Perl Documentation

16th December 2004

This compilation in one single PDF document has been conceived using the Perl pod2latex script to get Latex, then converted into PDF thanks to pdflatex. If you use the PDF version of this documentation (or a paper version stemming from the pdf one) for any other use than a personal one, I would be thankful if you could keep me informed by email (Paul.Gaborit@enstimac.fr), although you are under no obligation.

Cette compilation en un seul document PDF a été conçue en utilisant le script Perl `pod2latex` pour obtenir du LaTeX puis convertie en PDF grâce à `pdflatex`. Si vous utilisez la version PDF de cette documentation (ou une version papier issue de la version PDF) pour tout autre usage qu'un usage personnel, je vous serai reconnaissant de m'en informer par un petit message (Paul.Gaborit @ enstimac.fr) mais rien ne vous y oblige.

16th December 2004 – Paul Gaborit

Part I

Overview

Chapter 1

perl

Practical Extraction and Report Language

1.1 SYNOPSIS

perl [**-sTuU**] [**-hv**] [**-V**[:*configvar*]] [**-cw**] [**-d**[:*debugger*]] [**-D**[*number/list*]]
[**-pna**] [**-F***pattern*] [**-l**[*octal*]] [**-0**[*octal*]] [**-I***dir*] [**-m**[-]*module*] [**-M**[-]*'module...'*] [**-P**] [**-S**] [**-x**[*dir*]]
[**-i**[*extension*]] [**-e** *'command'*] [–] [*programfile*] [*argument*]...

If you're new to Perl, you should start with *perlintro*, which is a general intro for beginners and provides some background to help you navigate the rest of Perl's extensive documentation.

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

1.1.1 Overview

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

1.1.2 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

    perlboot            Perl OO tutorial for beginners
    perltoot            Perl OO tutorial, part 1
    perltooc            Perl OO tutorial, part 2
    perlbot             Perl OO tricks and examples

    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
```

1.1.3 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
perlrun             Perl execution and options
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
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
  perlothrtut       Old Perl threads tutorial

perlport            Perl portability guide
perllocale          Perl locale support
perluniintro        Perl Unicode introduction
perlunicode         Perl Unicode support
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
```

```
    perlutil            utilities packaged with the Perl distribution

    perlcompile         Perl compiler suite intro

    perlfilter          Perl source filters
```

1.1.4 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
    perlclib            Internal replacements for standard C library functions
    perlguts            Perl internal functions for those doing extensions
    perlcall            Perl calling conventions from C

    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
```

1.1.5 Miscellaneous

```
    perlbook            Perl book information
    perltodo            Perl things to do

    perldoc             Look up Perl documentation in Pod format

    perlhist            Perl history records
    perldelta           Perl changes since previous version
    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
    perl573delta        Perl changes in version 5.7.3
    perl572delta        Perl changes in version 5.7.2
    perl571delta        Perl changes in version 5.7.1
    perl570delta        Perl changes in version 5.7.0
    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

    perlartistic        Perl Artistic License
    perlgpl             GNU General Public License
```

1.1.6 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)
```

1.1.7 Platform-Specific

perlaix	Perl notes for AIX
perlamiga	Perl notes for AmigaOS
perlapollo	Perl notes for Apollo DomainOS
perlbeos	Perl notes for BeOS
perlbs2000	Perl notes for POSIX-BC BS2000
perlce	Perl notes for WinCE
perlcygwin	Perl notes for Cygwin
perldgux	Perl notes for DG/UX
perldos	Perl notes for DOS
perlepoc	Perl notes for EPOC
perlfreebsd	Perl notes for FreeBSD
perlhpux	Perl notes for HP-UX
perlhurd	Perl notes for Hurd
perlirix	Perl notes for Irix
perlmachten	Perl notes for Power MachTen
perlmacos	Perl notes for Mac OS (Classic)
perlmacosx	Perl notes for Mac OS X
perlmint	Perl notes for MiNT
perlmpeix	Perl notes for MPE/iX
perlnetware	Perl notes for NetWare
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
perlsolaris	Perl notes for Solaris
perltru64	Perl notes for Tru64
perluts	Perl notes for UTS
perlvmesa	Perl notes for VM/ESA
perlvms	Perl notes for VMS
perlvos	Perl notes for Stratus VOS
perlwin32	Perl notes for Windows

By default, the manpages listed above are installed in the */usr/local/man/* directory.

