an integer, or scalar that's convertible to an integer will attempt to read records instead of lines, with the maximum record size being the referenced integer number of characters. So this:

```
    local $/ = \32768; # or \"32768", or \$var_containing_32768
    open my $fh, "<", $myfile or die $!;
    local $_ = <$fh>;
```

will read a record of no more than 32768 characters from $fh. If you're not reading from a record-oriented file (or your OS doesn't have record-oriented files), then you'll likely get a full chunk of data with every read. If a record is larger than the record size you've set, you'll get the record back in pieces. Trying to set the record size to zero or less is deprecated and will cause $/ to have the value of "undef", which will cause reading in the (rest of the) whole file.

As of 5.19.9 setting `$/` to any other form of reference will throw a fatal exception. This is in preparation for supporting new ways to set `$/` in the future.

On VMS only, record reads bypass PerlIO layers and any associated buffering, so you must not mix record and non-record reads on the same filehandle. Record mode mixes with line mode only when the same buffering layer is in use for both modes.

You cannot call `input_record_separator()` on a handle, only as a static method. See `IO-Handle`.

See also Section 56.3.1 [perlport Newlines], page 952. Also see [$.], page 1395.

Mnemonic: / delimits line boundaries when quoting poetry.

IO::Handle->output_record_separator(EXPR)
$OUTPUT_RECORD_SEPARATOR
$ORS

$\

The output record separator for the print operator. If defined, this value is printed after the last of print's arguments. Default is `undef`.

You cannot call `output_record_separator()` on a handle, only as a static method. See `IO-Handle`.

Mnemonic: you set `$\` instead of adding "\n" at the end of the print. Also, it's just like `$/`, but it's what you get "back" from Perl.

HANDLE->autoflush(EXPR)
$OUTPUT_AUTOFLUSH
`$|`

If set to nonzero, forces a flush right away and after every write or print on the currently selected output channel. Default is 0 (regardless of whether the channel is really buffered by the system or not; `$|` tells you only whether you've asked Perl explicitly to flush after each write). STDOUT will typically be line buffered if output is to the terminal and block buffered otherwise. Setting this variable is useful primarily when you are outputting to a pipe or socket, such as when you are running a Perl program under **rsh** and want to see the output as it's happening. This has no effect on input buffering. See [perlfunc getc], page 388 for that. See [perlfunc select], page 443 on how to select the output channel. See also `IO-Handle`.

Mnemonic: when you want your pipes to be piping hot.

${^LAST_FH}

This read-only variable contains a reference to the last-read filehandle. This is set by `<HANDLE>`, `readline`, `tell`, `eof` and `seek`. This is the same handle that `$.` and `tell` and `eof` without arguments use. It is also the handle used when Perl appends ", <STDIN> line 1" to an error or warning message.

This variable was added in Perl v5.18.0.

86.3.3.1 Variables related to formats

The special variables for formats are a subset of those for filehandles. See Section 24.1 [perlform NAME], page 343 for more information about Perl's formats.

$ACCUMULATOR
$^A

The current value of the `write()` accumulator for `format()` lines. A format contains `formline()` calls that put their result into `$^A`. After calling its format, `write()` prints out the contents of `$^A` and empties. So you never really see the contents of `$^A` unless you call `formline()` yourself and then look at it. See Section 24.1 [perlform NAME], page 343 and [perlfunc formline PICTURE,LIST], page 387.

IO::Handle->format_formfeed(EXPR)
$FORMAT_FORMFEED
$^L

What formats output as a form feed. The default is `\f`.

You cannot call `format_formfeed()` on a handle, only as a static method. See `IO-Handle`.

HANDLE->format_page_number(EXPR)
$FORMAT_PAGE_NUMBER
$%

The current page number of the currently selected output channel.

Mnemonic: `%` is page number in **nroff**.

HANDLE->format_lines_left(EXPR)
$FORMAT_LINES_LEFT
$-

The number of lines left on the page of the currently selected output channel.

Mnemonic: lines_on_page - lines_printed.

IO::Handle->format_line_break_characters EXPR
$FORMAT_LINE_BREAK_CHARACTERS
$:

The current set of characters after which a string may be broken to fill continuation fields (starting with `^`) in a format. The default is " \n-", to break on a space, newline, or a hyphen.

You cannot call `format_line_break_characters()` on a handle, only as a static method. See `IO-Handle`.

Mnemonic: a "colon" in poetry is a part of a line.

HANDLE->format_lines_per_page(EXPR)
$FORMAT_LINES_PER_PAGE
$=

The current page length (printable lines) of the currently selected output channel. The default is 60.

Mnemonic: = has horizontal lines.

HANDLE->format_top_name(EXPR)
$FORMAT_TOP_NAME
$^

The name of the current top-of-page format for the currently selected output channel. The default is the name of the filehandle with _TOP appended. For example, the default format top name for the STDOUT filehandle is STDOUT_TOP.

Mnemonic: points to top of page.

HANDLE->format_name(EXPR)
$FORMAT_NAME
$~

The name of the current report format for the currently selected output channel. The default format name is the same as the filehandle name. For example, the default format name for the STDOUT filehandle is just STDOUT.

Mnemonic: brother to $^.

86.3.4 Error Variables

The variables $@, $!, $^E, and $? contain information about different types of error conditions that may appear during execution of a Perl program. The variables are shown ordered by the "distance" between the subsystem which reported the error and the Perl process. They correspond to errors detected by the Perl interpreter, C library, operating system, or an external program, respectively.

To illustrate the differences between these variables, consider the following Perl expression, which uses a single-quoted string. After execution of this statement, perl may have set all four special error variables:

```
eval q{
    open my $pipe, "/cdrom/install |" or die $!;
    my @res = <$pipe>;
    close $pipe or die "bad pipe: $?, $!";
};
```

When perl executes the eval() expression, it translates the open(), <PIPE>, and close calls in the C run-time library and thence to the operating system kernel. perl sets $! to the C library's errno if one of these calls fails.

$@ is set if the string to be eval-ed did not compile (this may happen if open or close were imported with bad prototypes), or if Perl code executed during evaluation die()d. In these cases the value of $@ is the compile error, or the argument to die (which will interpolate $! and $?). (See also Fatal, though.)

Under a few operating systems, `$^E` may contain a more verbose error indicator, such as in this case, "CDROM tray not closed." Systems that do not support extended error messages leave `$^E` the same as `$!`.

Finally, `$?` may be set to non-0 value if the external program `/cdrom/install` fails. The upper eight bits reflect specific error conditions encountered by the program (the program's `exit()` value). The lower eight bits reflect mode of failure, like signal death and core dump information. See wait(2) for details. In contrast to `$!` and `$^E`, which are set only if error condition is detected, the variable `$?` is set on each `wait` or pipe `close`, overwriting the old value. This is more like `$@`, which on every `eval()` is always set on failure and cleared on success.

For more details, see the individual descriptions at `$@`, `$!`, `$^E`, and `$?`.

${^CHILD_ERROR_NATIVE}

The native status returned by the last pipe close, backtick (``) command, successful call to `wait()` or `waitpid()`, or from the `system()` operator. On POSIX-like systems this value can be decoded with the WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG and WIFCONTINUED functions provided by the `POSIX` module.

Under VMS this reflects the actual VMS exit status; i.e. it is the same as `$?` when the pragma `use vmsish 'status'` is in effect.

This variable was added in Perl v5.10.0.

$EXTENDED_OS_ERROR
$^E

Error information specific to the current operating system. At the moment, this differs from `$!` under only VMS, OS/2, and Win32 (and for MacPerl). On all other platforms, `$^E` is always just the same as `$!`.

Under VMS, `$^E` provides the VMS status value from the last system error. This is more specific information about the last system error than that provided by `$!`. This is particularly important when `$!` is set to **EVMSERR**.

Under OS/2, `$^E` is set to the error code of the last call to OS/2 API either via CRT, or directly from perl.

Under Win32, `$^E` always returns the last error information reported by the Win32 call `GetLastError()` which describes the last error from within the Win32 API. Most Win32-specific code will report errors via `$^E`. ANSI C and Unix-like calls set `errno` and so most portable Perl code will report errors via `$!`.

Caveats mentioned in the description of `$!` generally apply to `$^E`, also.

This variable was added in Perl 5.003.

Mnemonic: Extra error explanation.

$EXCEPTIONS_BEING_CAUGHT
$^S

Current state of the interpreter.

```
$^S         State
---------   -------------------------------------
```

```
undef       Parsing module, eval, or main program
true (1)    Executing an eval
false (0)   Otherwise
```

The first state may happen in `$SIG{__DIE__}` and `$SIG{__WARN__}` handlers.

The English name $EXCEPTIONS_BEING_CAUGHT is slightly misleading, because the `undef` value does not indicate whether exceptions are being caught, since compilation of the main program does not catch exceptions.

This variable was added in Perl 5.004.

$WARNING
$^W

The current value of the warning switch, initially true if **-w** was used, false otherwise, but directly modifiable.

See also `warnings`.

Mnemonic: related to the **-w** switch.

${^WARNING_BITS}

The current set of warning checks enabled by the `use warnings` pragma. It has the same scoping as the `$^H` and `%^H` variables. The exact values are considered internal to the `warnings` pragma and may change between versions of Perl.

This variable was added in Perl v5.6.0.

$OS_ERROR
$ERRNO

$!

When referenced, `$!` retrieves the current value of the C `errno` integer variable. If `$!` is assigned a numerical value, that value is stored in `errno`. When referenced as a string, `$!` yields the system error string corresponding to `errno`.

Many system or library calls set `errno` if they fail, to indicate the cause of failure. They usually do **not** set `errno` to zero if they succeed. This means `errno`, hence `$!`, is meaningful only *immediately* after a **failure**:

```
if (open my $fh, "<", $filename) {
            # Here $! is meaningless.
            ...
}
else {
            # ONLY here is $! meaningful.
            ...
            # Already here $! might be meaningless.
}
# Since here we might have either success or failure,
# $! is meaningless.
```

Here, *meaningless* means that `$!` may be unrelated to the outcome of the `open()` operator. Assignment to `$!` is similarly ephemeral. It can be used immediately before invoking the `die()` operator, to set the exit value, or to inspect the system error string corresponding to error *n*, or to restore `$!` to a meaningful state.

Mnemonic: What just went bang?

%OS_ERROR
%ERRNO

%!

Each element of `%!` has a true value only if `$!` is set to that value. For example, `$!{ENOENT}` is true if and only if the current value of `$!` is `ENOENT`; that is, if the most recent error was "No such file or directory" (or its moral equivalent: not all operating systems give that exact error, and certainly not all languages). To check if a particular key is meaningful on your system, use `exists $!{the_key}`; for a list of legal keys, use `keys %!`. See `Errno` for more information, and also see [$!], page 1400.

This variable was added in Perl 5.005.

$CHILD_ERROR
$?

The status returned by the last pipe close, backtick (``) command, successful call to `wait()` or `waitpid()`, or from the `system()` operator. This is just the 16-bit status word returned by the traditional Unix `wait()` system call (or else is made up to look like it). Thus, the exit value of the subprocess is really (`$? >> 8`), and `$? & 127` gives which signal, if any, the process died from, and `$? & 128` reports whether there was a core dump.

Additionally, if the `h_errno` variable is supported in C, its value is returned via `$?` if any `gethost*()` function fails.

If you have installed a signal handler for `SIGCHLD`, the value of `$?` will usually be wrong outside that handler.

Inside an `END` subroutine `$?` contains the value that is going to be given to `exit()`. You can modify `$?` in an `END` subroutine to change the exit status of your program. For example:

```
END {
    $? = 1 if $? == 255;  # die would make it 255
}
```

Under VMS, the pragma `use vmsish 'status'` makes `$?` reflect the actual VMS exit status, instead of the default emulation of POSIX status; see [perlvms $?], page 1424 for details.

Mnemonic: similar to **sh** and **ksh**.

$EVAL_ERROR
$@

The Perl syntax error message from the last `eval()` operator. If `$@` is the null string, the last `eval()` parsed and executed correctly (although the operations you invoked may have failed in the normal fashion).

Warning messages are not collected in this variable. You can, however, set up a routine to process warnings by setting `$SIG{__WARN__}` as described in [%SIG], page 1383.

Mnemonic: Where was the syntax error `"at"`?

86.3.5 Variables related to the interpreter state

These variables provide information about the current interpreter state.

$COMPILING
$^C

The current value of the flag associated with the **-c** switch. Mainly of use with **-MO=...** to allow code to alter its behavior when being compiled, such as for example to `AUTOLOAD` at compile time rather than normal, deferred loading. Setting `$^C = 1` is similar to calling `B::minus_c`.

This variable was added in Perl v5.6.0.

$DEBUGGING
$^D

The current value of the debugging flags. May be read or set. Like its command-line equivalent, you can use numeric or symbolic values, eg `$^D = 10` or `$^D = "st"`.

Mnemonic: value of **-D** switch.

${^ENCODING}

DEPRECATED!!!

The *object reference* to the `Encode` object that is used to convert the source code to Unicode. Thanks to this variable your Perl script does not have to be written in UTF-8. Default is `undef`.

Setting this variable to any other value than `undef` is deprecated due to fundamental defects in its design and implementation. It is planned to remove it from a future Perl version. Its purpose was to allow your non-ASCII Perl scripts to not have to be written in UTF-8; this was useful before editors that worked on UTF-8 encoded text were common, but that was long ago. It causes problems, such as affecting the operation of other modules that aren't expecting it, causing general mayhem. Its use can lead to segfaults.

If you need something like this functionality, you should use the `encoding` pragma, which is also deprecated, but has fewer nasty side effects.

If you are coming here because code of yours is being adversely affected by someone's use of this variable, you can usually work around it by doing this:

```
 local ${^ENCODING};
```

near the beginning of the functions that are getting broken. This undefines the variable during the scope of execution of the including function.

This variable was added in Perl 5.8.2.

${^GLOBAL_PHASE}

The current phase of the perl interpreter.

Possible values are:

CONSTRUCT

The PerlInterpreter* is being constructed via perl_construct. This value is mostly there for completeness and for use via the underlying C variable PL_phase. It's not really possible for Perl code to be executed unless construction of the interpreter is finished.

START

This is the global compile-time. That includes, basically, every BEGIN block executed directly or indirectly from during the compile-time of the top-level program.

This phase is not called "BEGIN" to avoid confusion with BEGIN-blocks, as those are executed during compile-time of any compilation unit, not just the top-level program. A new, localised compile-time entered at run-time, for example by constructs as eval "use SomeModule" are not global interpreter phases, and therefore aren't reflected by ${^GLOBAL_PHASE}.

CHECK

Execution of any CHECK blocks.

INIT

Similar to "CHECK", but for INIT-blocks, not CHECK blocks.

RUN

The main run-time, i.e. the execution of PL_main_root.

END

Execution of any END blocks.

DESTRUCT

Global destruction.

Also note that there's no value for UNITCHECK-blocks. That's because those are run for each compilation unit individually, and therefore is not a global interpreter phase.

Not every program has to go through each of the possible phases, but transition from one phase to another can only happen in the order described in the above list.

An example of all of the phases Perl code can see:

```
    BEGIN { print "compile-time: ${^GLOBAL_PHASE}\n" }

    INIT  { print "init-time: ${^GLOBAL_PHASE}\n" }

    CHECK { print "check-time: ${^GLOBAL_PHASE}\n" }

    {
        package Print::Phase;

        sub new {
            my ($class, $time) = @_;
            return bless \$time, $class;
        }

        sub DESTROY {
            my $self = shift;
            print "$$self: ${^GLOBAL_PHASE}\n";
        }
    }

    print "run-time: ${^GLOBAL_PHASE}\n";

    my $runtime = Print::Phase->new(
        "lexical variables are garbage collected before END"
    );

    END   { print "end-time: ${^GLOBAL_PHASE}\n" }

    our $destruct = Print::Phase->new(
        "package variables are garbage collected after END"
    );
```

This will print out

```
compile-time: START
check-time: CHECK
init-time: INIT
run-time: RUN
lexical variables are garbage collected before END: RUN
end-time: END
package variables are garbage collected after END: DESTRUCT
```

This variable was added in Perl 5.14.0.