Extensive additional documentation for Perl modules is available. The default configuration for perl will place this additional documentation in the */usr/local/lib/perl5/man* directory (or else in the *man* subdirectory of the Perl library directory). Some of this additional documentation is distributed standard with Perl, but you'll also find documentation for third-party modules there.

You should be able to view Perl's documentation with your man(1) program by including the proper directories in the appropriate start-up files, or in the MANPATH environment variable. To find out where the configuration has installed the manpages, type:

```
    perl -V:man.dir
```

If the directories have a common stem, such as */usr/local/man/man1* and */usr/local/man/man3*, you need only to add that stem (*/usr/local/man*) to your man(1) configuration files or your MANPATH environment variable. If they do not share a stem, you'll have to add both stems.

If that doesn't work for some reason, you can still use the supplied *perldoc* script to view module information. You might also look into getting a replacement man program.

If something strange has gone wrong with your program and you're not sure where you should look for help, try the **-w** switch first. It will often point out exactly where the trouble is.

1.2 DESCRIPTION

Perl is a language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal).

Perl combines (in the author's opinion, anyway) some of the best features of C, **sed**, **awk**, and **sh**, so people familiar with those languages should have little difficulty with it. (Language historians will also note some vestiges of **csh**, Pascal, and even BASIC-PLUS.) Expression syntax corresponds closely to C expression syntax. Unlike most Unix utilities, Perl does not arbitrarily limit the size of your data–if you've got the memory, Perl can slurp in your whole file as a single string. Recursion is of unlimited depth. And the tables used by hashes (sometimes called "associative arrays") grow as necessary to prevent degraded performance. Perl can use sophisticated pattern matching techniques to scan large amounts of data quickly. Although optimized for scanning text, Perl can also deal with binary data, and can make dbm files look like hashes. Setuid Perl scripts are safer than C programs through a dataflow tracing mechanism that prevents many stupid security holes.

If you have a problem that would ordinarily use **sed** or **awk** or **sh**, but it exceeds their capabilities or must run a little faster, and you don't want to write the silly thing in C, then Perl may be for you. There are also translators to turn your **sed** and **awk** scripts into Perl scripts.

But wait, there's more...

Begun in 1993 (see *perlhist*), Perl version 5 is nearly a complete rewrite that provides the following additional benefits:

- modularity and reusability using innumerable modules

  Described in *perlmod*, *perlmodlib*, and *perlmodinstall*.

- embeddable and extensible

  Described in *perlembed*, *perlxstut*, *perlxs*, *perlcall*, *perlguts*, and *xsubpp*.

- roll-your-own magic variables (including multiple simultaneous DBM implementations)

  Described in *perltie* and AnyDBM_File.

- subroutines can now be overridden, autoloaded, and prototyped

  Described in *perlsub*.

- arbitrarily nested data structures and anonymous functions

  Described in *perlreftut*, *perlref*, *perldsc*, and *perllol*.

- object-oriented programming

  Described in *perlobj*, *perlboot*, *perltoot*, *perltooc*, and *perlbot*.

- support for light-weight processes (threads)

  Described in *perlthrtut* and *threads*.

- support for Unicode, internationalization, and localization

  Described in *perluniintro*, *perllocale* and *Locale::Maketext*.

- lexical scoping

  Described in *perlsub*.

- regular expression enhancements

  Described in *perlre*, with additional examples in *perlop*.

- enhanced debugger and interactive Perl environment, with integrated editor support

  Described in *perldebtut*, *perldebug* and *perldebguts*.

- POSIX 1003.1 compliant library

  Described in *POSIX*.