$^H

WARNING: This variable is strictly for internal use only. Its availability, behavior, and contents are subject to change without notice.

This variable contains compile-time hints for the Perl interpreter. At the end of compilation of a BLOCK the value of this variable is restored to the value when the interpreter started to compile the BLOCK.

When perl begins to parse any block construct that provides a lexical scope (e.g., eval body, required file, subroutine body, loop body, or conditional block), the existing value of `$^H` is saved, but its value is left unchanged. When the compilation of the block is completed, it regains the saved value. Between the points where its value is saved and restored, code that executes within BEGIN blocks is free to change the value of `$^H`.

This behavior provides the semantic of lexical scoping, and is used in, for instance, the `use strict` pragma.

The contents should be an integer; different bits of it are used for different pragmatic flags. Here's an example:

```
sub add_100 { $^H |= 0x100 }

sub foo {
    BEGIN { add_100() }
    bar->baz($boon);
}
```

Consider what happens during execution of the BEGIN block. At this point the BEGIN block has already been compiled, but the body of `foo()` is still being compiled. The new value of `$^H` will therefore be visible only while the body of `foo()` is being compiled.

Substitution of `BEGIN { add_100() }` block with:

```
BEGIN { require strict; strict->import('vars') }
```

demonstrates how `use strict 'vars'` is implemented. Here's a conditional version of the same lexical pragma:

```
BEGIN {
    require strict; strict->import('vars') if $condition
}
```

This variable was added in Perl 5.003.

%^H

The `%^H` hash provides the same scoping semantic as `$^H`. This makes it useful for implementation of lexically scoped pragmas. See Section 57.1 [perlpragma NAME], page 986. All the entries are stringified when accessed at runtime, so only simple values can be accommodated. This means no pointers to objects, for example.

When putting items into `%^H`, in order to avoid conflicting with other users of the hash there is a convention regarding which keys to use. A module should use only keys that begin with the module's name (the name of its main package) and a `"/"` character. For example, a module `Foo::Bar` should use keys such as `Foo::Bar/baz`.

This variable was added in Perl v5.6.0.

${^OPEN}

An internal variable used by PerlIO. A string in two parts, separated by a `\0` byte, the first part describes the input layers, the second part describes the output layers.

This variable was added in Perl v5.8.0.

$PERLDB
$^P

The internal variable for debugging support. The meanings of the various bits are subject to change, but currently indicate:

0x01

Debug subroutine enter/exit.

0x02

Line-by-line debugging. Causes `DB::DB()` subroutine to be called for each statement executed. Also causes saving source code lines (like 0x400).

0x04

Switch off optimizations.

0x08

Preserve more data for future interactive inspections.

0x10

Keep info about source lines on which a subroutine is defined.

0x20

Start with single-step on.

0x40

Use subroutine address instead of name when reporting.

0x80

Report `goto &subroutine` as well.

0x100

Provide informative "file" names for evals based on the place they were compiled.

0x200

Provide informative names to anonymous subroutines based on the place they were compiled.

0x400

Save source code lines into `@{"_<$filename"}`.

0x800

When saving source, include evals that generate no subroutines.

0x1000

When saving source, include source that did not compile.

Some bits may be relevant at compile-time only, some at run-time only. This is a new mechanism and the details may change. See also Section 13.1 [perldebguts NAME], page 90.

${^TAINT}

Reflects if taint mode is on or off. 1 for on (the program was run with **-T**), 0 for off, -1 when only taint warnings are enabled (i.e. with **-t** or **-TU**).

This variable is read-only.

This variable was added in Perl v5.8.0.

${^UNICODE}

Reflects certain Unicode settings of Perl. See Section 69.1 [perlrun NAME], page 1176 documentation for the `-C` switch for more information about the possible values.

This variable is set during Perl startup and is thereafter read-only.

This variable was added in Perl v5.8.2.

${^UTF8CACHE}

This variable controls the state of the internal UTF-8 offset caching code. 1 for on (the default), 0 for off, -1 to debug the caching code by checking all its results against linear scans, and panicking on any discrepancy.

This variable was added in Perl v5.8.9. It is subject to change or removal without notice, but is currently used to avoid recalculating the boundaries of multi-byte UTF-8-encoded characters.

${^UTF8LOCALE}

This variable indicates whether a UTF-8 locale was detected by perl at startup. This information is used by perl when it's in adjust-utf8ness-to-locale mode (as when run with the `-CL` command-line switch); see Section 69.1 [perlrun NAME], page 1176 for more info on this.

This variable was added in Perl v5.8.8.

86.3.6 Deprecated and removed variables

Deprecating a variable announces the intent of the perl maintainers to eventually remove the variable from the language. It may still be available despite its status. Using a deprecated variable triggers a warning.

Once a variable is removed, its use triggers an error telling you the variable is unsupported.

See Section 16.1 [perldiag NAME], page 137 for details about error messages.

$#

`$#` was a variable that could be used to format printed numbers. After a deprecation cycle, its magic was removed in Perl v5.10.0 and using it now triggers a warning: `$# is no longer supported`.

This is not the sigil you use in front of an array name to get the last index, like `$#array`. That's still how you get the last index of an array in Perl. The two have nothing to do with each other.

Deprecated in Perl 5.

Removed in Perl v5.10.0.

$*

`$*` was a variable that you could use to enable multiline matching. After a deprecation cycle, its magic was removed in Perl v5.10.0. Using it now triggers a warning: `$* is no longer supported`. You should use the `/s` and `/m` regexp modifiers instead.

Deprecated in Perl 5.

Removed in Perl v5.10.0.

$[

This variable stores the index of the first element in an array, and of the first character in a substring. The default is 0, but you could theoretically set it to 1 to make Perl behave more like **awk** (or Fortran) when subscripting and when evaluating the index() and substr() functions.

As of release 5 of Perl, assignment to `$[` is treated as a compiler directive, and cannot influence the behavior of any other file. (That's why you can only assign compile-time constants to it.) Its use is highly discouraged.

Prior to Perl v5.10.0, assignment to `$[` could be seen from outer lexical scopes in the same file, unlike other compile-time directives (such as `strict`). Using local() on it would bind its value strictly to a lexical block. Now it is always lexically scoped.

As of Perl v5.16.0, it is implemented by the `arybase` module. See `arybase` for more details on its behaviour.

Under `use v5.16`, or `no feature "array_base"`, `$[` no longer has any effect, and always contains 0. Assigning 0 to it is permitted, but any other value will produce an error.

Mnemonic: [begins subscripts.

Deprecated in Perl v5.12.0.

87 perlvms

87.1 NAME

perlvms - VMS-specific documentation for Perl

87.2 DESCRIPTION

Gathered below are notes describing details of Perl 5's behavior on VMS. They are a supplement to the regular Perl 5 documentation, so we have focussed on the ways in which Perl 5 functions differently under VMS than it does under Unix, and on the interactions between Perl and the rest of the operating system. We haven't tried to duplicate complete descriptions of Perl features from the main Perl documentation, which can be found in the `[.pod]` subdirectory of the Perl distribution.

We hope these notes will save you from confusion and lost sleep when writing Perl scripts on VMS. If you find we've missed something you think should appear here, please don't hesitate to drop a line to vmsperl@perl.org.

87.3 Installation

Directions for building and installing Perl 5 can be found in the file `README.vms` in the main source directory of the Perl distribution.

87.4 Organization of Perl Images

87.4.1 Core Images

During the build process, three Perl images are produced. `Miniperl.Exe` is an executable image which contains all of the basic functionality of Perl, but cannot take advantage of Perl XS extensions and has a hard-wired list of library locations for loading pure-Perl modules. It is used extensively to build and test Perl and various extensions, but is not installed.

Most of the complete Perl resides in the shareable image `PerlShr.Exe`, which provides a core to which the Perl executable image and all Perl extensions are linked. It is generally located via the logical name `PERLSHR`. While it's possible to put the image in `SYS$SHARE` to make it loadable, that's not recommended. And while you may wish to INSTALL the image for performance reasons, you should not install it with privileges; if you do, the result will not be what you expect as image privileges are disabled during Perl start-up.

Finally, `Perl.Exe` is an executable image containing the main entry point for Perl, as well as some initialization code. It should be placed in a public directory, and made world executable. In order to run Perl with command line arguments, you should define a foreign command to invoke this image.

87.4.2 Perl Extensions

Perl extensions are packages which provide both XS and Perl code to add new functionality to perl. (XS is a meta-language which simplifies writing C code which interacts with Perl, see `perlxs` for more details.) The Perl code for an extension is treated like any other library module - it's made available in your script through the appropriate `use` or `require` statement, and usually defines a Perl package containing the extension.

The portion of the extension provided by the XS code may be connected to the rest of Perl in either of two ways. In the **static** configuration, the object code for the extension is linked directly into `PerlShr.Exe`, and is initialized whenever Perl is invoked. In the **dynamic** configuration, the extension's machine code is placed into a separate shareable image, which is mapped by Perl's DynaLoader when the extension is `use`d or `require`d in your script. This allows you to maintain the extension as a separate entity, at the cost of keeping track of the additional shareable image. Most extensions can be set up as either static or dynamic.

The source code for an extension usually resides in its own directory. At least three files are generally provided: *Extshortname*`.xs` (where *Extshortname* is the portion of the extension's name following the last `::`), containing the XS code, *Extshortname*`.pm`, the Perl library module for the extension, and `Makefile.PL`, a Perl script which uses the `MakeMaker` library modules supplied with Perl to generate a `Descrip.MMS` file for the extension.

87.4.3 Installing static extensions

Since static extensions are incorporated directly into `PerlShr.Exe`, you'll have to rebuild Perl to incorporate a new extension. You should edit the main `Descrip.MMS` or `Makefile` you use to build Perl, adding the extension's name to the `ext` macro, and the extension's object file to the `extobj` macro. You'll also need to build the extension's object file, either by adding dependencies to the main `Descrip.MMS`, or using a separate `Descrip.MMS` for the extension. Then, rebuild `PerlShr.Exe` to incorporate the new code.

Finally, you'll need to copy the extension's Perl library module to the `[.`*Extname*`]` subdirectory under one of the directories in `@INC`, where *Extname* is the name of the extension, with all `::` replaced by `.` (e.g. the library module for extension Foo::Bar would be copied to a `[.Foo.Bar]` subdirectory).

87.4.4 Installing dynamic extensions

In general, the distributed kit for a Perl extension includes a file named Makefile.PL, which is a Perl program which is used to create a `Descrip.MMS` file which can be used to build and install the files required by the extension. The kit should be unpacked into a directory tree **not** under the main Perl source directory, and the procedure for building the extension is simply

```
$ perl Makefile.PL  ! Create Descrip.MMS
$ mmk               ! Build necessary files
$ mmk test          ! Run test code, if supplied
$ mmk install       ! Install into public Perl tree
```

VMS support for this process in the current release of Perl is sufficient to handle most extensions. (See the MakeMaker documentation for more details on installation options for extensions.)

- the `[.Lib.Auto.`*Arch$PVersExtname*`]` subdirectory of one of the directories in `@INC` (where *PVers* is the version of Perl you're using, as supplied in `$]`, with '.' converted to '_'), or
- one of the directories in `@INC`, or

- a directory which the extensions Perl library module passes to the DynaLoader when asking it to map the shareable image, or
- `Sys$Share` or `Sys$Library`.

If the shareable image isn't in any of these places, you'll need to define a logical name *Extshortname*, where *Extshortname* is the portion of the extension's name after the last `::`, which translates to the full file specification of the shareable image.

87.5 File specifications

87.5.1 Syntax

We have tried to make Perl aware of both VMS-style and Unix-style file specifications wherever possible. You may use either style, or both, on the command line and in scripts, but you may not combine the two styles within a single file specification. VMS Perl interprets Unix pathnames in much the same way as the CRTL (*e.g.* the first component of an absolute path is read as the device name for the VMS file specification). There are a set of functions provided in the `VMS::Filespec` package for explicit interconversion between VMS and Unix syntax; its documentation provides more details.

We've tried to minimize the dependence of Perl library modules on Unix syntax, but you may find that some of these, as well as some scripts written for Unix systems, will require that you use Unix syntax, since they will assume that '/' is the directory separator, *etc.* If you find instances of this in the Perl distribution itself, please let us know, so we can try to work around them.

Also when working on Perl programs on VMS, if you need a syntax in a specific operating system format, then you need either to check the appropriate DECC$ feature logical, or call a conversion routine to force it to that format.

The feature logical name DECC$FILENAME_UNIX_REPORT modifies traditional Perl behavior in the conversion of file specifications from Unix to VMS format in order to follow the extended character handling rules now expected by the CRTL. Specifically, when this feature is in effect, the `./.../` in a Unix path is now translated to `[.^.^.^.]` instead of the traditional VMS `[...]`. To be compatible with what MakeMaker expects, if a VMS path cannot be translated to a Unix path, it is passed through unchanged, so `unixify("[...]")` will return `[...]`.

There are several ambiguous cases where a conversion routine cannot determine whether an input filename is in Unix format or in VMS format, since now both VMS and Unix file specifications may have characters in them that could be mistaken for syntax delimiters of the other type. So some pathnames simply cannot be used in a mode that allows either type of pathname to be present. Perl will tend to assume that an ambiguous filename is in Unix format.

Allowing "." as a version delimiter is simply incompatible with determining whether a pathname is in VMS format or in Unix format with extended file syntax. There is no way to know whether "perl-5.8.6" is a Unix "perl-5.8.6" or a VMS "perl-5.8;6" when passing it to unixify() or vmsify().

The DECC$FILENAME_UNIX_REPORT logical name controls how Perl interprets filenames to the extent that Perl uses the CRTL internally for many purposes, and attempts to follow CRTL conventions for reporting filenames. The DECC$FILENAME_UNIX_ONLY feature differs in that it expects all filenames passed to the C run-time to be already in Unix format. This feature is not yet supported in Perl since Perl uses traditional OpenVMS file specifications internally and in the test harness, and it is not yet clear whether this mode will be useful or useable. The feature logical name DECC$POSIX_COMPLIANT_PATHNAMES is new with the RMS Symbolic Link SDK and included with OpenVMS v8.3, but is not yet supported in Perl.

87.5.2 Filename Case

Perl enables DECC$EFS_CASE_PRESERVE and DECC$ARGV_PARSE_STYLE by default. Note that the latter only takes effect when extended parse is set in the process in which Perl is running. When these features are explicitly disabled in the environment or the CRTL does not support them, Perl follows the traditional CRTL behavior of downcasing command-line arguments and returning file specifications in lower case only.

N. B. It is very easy to get tripped up using a mixture of other programs, external utilities, and Perl scripts that are in varying states of being able to handle case preservation. For example, a file created by an older version of an archive utility or a build utility such as MMK or MMS may generate a filename in all upper case even on an ODS-5 volume. If this filename is later retrieved by a Perl script or module in a case preserving environment, that upper case name may not match the mixed-case or lower-case expectations of the Perl code. Your best bet is to follow an all-or-nothing approach to case preservation: either don't use it at all, or make sure your entire toolchain and application environment support and use it.

OpenVMS Alpha v7.3-1 and later and all version of OpenVMS I64 support case sensitivity as a process setting (see `SET PROCESS /CASE_LOOKUP=SENSITIVE`). Perl does not currently support case sensitivity on VMS, but it may in the future, so Perl programs should use the `File::Spec->case_tolerant` method to determine the state, and not the `$^O` variable.

87.5.3 Symbolic Links

When built on an ODS-5 volume with symbolic links enabled, Perl by default supports symbolic links when the requisite support is available in the filesystem and CRTL (generally 64-bit OpenVMS v8.3 and later). There are a number of limitations and caveats to be aware of when working with symbolic links on VMS. Most notably, the target of a valid symbolic link must be expressed as a Unix-style path and it must exist on a volume visible from your POSIX root (see the `SHOW ROOT` command in DCL help). For further details on symbolic link capabilities and requirements, see chapter 12 of the CRTL manual that ships with OpenVMS v8.3 or later.