Okay, that's *definitely* enough hype.

1.3 AVAILABILITY

Perl is available for most operating systems, including virtually all Unix-like platforms. See Supported Platforms in *perlport* for a listing.

1.4 ENVIRONMENT

See *perlrun*.

1.5 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 .

1.6 FILES

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

1.7 SEE ALSO

```
 a2p    awk to perl translator
 s2p    sed to perl translator

 http://www.perl.com/       the Perl Home Page
 http://www.cpan.org/       the Comprehensive Perl Archive
 http://www.perl.org/       Perl Mongers (Perl user groups)
```

1.8 DIAGNOSTICS

The `use warnings` pragma (and the **-w** switch) produces some lovely diagnostics.

See *perldiag* 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 *perlsec*.

Did we mention that you should definitely consider using the **-w** switch?

1.9 BUGS

The **-w** switch 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.

1.10 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.

Chapter 2

perlintro – a brief introduction and overview of Perl

2.1 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.

2.1.1 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 *perl*, *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.

2.1.2 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).

For more information, including instructions for other platforms such as Windows and Mac OS, read *perlrun*.

2.1.3 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 *perlsyn*.

2.1.4 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 pre-declare your variable types.

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 *perlvar*. 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 *perlvar*.

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 *perlvar*.

Scalars, arrays and hashes are documented more fully in *perldata*.

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 *perlreftut*, *perllol*, *perlref* and *perldsc*.

2.1.5 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 $a = "foo";
if ($some_condition) {
    my $b = "bar";
    print $a;           # prints "foo"
    print $b;           # prints "bar"
}
print $a;               # prints "foo"
print $b;               # prints nothing; $b 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 $b` would cause a compile-time error and prevent you from running the program. Using `strict` is highly recommended.

2.1.6 Conditional and looping constructs

Perl has most of the usual conditional and looping constructs except for case/switch (but if you really want it, there is a Switch module in Perl 5.8 and newer, and on CPAN. See the section on modules, below, for more information about modules and CPAN).

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";
    }

    # you don't have to use the default $_ either...
    foreach my $key (keys %hash) {
        print "The value of $key is $hash{$key}\n";
    }
```

For more detail on looping constructs (and some that weren't mentioned in this overview) see *perlsyn*.

2.1.7 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 *perlfunc* and you can easily read about any given function by using `perldoc -f` *`functionname`*.

Perl operators are documented in full in *perlop*, 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 *perlop* for more detail.)

Miscellaneous

```
=   assignment
.   string concatenation
x   string multiplication
..  range operator (creates a list of numbers)
```

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";
```

2.1.8 Files and I/O

You can open a file for input or output using the `open()` function. It's documented in extravagant detail in *perlfunc* and *perlopentut*, but in short:

```
open(INFILE,  "input.txt")   or die "Can't open input.txt: $!";
open(OUTFILE, ">output.txt") or die "Can't open output.txt: $!";
open(LOGFILE, ">>my.log")    or die "Can't open logfile: $!";
```

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  = <INFILE>;
my @lines = <INFILE>;
```

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 (<INFILE>) {     # 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 OUTFILE $record;
print LOGFILE $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 INFILE;
```

2.1.9 Regular expressions

Perl's regular expression support is both broad and deep, and is the subject of lengthy documentation in *perlrequick*, *perlretut*, 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 *perlop*. It operates on `$_` by default, or can be bound to another variable using the =˜ binding operator (also documented in *perlop*).

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 *perlop*.

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 *perlre*, 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}/         a 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 *perlrequick*, *perlretut*, and *perlre*.