87.5.4 Wildcard expansion

File specifications containing wildcards are allowed both on the command line and within Perl globs (e.g. `<*.c>`). If the wildcard filespec uses VMS syntax, the resultant filespecs will follow VMS syntax; if a Unix-style filespec is passed in, Unix-style filespecs will be returned. Similar to the behavior of wildcard globbing for a Unix shell, one can escape command line wildcards with double quotation marks `"` around a perl program command line argument. However, owing to the stripping of `"` characters carried out by the C handling of argv you will need to escape a construct such as this one (in a directory containing the files PERL.C, PERL.EXE, PERL.H, and PERL.OBJ):

```
$ perl -e "print join(' ',@ARGV)" perl.*
perl.c perl.exe perl.h perl.obj
```

in the following triple quoted manner:

```
$ perl -e "print join(' ',@ARGV)" """perl.*"""
perl.*
```

In both the case of unquoted command line arguments or in calls to glob() VMS wildcard expansion is performed. (csh-style wildcard expansion is available if you use File::Glob::glob.) If the wildcard filespec contains a device or directory specification, then the resultant filespecs will also contain a device and directory; otherwise, device and directory information are removed. VMS-style resultant filespecs will contain a full device and directory, while Unix-style resultant filespecs will contain only as much of a directory path as was present in the input filespec. For example, if your default directory is Perl_Root:[000000], the expansion of [.t]*.* will yield filespecs like "perl_root:[t]base.dir", while the expansion of t/*/* will yield filespecs like "t/base.dir". (This is done to match the behavior of glob expansion performed by Unix shells.)

Similarly, the resultant filespec will contain the file version only if one was present in the input filespec.

87.5.5 Pipes

Input and output pipes to Perl filehandles are supported; the "file name" is passed to lib$spawn() for asynchronous execution. You should be careful to close any pipes you have opened in a Perl script, lest you leave any "orphaned" subprocesses around when Perl exits.

You may also use backticks to invoke a DCL subprocess, whose output is used as the return value of the expression. The string between the backticks is handled as if it were the argument to the system operator (see below). In this case, Perl will wait for the subprocess to complete before continuing.

The mailbox (MBX) that perl can create to communicate with a pipe defaults to a buffer size of 8192 on 64-bit systems, 512 on VAX. The default buffer size is adjustable via the logical name PERL_MBX_SIZE provided that the value falls between 128 and the SYSGEN parameter MAXBUF inclusive. For example, to set the mailbox size to 32767 use $ENV{'PERL_MBX_SIZE'} = 32767; and then open and use pipe constructs. An alternative would be to issue the command:

```
$ Define PERL_MBX_SIZE 32767
```

before running your wide record pipe program. A larger value may improve performance at the expense of the BYTLM UAF quota.

87.6 PERL5LIB and PERLLIB

The PERL5LIB and PERLLIB logical names work as documented in Section 1.1 [perl NAME], page 1, except that the element separator is '|' instead of ':'. The directory specifications may use either VMS or Unix syntax.

87.7 The Perl Forked Debugger

The Perl forked debugger places the debugger commands and output in a separate X-11 terminal window so that commands and output from multiple processes are not mixed together.

Perl on VMS supports an emulation of the forked debugger when Perl is run on a VMS system that has X11 support installed.

To use the forked debugger, you need to have the default display set to an X-11 Server and some environment variables set that Unix expects.

The forked debugger requires the environment variable `TERM` to be `xterm`, and the environment variable `DISPLAY` to exist. `xterm` must be in lower case.

```
$define TERM "xterm"

$define DISPLAY "hostname:0.0"
```

Currently the value of `DISPLAY` is ignored. It is recommended that it be set to be the hostname of the display, the server and screen in Unix notation. In the future the value of DISPLAY may be honored by Perl instead of using the default display.

It may be helpful to always use the forked debugger so that script I/O is separated from debugger I/O. You can force the debugger to be forked by assigning a value to the logical name <PERLDB_PIDS> that is not a process identification number.

```
$define PERLDB_PIDS XXXX
```

87.8 PERL_VMS_EXCEPTION_DEBUG

The PERL_VMS_EXCEPTION_DEBUG being defined as "ENABLE" will cause the VMS debugger to be invoked if a fatal exception that is not otherwise handled is raised. The purpose of this is to allow debugging of internal Perl problems that would cause such a condition.

This allows the programmer to look at the execution stack and variables to find out the cause of the exception. As the debugger is being invoked as the Perl interpreter is about to do a fatal exit, continuing the execution in debug mode is usually not practical.

Starting Perl in the VMS debugger may change the program execution profile in a way that such problems are not reproduced.

The `kill` function can be used to test this functionality from within a program.

In typical VMS style, only the first letter of the value of this logical name is actually checked in a case insensitive mode, and it is considered enabled if it is the value "T","1" or "E".

This logical name must be defined before Perl is started.

87.9 Command line

87.9.1 I/O redirection and backgrounding

Perl for VMS supports redirection of input and output on the command line, using a subset of Bourne shell syntax:

- `<file` reads stdin from `file`,
- `>file` writes stdout to `file`,
- `>>file` appends stdout to `file`,
- `2>file` writes stderr to `file`,
- `2>>file` appends stderr to `file`, and
- `2>&1` redirects stderr to stdout.

In addition, output may be piped to a subprocess, using the character '`|`'. Anything after this character on the command line is passed to a subprocess for execution; the subprocess takes the output of Perl as its input.

Finally, if the command line ends with '&', the entire command is run in the background as an asynchronous subprocess.

87.9.2 Command line switches

The following command line switches behave differently under VMS than described in Section 69.1 [perlrun NAME], page 1176. Note also that in order to pass uppercase switches to Perl, you need to enclose them in double-quotes on the command line, since the CRTL downcases all unquoted strings.

On newer 64 bit versions of OpenVMS, a process setting now controls if the quoting is needed to preserve the case of command line arguments.

-i

If the `-i` switch is present but no extension for a backup copy is given, then inplace editing creates a new version of a file; the existing copy is not deleted. (Note that if an extension is given, an existing file is renamed to the backup file, as is the case under other operating systems, so it does not remain as a previous version under the original filename.)

-S

If the `"-S"` or `-"S"` switch is present *and* the script name does not contain a directory, then Perl translates the logical name DCL$PATH as a searchlist, using each translation as a directory in which to look for the script. In addition, if no file type is specified, Perl looks in each directory for a file matching the name specified, with a blank type, a type of `.pl`, and a type of `.com`, in that order.

-u

The `-u` switch causes the VMS debugger to be invoked after the Perl program is compiled, but before it has run. It does not create a core dump file.

87.10 Perl functions

As of the time this document was last revised, the following Perl functions were implemented in the VMS port of Perl (functions marked with * are discussed in more detail below):

```
file tests*, abs, alarm, atan, backticks*, binmode*, bless,
caller, chdir, chmod, chown, chomp, chop, chr,
close, closedir, cos, crypt*, defined, delete, die, do, dump*,
each, endgrent, endpwent, eof, eval, exec*, exists, exit, exp,
fileno, flock  getc, getgrent*, getgrgid*, getgrnam, getlogin,
getppid, getpwent*, getpwnam*, getpwuid*, glob, gmtime*, goto,
grep, hex, ioctl, import, index, int, join, keys, kill*,
last, lc, lcfirst, lchown*, length, link*, local, localtime, log,
lstat, m//, map, mkdir, my, next, no, oct, open, opendir, ord,
pack, pipe, pop, pos, print, printf, push, q//, qq//, qw//,
qx//*, quotemeta, rand, read, readdir, readlink*, redo, ref,
rename, require, reset, return, reverse, rewinddir, rindex,
rmdir, s///, scalar, seek, seekdir, select(internal),
select (system call)*, setgrent, setpwent, shift, sin, sleep,
socketpair, sort, splice, split, sprintf, sqrt, srand, stat,
study, substr, symlink*, sysread, system*, syswrite, tell,
telldir, tie, time, times*, tr///, uc, ucfirst, umask,
undef, unlink*, unpack, untie, unshift, use, utime*,
values, vec, wait, waitpid*, wantarray, warn, write, y///
```