2.1.10 Writing subroutines

Writing subroutines is easy:

```
    sub log {
        my $logmessage = shift;
        print LOGFILE $logmessage;
    }
```

What's that `shift`? Well, the arguments to a subroutine are available to us as a special array called `@_` (see *perlvar* 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;
    }
```

For more information on writing subroutines, see *perlsub*.

2.1.11 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 *perlboot*, *perltoot*, *perltooc* and *perlobj*.

As a beginning Perl programmer, your most common use of OO Perl will be in using third-party modules, which are documented below.

2.1.12 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 *perlmodinstall*

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.

perlmod describes Perl modules in general. *perlmodlib* lists the modules which came with your Perl installation.

If you feel the urge to write Perl modules, *perlnewmod* will give you good advice.

2.2 AUTHOR

Kirrily "Skud" Robert <skud@cpan.org>

Part II

Tutorials

Chapter 3

perlreftut

Mark's very short tutorial about references

3.1 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%.

3.2 Who Needs Complicated Data Structures?

One problem that came up all the time in Perl 4 was how to represent a hash whose values were lists. Perl 4 had hashes, of course, but the values had to be scalars; they couldn'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 can't be lists, you lose. In Perl 4, hash values can't be lists; they can only be strings. 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.

3.3 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 "George Bush".

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 "George Bush", 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.

3.4 Syntax

There are just two ways to make a reference, and just two ways to use it once you have it.

3.4.1 Making References

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
```

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.

3.4.2 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:

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";
}
```

Use Rule 2

Use Rule 1 is all you really need, because it tells you how to to 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.

3.4.3 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:

3.4.4 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]`.

3.5 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 they 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.

3.6 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 *perlref* manual page, which discusses 100% of the details.

Some of the highlights of *perlref*:

- 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 *perllol* instead of *perlref*; it discusses lists of lists and multidimensional arrays in detail. After that, you should move on to *perldsc*; 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.

3.7 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.

3.8 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*.

3.8.1 Distribution Conditions

Copyright 1998 The Perl Journal.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in these files are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

Chapter 4

perldsc

Perl Data Structures Cookbook

4.1 DESCRIPTION

The single feature most sorely lacking in the Perl programming language prior to its 5.0 release was complex data structures. Even without direct language support, some valiant programmers did manage to emulate them, but it was hard work and not for the faint of heart. You could occasionally get away with the `$m{$AoA,$b}` notation borrowed from **awk** in which the keys are actually more like a single concatenated string `"$AoA$b"`, but traversal and sorting were difficult. More desperate programmers even hacked Perl's internal symbol table directly, a strategy that proved hard to develop and maintain–to put it mildly.

The 5.0 release of Perl let us have complex data structures. You may now write something like this and all of a sudden, you'd have an array with three dimensions!

```
for $x (1 .. 10) {
    for $y (1 .. 10) {
        for $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.

4.2 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 `@ARRAY`s and `%HASH`es 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 the perlref(1) man page. 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:

```
    @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]`.

4.3 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 $i (1..10) {
        @array = somefunc($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 $i (1..10) {
    @array = somefunc($i);
    $counts[$i] = scalar @array;
}
```

Here's the case of taking a reference to the same memory location again and again:

```
for $i (1..10) {
    @array = somefunc($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:

```
for $i (1..10) {
    @array = somefunc($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:

```
for $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 $i (1..10) {
        my @array = somefunc($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
```

4.4 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.

4.5 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]
```

4.6 DEBUGGING

Before version 5.002, the standard Perl debugger didn't do a very nice job of printing out complex data structures. With 5.002 or above, the debugger includes several new features, including command line editing as well as the `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'
```

4.7 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.

4.8 ARRAYS OF ARRAYS

4.8.1 Declaration of an ARRAY OF ARRAYS

```
@AoA = (
       [ "fred", "barney" ],
       [ "george", "jane", "elroy" ],
       [ "homer", "marge", "bart" ],
     );
```

4.8.2 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";
```

4.8.3 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";
    }
}
```

4.9 HASHES OF ARRAYS

4.9.1 Declaration of a HASH OF ARRAYS

```
%HoA = (
       flintstones        => [ "fred", "barney" ],
       jetsons            => [ "george", "jane", "elroy" ],
       simpsons           => [ "homer", "marge", "bart" ],
     );
```

4.9.2 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";
```

4.9.3 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";
}
```