The following functions were not implemented in the VMS port, and calling them produces a fatal error (usually) or undefined behavior (rarely, we hope):

```
chroot, dbmclose, dbmopen, fork*, getpgrp, getpriority,
msgctl, msgget, msgsend, msgrcv, semctl,
semget, semop, setpgrp, setpriority, shmctl, shmget,
shmread, shmwrite, syscall
```

The following functions are available on Perls compiled with Dec C 5.2 or greater and running VMS 7.0 or greater:

```
truncate
```

The following functions are available on Perls built on VMS 7.2 or greater:

```
fcntl (without locking)
```

The following functions may or may not be implemented, depending on what type of socket support you've built into your copy of Perl:

```
accept, bind, connect, getpeername,
gethostbyname, getnetbyname, getprotobyname,
getservbyname, gethostbyaddr, getnetbyaddr,
getprotobynumber, getservbyport, gethostent,
getnetent, getprotoent, getservent, sethostent,
setnetent, setprotoent, setservent, endhostent,
endnetent, endprotoent, endservent, getsockname,
getsockopt, listen, recv, select(system call)*,
send, setsockopt, shutdown, socket
```

The following function is available on Perls built on 64 bit OpenVMS v8.2 with hard links enabled on an ODS-5 formatted build disk. CRTL support is in principle available as of OpenVMS v7.3-1, and better configuration support could detect this.

```
link
```

The following functions are available on Perls built on 64 bit OpenVMS v8.2 and later. CRTL support is in principle available as of OpenVMS v7.3-2, and better configuration support could detect this.

getgrgid, getgrnam, getpwnam, getpwuid,
setgrent, ttyname

The following functions are available on Perls built on 64 bit OpenVMS v8.2 and later.

statvfs, socketpair

File tests

The tests -b, -B, -c, -C, -d, -e, -f, -o, -M, -s, -S, -t, -T, and -z work as advertised. The return values for -r, -w, and -x tell you whether you can actually access the file; this may not reflect the UIC-based file protections. Since real and effective UIC don't differ under VMS, -O, -R, -W, and -X are equivalent to -o, -r, -w, and -x. Similarly, several other tests, including -A, -g, -k, -l, -p, and -u, aren't particularly meaningful under VMS, and the values returned by these tests reflect whatever your CRTL stat() routine does to the equivalent bits in the st_mode field. Finally, -d returns true if passed a device specification without an explicit directory (e.g. DUA1:), as well as if passed a directory.

There are DECC feature logical names AND ODS-5 volume attributes that also control what values are returned for the date fields.

Note: Some sites have reported problems when using the file-access tests (-r, -w, and -x) on files accessed via DEC's DFS. Specifically, since DFS does not currently provide access to the extended file header of files on remote volumes, attempts to examine the ACL fail, and the file tests will return false, with $! indicating that the file does not exist. You can use stat on these files, since that checks UIC-based protection only, and then manually check the appropriate bits, as defined by your C compiler's stat.h, in the mode value it returns, if you need an approximation of the file's protections.

backticks

Backticks create a subprocess, and pass the enclosed string to it for execution as a DCL command. Since the subprocess is created directly via lib$spawn(), any valid DCL command string may be specified.

binmode FILEHANDLE

The binmode operator will attempt to insure that no translation of carriage control occurs on input from or output to this filehandle. Since this involves reopening the file and then restoring its file position indicator, if this function returns FALSE, the underlying filehandle may no longer point to an open file, or may point to a different position in the file than before binmode was called.

Note that binmode is generally not necessary when using normal filehandles; it is provided so that you can control I/O to existing record-structured files when necessary. You can also use the vmsfopen function in the VMS::Stdio extension to gain finer control of I/O to files and devices with different record structures.

crypt PLAINTEXT, USER

The crypt operator uses the sys$hash_password system service to generate the hashed representation of PLAINTEXT. If USER is a valid username, the algorithm and salt values are taken from that user's UAF record. If it is not, then the preferred algorithm and a salt of 0 are used. The quadword encrypted value is returned as an 8-character string.

The value returned by crypt may be compared against the encrypted password from the UAF returned by the getpw* functions, in order to authenticate users. If you're going to do this, remember that the encrypted password in the UAF was generated using uppercase username and password strings; you'll have to upcase the arguments to crypt to insure that you'll get the proper value:

```
sub validate_passwd {
    my($user,$passwd) = @_;
    my($pwdhash);
    if ( !($pwdhash = (getpwnam($user))[1]) ||
           $pwdhash ne crypt("\U$passwd","\U$name") ) {
        intruder_alert($name);
    }
    return 1;
}
```

die

die will force the native VMS exit status to be an SS$_ABORT code if neither of the $! or $? status values are ones that would cause the native status to be interpreted as being what VMS classifies as SEVERE_ERROR severity for DCL error handling.

When PERL_VMS_POSIX_EXIT is active (see [$?], page 1424 below), the native VMS exit status value will have either one of the $! or $? or $^E or the Unix value 255 encoded into it in a way that the effective original value can be decoded by other programs written in C, including Perl and the GNV package. As per the normal non-VMS behavior of die if either $! or $? are non-zero, one of those values will be encoded into a native VMS status value. If both of the Unix status values are 0, and the $^E value is set one of ERROR or SEVERE_ERROR severity, then the $^E value will be used as the exit code as is. If none of the above apply, the Unix value of 255 will be encoded into a native VMS exit status value.

Please note a significant difference in the behavior of die in the PERL_VMS_POSIX_EXIT mode is that it does not force a VMS SEVERE_ERROR status on exit. The Unix exit values of 2 through 255 will be encoded in VMS status values with severity levels of SUCCESS. The Unix exit value of 1 will be encoded in a VMS status value with a severity level of ERROR. This is to be compatible with how the VMS C library encodes these values.

The minimum severity level set by die in PERL_VMS_POSIX_EXIT mode may be changed to be ERROR or higher in the future depending on the results of testing and further review.

See [$?], page 1424 for a description of the encoding of the Unix value to produce a native VMS status containing it.

dump

Rather than causing Perl to abort and dump core, the dump operator invokes the VMS debugger. If you continue to execute the Perl program under the debugger, control will be transferred to the label specified as the argument to `dump`, or, if no label was specified, back to the beginning of the program. All other state of the program (*e.g.* values of variables, open file handles) are not affected by calling `dump`.

exec LIST

A call to `exec` will cause Perl to exit, and to invoke the command given as an argument to `exec` via `lib$do_command`. If the argument begins with '@' or '$' (other than as part of a filespec), then it is executed as a DCL command. Otherwise, the first token on the command line is treated as the filespec of an image to run, and an attempt is made to invoke it (using `.Exe` and the process defaults to expand the filespec) and pass the rest of `exec`'s argument to it as parameters. If the token has no file type, and matches a file with null type, then an attempt is made to determine whether the file is an executable image which should be invoked using `MCR` or a text file which should be passed to DCL as a command procedure.

fork

While in principle the `fork` operator could be implemented via (and with the same rather severe limitations as) the CRTL `vfork()` routine, and while some internal support to do just that is in place, the implementation has never been completed, making `fork` currently unavailable. A true kernel `fork()` is expected in a future version of VMS, and the pseudo-fork based on interpreter threads may be available in a future version of Perl on VMS (see Section 23.1 [perlfork NAME], page 337). In the meantime, use `system`, backticks, or piped filehandles to create subprocesses.

getpwent

getpwnam

getpwuid

These operators obtain the information described in Section 25.1 [perlfunc NAME], page 351, if you have the privileges necessary to retrieve the named user's UAF information via `sys$getuai`. If not, then only the `$name`, `$uid`, and `$gid` items are returned. The `$dir` item contains the login directory in VMS syntax, while the `$comment` item contains the login directory in Unix syntax. The `$gcos` item contains the owner field from the UAF record. The `$quota` item is not used.

gmtime

The `gmtime` operator will function properly if you have a working CRTL `gmtime()` routine, or if the logical name SYS$TIMEZONE_DIFFERENTIAL is defined as the number of seconds which must be added to UTC to yield local time. (This logical name is defined automatically if you are running a version of VMS with built-in UTC support.) If neither of these cases is true, a warning message is printed, and `undef` is returned.

kill

In most cases, `kill` is implemented via the undocumented system service `$SIGPRC`, which has the same calling sequence as `$FORCEX`, but throws an exception in the target process rather than forcing it to call `$EXIT`. Generally speaking, `kill` follows the behavior of the CRTL's `kill()` function, but unlike that function can be called from within a signal handler. Also, unlike the `kill` in some versions of the CRTL, Perl's `kill` checks the validity of the signal passed in and returns an error rather than attempting to send an unrecognized signal.

Also, negative signal values don't do anything special under VMS; they're just converted to the corresponding positive value.

qx//

See the entry on `backticks` above.

select (system call)

If Perl was not built with socket support, the system call version of `select` is not available at all. If socket support is present, then the system call version of `select` functions only for file descriptors attached to sockets. It will not provide information about regular files or pipes, since the CRTL `select()` routine does not provide this functionality.

stat EXPR

Since VMS keeps track of files according to a different scheme than Unix, it's not really possible to represent the file's ID in the `st_dev` and `st_ino` fields of a `struct stat`. Perl tries its best, though, and the values it uses are pretty unlikely to be the same for two different files. We can't guarantee this, though, so caveat scriptor.

system LIST

The `system` operator creates a subprocess, and passes its arguments to the subprocess for execution as a DCL command. Since the subprocess is created directly via `lib$spawn()`, any valid DCL command string may be specified. If the string begins with '@', it is treated as a DCL command unconditionally. Otherwise, if the first token contains a character used as a delimiter in file specification (e.g. `:` or `]`), an attempt is made to expand it using a default type of `.Exe` and the process defaults, and if successful, the resulting file is invoked via `MCR`. This allows you to invoke an image directly simply by passing the file specification to `system`, a common Unixish idiom. If the token has no file type, and matches a file with null type, then an attempt is made to determine whether the file is an executable image which should be invoked using `MCR` or a text file which should be passed to DCL as a command procedure.

If LIST consists of the empty string, `system` spawns an interactive DCL subprocess, in the same fashion as typing **SPAWN** at the DCL prompt.

Perl waits for the subprocess to complete before continuing execution in the current process. As described in Section 25.1 [perlfunc NAME], page 351, the return value of `system` is a fake "status" which follows POSIX semantics unless the pragma `use vmsish 'status'` is in effect; see the description of `$?` in this document for more detail.

time

The value returned by `time` is the offset in seconds from 01-JAN-1970 00:00:00 (just like the CRTL's times() routine), in order to make life easier for code coming in from the POSIX/Unix world.

times

The array returned by the `times` operator is divided up according to the same rules the CRTL `times()` routine. Therefore, the "system time" elements will always be 0, since there is no difference between "user time" and "system" time under VMS, and the time accumulated by a subprocess may or may not appear separately in the "child time" field, depending on whether `times()` keeps track of subprocesses separately. Note especially that the VAXCRTL (at least) keeps track only of subprocesses spawned using `fork()` and `exec()`; it will not accumulate the times of subprocesses spawned via pipes, `system()`, or backticks.

unlink LIST

`unlink` will delete the highest version of a file only; in order to delete all versions, you need to say

```
1 while unlink LIST;
```

You may need to make this change to scripts written for a Unix system which expect that after a call to `unlink`, no files with the names passed to `unlink` will exist. (Note: This can be changed at compile time; if you `use Config` and `$Config{'d_unlink_all_versions'}` is `define`, then `unlink` will delete all versions of a file on the first call.)

`unlink` will delete a file if at all possible, even if it requires changing file protection (though it won't try to change the protection of the parent directory). You can tell whether you've got explicit delete access to a file by using the `VMS::Filespec::candelete` operator. For instance, in order to delete only files to which you have delete access, you could say something like

```
sub safe_unlink {
    my($file,$num);
    foreach $file (@_) {
        next unless VMS::Filespec::candelete($file);
        $num += unlink $file;
    }
    $num;
}
```

(or you could just use `VMS::Stdio::remove`, if you've installed the VMS::Stdio extension distributed with Perl). If `unlink` has to change the file protection to delete the file, and you interrupt it in midstream, the file may be left intact, but with a changed ACL allowing you delete access.

This behavior of `unlink` is to be compatible with POSIX behavior and not traditional VMS behavior.

utime LIST

This operator changes only the modification time of the file (VMS revision date) on ODS-2 volumes and ODS-5 volumes without access dates enabled. On ODS-5 volumes with access dates enabled, the true access time is modified.

waitpid PID,FLAGS

If PID is a subprocess started by a piped `open()` (see `open`), `waitpid` will wait for that subprocess, and return its final status value in `$?`. If PID is a subprocess created in some other way (e.g. SPAWNed before Perl was invoked), `waitpid` will simply check once per second whether the process has completed, and return when it has. (If PID specifies a process that isn't a subprocess of the current process, and you invoked Perl with the `-w` switch, a warning will be issued.)

Returns PID on success, -1 on error. The FLAGS argument is ignored in all cases.

87.11 Perl variables

The following VMS-specific information applies to the indicated "special" Perl variables, in addition to the general information in Section 86.1 [perlvar NAME], page 1375. Where there is a conflict, this information takes precedence.

%ENV

The operation of the `%ENV` array depends on the translation of the logical name `PERL_ENV_TABLES`. If defined, it should be a search list, each element of which specifies a location for `%ENV` elements. If you tell Perl to read or set the element `$ENV{`*name*`}`, then Perl uses the translations of `PERL_ENV_TABLES` as follows:

CRTL_ENV

This string tells Perl to consult the CRTL's internal `environ` array of key-value pairs, using *name* as the key. In most cases, this contains only a few keys, but if Perl was invoked via the C `exec[lv]e()` function, as is the case for some embedded Perl applications or when running under a shell such as GNV bash, the `environ` array may have been populated by the calling program.

CLISYM_[LOCAL]

A string beginning with `CLISYM_`tells Perl to consult the CLI's symbol tables, using *name* as the name of the symbol. When reading an element of `%ENV`, the local symbol table is scanned first, followed by the global symbol table.. The characters following `CLISYM_` are significant when an element of `%ENV` is set or deleted: if the complete string is `CLISYM_LOCAL`, the change is made in the local symbol table; otherwise the global symbol table is changed.

Any other string

If an element of `PERL_ENV_TABLES` translates to any other string, that string is used as the name of a logical name table, which is consulted using *name* as the logical name. The normal search order of access modes is used.

PERL_ENV_TABLES is translated once when Perl starts up; any changes you make while Perl is running do not affect the behavior of %ENV. If PERL_ENV_TABLES is not defined, then Perl defaults to consulting first the logical name tables specified by LNM$FILE_DEV, and then the CRTL environ array. This default order is reversed when the logical name GNV$UNIX_SHELL is defined, such as when running under GNV bash.

In all operations on %ENV, the key string is treated as if it were entirely uppercase, regardless of the case actually specified in the Perl expression.

When an element of %ENV is read, the locations to which PERL_ENV_TABLES points are checked in order, and the value obtained from the first successful lookup is returned. If the name of the %ENV element contains a semi-colon, it and any characters after it are removed. These are ignored when the CRTL environ array or a CLI symbol table is consulted. However, the name is looked up in a logical name table, the suffix after the semi-colon is treated as the translation index to be used for the lookup. This lets you look up successive values for search list logical names. For instance, if you say

```
   $  Define STORY  once,upon,a,time,there,was
   $  perl -e "for ($i = 0; $i <= 6; $i++) " -
   _$ -e "{ print $ENV{'story;'.$i},' '}"
```

Perl will print ONCE UPON A TIME THERE WAS, assuming, of course, that PERL_ENV_TABLES is set up so that the logical name story is found, rather than a CLI symbol or CRTL environ element with the same name.

When an element of %ENV is set to a defined string, the corresponding definition is made in the location to which the first translation of PERL_ENV_TABLES points. If this causes a logical name to be created, it is defined in supervisor mode. (The same is done if an existing logical name was defined in executive or kernel mode; an existing user or supervisor mode logical name is reset to the new value.) If the value is an empty string, the logical name's translation is defined as a single NUL (ASCII \0) character, since a logical name cannot translate to a zero-length string. (This restriction does not apply to CLI symbols or CRTL environ values; they are set to the empty string.)