4.10 ARRAYS OF HASHES

4.10.1 Declaration of an ARRAY OF HASHES

```
@AoH = (
       {
           Lead     => "fred",
           Friend   => "barney",
       },
       {
           Lead     => "george",
           Wife     => "jane",
           Son      => "elroy",
       },
       {
           Lead     => "homer",
           Wife     => "marge",
           Son      => "bart",
       }
 );
```

4.10.2 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";
```

4.10.3 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";
    }
}
```

4.11 HASHES OF HASHES

4.11.1 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",
       },
);
```

4.11.2 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};
}
```

4.11.3 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";
}
```

4.12 MORE ELABORATE RECORDS

4.12.1 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");
```

4.12.2 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, },
       ],
    },
 );
```

4.12.3 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";
    }
```

4.13 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.

4.14 SEE ALSO

perlref(1), perllol(1), perldata(1), perlobj(1)

4.15 AUTHOR

Tom Christiansen <*tchrist@perl.com*>

Last update: Wed Oct 23 04:57:50 MET DST 1996

Chapter 5

perllol

Manipulating Arrays of Arrays in Perl

5.1 DESCRIPTION

5.1.1 Declaration and Access of Arrays of Arrays

The simplest thing to build is an array of arrays (sometimes imprecisely 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:

```
    # assign to our array, an array of array references
    @AoA = (
           [ "fred", "barney" ],
           [ "george", "jane", "elroy" ],
           [ "homer", "marge", "bart" ],
    );

    print $AoA[2][2];
  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", ],
        [ "homer", "bart", "marge", "maggie", ],
        [ "george", "jane", "elroy", "judy", ],
    ];

    print $ref_to_AoA->[2][2];
```

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.

5.1.2 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 very important that you make sure to use the `[]` array reference constructor. That's because this will be very wrong:

```
$AoA[$i] = @tmp;
```

You see, assigning a named array like that to a scalar just counts the number of elements in @tmp, which probably isn't what you want.

If you are running under `use strict`, 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";
```

Notice that I *couldn't* say just:

```
    push $AoA[0], "wilma", "betty";  # WRONG!
```

In fact, that wouldn't even compile. How come? Because the argument to push() must be a real array, not just a reference to such.

5.1.3 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 $aref ( @AoA ) {
        print "\t [ @$aref ],\n";
    }
```

If you wanted to keep track of subscripts, you might do this:

```
    for $i ( 0 .. $#AoA ) {
        print "\t elt $i is [ @{$AoA[$i]} ],\n";
    }
```

or maybe even this. Notice the inner loop.

```
    for $i ( 0 .. $#AoA ) {
        for $j ( 0 .. $#{$AoA[$i]} ) {
            print "elt $i $j is $AoA[$i][$j]\n";
        }
    }
```

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} ) {
            print "elt $i $j is $AoA[$i][$j]\n";
        }
    }
```

Hmm... that's still a bit ugly. How about this:

```
    for $i ( 0 .. $#AoA ) {
        $aref = $AoA[$i];
        $n = @$aref - 1;
        for $j ( 0 .. $n ) {
            print "elt $i $j is $AoA[$i][$j]\n";
        }
    }
```

5.1.4 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. (Remember, of course, that you can always write a loop to do a slice operation.)

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 ];
```

but as you might well imagine, this is 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 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;
}
```

5.2 SEE ALSO

perldata(1), perlref(1), perldsc(1)

5.3 AUTHOR

Tom Christiansen <*tchrist@perl.com*>

Last update: Thu Jun 4 16:16:23 MDT 1998

Chapter 6

perlrequick

Perl regular expressions quick start

6.1 DESCRIPTION

This page covers the very basics of understanding, creating and using regular expressions ('regexes') in Perl.

6.2 The Guide

6.2.1 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, 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
```

6.2.2 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 ˆ in the first position of a character class denotes a **negated character class**, which matches any character but those in the brackets. Both `[...]` and `[ˆ...]` 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:

- \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.

6.2.3 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.

6.2.4 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
```

6.2.5 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** `\1`, `\2`, ... Backreferences are matching variables that can be used *inside* a regex:

```
/(\w\w\w)\s\1/; # find sequences like 'the the' in string
```

`$1`, `$2`, ... should only be used outside of a regex, and `\1`, `\2`, ... only inside a regex.

6.2.6 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+\1/;    # 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.

6.2.7 More matching

There are a few more things you might want to know about matching operators. In the code

```
    $pattern = 'Seuss';
    while (<>) {
        print if /$pattern/;
    }
```

perl has to re-evaluate `$pattern` each time through the loop. If `$pattern` won't be changing, use the `//o` modifier, to only perform variable substitutions once. If you don't want any substitutions at all, use the special delimiter `m''`:

```
    @pattern = ('Seuss');
    m/@pattern/; # matches 'Seuss'
    m'@pattern'; # matches the literal string '@pattern'
```

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'
```

6.2.8 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 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.

6.2.9 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.

6.3 BUGS

None.

6.4 SEE ALSO

This is just a quick start guide. For a more in-depth tutorial on regexes, see *perlretut* and for the reference page, see *perlre*.

6.5 AUTHOR AND COPYRIGHT

6.5.1 Acknowledgments

The author would like to thank Mark-Jason Dominus, Tom Christiansen, Ilya Zakharevich, Brad Hughes, and Mike Giroux for all their helpful comments.

Chapter 7

perlretut

Perl regular expressions tutorial

7.1 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 *perlre*. Regular expressions are an integral part of the `m//`, `s///`, `qr//` and `split` operators and so this tutorial also overlaps with Regexp Quote-Like Operators in *perlop* and split in *perlfunc*.

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 in 5.6.0.

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.

7.2 Part 1: The basics

7.2.1 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 `!~` 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., `""` is used as a delimiter, the forward slash `'/'` becomes an ordinary character and can be used in a 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. 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"        =~ /\143\x61\x74/ # matches, 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 ˆ and `$`. The anchor ˆ means match at the beginning of the string and the anchor `$` 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 ˆ 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 ˆ 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 ˆ 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 equivalence `$string eq 'bert'` and it would be more efficient. The `^...$` regexp really becomes useful when we add in the more powerful regexp tools below.

7.2.2 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 not just represent 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. 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'
    /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 away 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 `-]\^$`. `]` is special because it denotes the end of a character class. `$` 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 ˆ in the first position of a character class denotes a **negated character class**, which matches any character but those in the brackets. Both `[...]` and `[ˆ...]` 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 the write multiple times, so in the interest of saving keystrokes and making regexps more readable, Perl has several abbreviations for common character classes:

- \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 [ˆ0-9]
- \S is a negated \s; it represents any non-whitespace character [ˆ\s]
- \W is a negated \w; it represents any non-word character [ˆ\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.'
```

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 `[ˆ\d\w]` as `[\D\W]`; in fact `[ˆ\d\w]` is the same as `[ˆ\w]`, which is the same as `[\W]`. Think DeMorgan's laws.

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.

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 as empty. Then

```
""   =~ /^$/;    # matches
"\n" =~ /^$/;    # matches, "\n" is ignored

""   =~ /./;      # 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, ignores the "\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 ˆ and `$` to anchor at the beginning and end of lines within the string, rather than just the beginning and end of the string. Perl allows us to choose between ignoring and paying attention to newlines by using the `//s` and `//m` modifiers. `//s` and `//m` stand for single line and multi-line and they determine whether a string is to be treated as one continuous string, or as a set of lines. The two modifiers affect two aspects of how the regexp is interpreted: 1) how the `'.'` character class is defined, and 2) where the anchors ˆ and `$` are able to match. Here are the four possible combinations:

- no modifiers (//): Default behavior. `'.'` matches any character except `"\n"`. ˆ matches only at the beginning of the string and `$` matches only at the end or before a newline at the end.
- s modifier (//s): Treat string as a single long line. `'.'` matches any character, even `"\n"`. ˆ matches only at the beginning of the string and `$` 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"`. ˆ and `$` are able to match at the start or end of *any* line within the string.
- both s and m modifiers (//sm): Treat string as a single long line, but detect multiple lines. `'.'` matches any character, even `"\n"`. ˆ and `$`, 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 want, 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 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.