When an element of %ENV is set to undef, the element is looked up as if it were being read, and if it is found, it is deleted. (An item "deleted" from the CRTL environ array is set to the empty string.) Using delete to remove an element from %ENV has a similar effect, but after the element is deleted, another attempt is made to look up the element, so an inner-mode logical name or a name in another location will replace the logical name just deleted. In either case, only the first value found searching PERL_ENV_TABLES is altered. It is not possible at present to define a search list logical name via %ENV.

The element $ENV{DEFAULT} is special: when read, it returns Perl's current default device and directory, and when set, it resets them, regardless of the definition of PERL_ENV_TABLES. It cannot be cleared or deleted; attempts to do so are silently ignored.

Note that if you want to pass on any elements of the C-local environ array to a subprocess which isn't started by fork/exec, or isn't running a C program, you can "promote" them to logical names in the current process, which will then be inherited by all subprocesses, by saying

```
foreach my $key (qw[C-local keys you want promoted]) {
    my $temp = $ENV{$key}; # read from C-local array
    $ENV{$key} = $temp;    # and define as logical name
}
```

(You can't just say `$ENV{$key} = $ENV{$key}`, since the Perl optimizer is smart enough to elide the expression.)

Don't try to clear `%ENV` by saying `%ENV = ();`, it will throw a fatal error. This is equivalent to doing the following from DCL:

```
DELETE/LOGICAL *
```

You can imagine how bad things would be if, for example, the SYS$MANAGER or SYS$SYSTEM logical names were deleted.

At present, the first time you iterate over %ENV using `keys`, or `values`, you will incur a time penalty as all logical names are read, in order to fully populate %ENV. Subsequent iterations will not reread logical names, so they won't be as slow, but they also won't reflect any changes to logical name tables caused by other programs.

You do need to be careful with the logical names representing process-permanent files, such as `SYS$INPUT` and `SYS$OUTPUT`. The translations for these logical names are prepended with a two-byte binary value (0x1B 0x00) that needs to be stripped off if you want to use it. (In previous versions of Perl it wasn't possible to get the values of these logical names, as the null byte acted as an end-of-string marker)

$!

The string value of `$!` is that returned by the CRTL's strerror() function, so it will include the VMS message for VMS-specific errors. The numeric value of `$!` is the value of `errno`, except if errno is EVMSERR, in which case `$!` contains the value of vaxc$errno. Setting `$!` always sets errno to the value specified. If this value is EVMSERR, it also sets vaxc$errno to 4 (NONAME-F-NOMSG), so that the string value of `$!` won't reflect the VMS error message from before `$!` was set.

$^E

This variable provides direct access to VMS status values in vaxc$errno, which are often more specific than the generic Unix-style error messages in `$!`. Its numeric value is the value of vaxc$errno, and its string value is the corresponding VMS message string, as retrieved by sys$getmsg(). Setting `$^E` sets vaxc$errno to the value specified.

While Perl attempts to keep the vaxc$errno value to be current, if errno is not EVMSERR, it may not be from the current operation.

$?

The "status value" returned in `$?` is synthesized from the actual exit status of the subprocess in a way that approximates POSIX wait(5) semantics, in order to allow Perl programs to portably test for successful completion of subprocesses. The low order 8 bits of `$?` are always 0 under VMS, since the termination status of a process may or may not have been generated by an exception.

The next 8 bits contain the termination status of the program.

If the child process follows the convention of C programs compiled with the _POSIX_EXIT macro set, the status value will contain the actual value of 0 to 255 returned by that program on a normal exit.

With the _POSIX_EXIT macro set, the Unix exit value of zero is represented as a VMS native status of 1, and the Unix values from 2 to 255 are encoded by the equation:

```
VMS_status = 0x35a000 + (unix_value * 8) + 1.
```

And in the special case of Unix value 1 the encoding is:

```
VMS_status = 0x35a000 + 8 + 2 + 0x10000000.
```

For other termination statuses, the severity portion of the subprocess's exit status is used: if the severity was success or informational, these bits are all 0; if the severity was warning, they contain a value of 1; if the severity was error or fatal error, they contain the actual severity bits, which turns out to be a value of 2 for error and 4 for severe_error. Fatal is another term for the severe_error status.

As a result, `$?` will always be zero if the subprocess's exit status indicated successful completion, and non-zero if a warning or error occurred or a program compliant with encoding _POSIX_EXIT values was run and set a status.

How can you tell the difference between a non-zero status that is the result of a VMS native error status or an encoded Unix status? You can not unless you look at the ${^CHILD_ERROR_NATIVE} value. The ${^CHILD_ERROR_NATIVE} value returns the actual VMS status value and check the severity bits. If the severity bits are equal to 1, then if the numeric value for `$?` is between 2 and 255 or 0, then `$?` accurately reflects a value passed back from a Unix application. If `$?` is 1, and the severity bits indicate a VMS error (2), then `$?` is from a Unix application exit value.

In practice, Perl scripts that call programs that return _POSIX_EXIT type status values will be expecting those values, and programs that call traditional VMS programs will either be expecting the previous behavior or just checking for a non-zero status.

And success is always the value 0 in all behaviors.

When the actual VMS termination status of the child is an error, internally the `$!` value will be set to the closest Unix errno value to that error so that Perl scripts that test for error messages will see the expected Unix style error message instead of a VMS message.

Conversely, when setting `$?` in an END block, an attempt is made to convert the POSIX value into a native status intelligible to the operating system upon exiting Perl. What this boils down to is that setting `$?` to zero results in the generic success value SS$_NORMAL, and setting `$?` to a non-zero value results in the generic failure status SS$_ABORT. See also [perlport exit], page 973.

With the `PERL_VMS_POSIX_EXIT` logical name defined as "ENABLE", setting `$?` will cause the new value to be encoded into `$^E` so that either the original parent or child exit status values 0 to 255 can be automatically recovered by C programs expecting _POSIX_EXIT behavior. If both a parent and a child exit value are non-zero, then it will be assumed that this is actually a VMS native status value to be passed through. The special value of 0xFFFF is almost a NOOP as it will cause the current native VMS status in the C library to become the current native Perl VMS status, and is handled this way as it is known to not be a valid native VMS status value. It is recommend that only values in the range of normal Unix parent or child status numbers, 0 to 255 are used.

The pragma `use vmsish 'status'` makes `$?` reflect the actual VMS exit status instead of the default emulation of POSIX status described above. This pragma also disables the conversion of non-zero values to SS$_ABORT when setting `$?` in an END block (but zero will still be converted to SS$_NORMAL).

Do not use the pragma `use vmsish 'status'` with `PERL_VMS_POSIX_EXIT` enabled, as they are at times requesting conflicting actions and the consequence of ignoring this advice will be undefined to allow future improvements in the POSIX exit handling.

In general, with `PERL_VMS_POSIX_EXIT` enabled, more detailed information will be available in the exit status for DCL scripts or other native VMS tools, and will give the expected information for Posix programs. It has not been made the default in order to preserve backward compatibility.

N.B. Setting `DECC$FILENAME_UNIX_REPORT` implicitly enables `PERL_VMS_POSIX_EXIT`.

`$|`

Setting `$|` for an I/O stream causes data to be flushed all the way to disk on each write (*i.e.* not just to the underlying RMS buffers for a file). In other words, it's equivalent to calling fflush() and fsync() from C.

87.12 Standard modules with VMS-specific differences

87.12.1 SDBM_File

SDBM_File works properly on VMS. It has, however, one minor difference. The database directory file created has a `.sdbm_dir` extension rather than a `.dir` extension. `.dir` files are VMS filesystem directory files, and using them for other purposes could cause unacceptable problems.

87.13 Revision date

Please see the git repository for revision history.

87.14 AUTHOR

Charles Bailey bailey@cor.newman.upenn.edu Craig Berry craigberry@mac.com Dan Sugalski dan@sidhe.org John Malmberg wb8tyw@qsl.net