7.2.3 Matching this or that

Sometimes we would like to 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.

7.2.4 Grouping things and hierarchical matching

Alternation allows a regexp to choose among alternatives, but by itself it 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 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)/;
```

1. Start with the first letter in the string 'a'.
2. Try the first alternative in the first group 'abd'.
3. Match 'a' followed by 'b'. So far so good.
4. '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'.
5. Match 'a' followed by 'b' followed by 'c'. We are on a roll and have satisfied the first group. Set $1 to 'abc'.
6. Move on to the second group and pick the first alternative 'df'.
7. Match the 'd'.
8. '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'.
9. 'd' matches. The second grouping is satisfied, so set $2 to 'd'.
10. 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, during compilation stage, 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.

7.2.5 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. For example, here is a complex regexp and the matching variables indicated below it:

```
    /(ab(cd|ef)((gi)|j))/;
     1  2      34
```

so that if the regexp matched, e.g., `$2` would contain 'cd' or 'ef'. For convenience, perl sets `$+` 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).

Closely associated with the matching variables `$1`, `$2`, ... are the **backreferences** `\1`, `\2`, ... Backreferences are simply matching variables that can be used *inside* a regexp. This is a really nice feature - what matches later in a regexp can depend on what matched earlier in the regexp. Suppose we wanted to look for doubled words in text, like 'the the'. The following regexp finds all 3-letter doubles with a space in between:

```
    /(\w\w\w)\s\1/;
```

The grouping assigns a value to \1, so that the same 3 letter sequence is used for both parts. Here are some words with repeated parts:

```
    % simple_grep '^(\w\w\w\w|\w\w\w|\w\w|\w)\1$' /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 `\1` to look for a repeat. Although `$1` and `\1` represent the same thing, care should be taken to use matched variables `$1`, `$2`, ... only outside a regexp and backreferences `\1`, `\2`, ... only inside a regexp; not doing so may lead to surprising and/or undefined results.

In addition to what was matched, Perl 5.6.0 also provides the positions of what was matched with 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 $expr (1..$#-) {
        print "Match $expr: '${$expr}' at position ($-[$expr],$+[$expr])\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, `` $` = '' `` because the regexp matched at the first character position in the string and stopped, it never saw the second 'the'. It is important to note that using `` $` `` and `$'` slows down regexp matching quite a bit, and `$&` 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 them, 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] )
```

7.2.6 Matching repetitions

The examples in the previous section display an annoying weakness. We were only matching 3-letter words, or syllables of 4 letters or less. We'd like to be able to match words or syllables 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 determine the number of repeats of 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?` = 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+\1/;    # 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 produces $1 and the other does not.

% simple_grep '^(\w+)\1$' /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 matches)
```

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 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??` = match 'a' 0 or 1 times. Try 0 first, then 1.
- `a*?` = match 'a' 0 or more times, i.e., any number of times, but as few times as possible
- `a+?` = match 'a' 1 or more times, i.e., at least once, but as few times as possible
- `a{n,m}?` = match at least `n` times, not more than `m` times, as few times as possible
- `a{n,}?` = match at least `n` times, but as few times as possible
- `a{n}?` = 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 ˆ 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)
```

1. Start with the first letter in the string 't'.
2. The first quantifier '.*' starts out by matching the whole string 'the cat in the hat'.
3. 'a' in the regexp element 'at' doesn't match the end of the string. Backtrack one character.
4. 'a' in the regexp element 'at' still doesn't match the last letter of the string 't', so backtrack one more character.
5. Now we can match the 'a' and the 't'.
6. 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.
7. 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 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.

7.2.7 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 sans 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/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 a little computer language that specifies patterns.

7.2.8 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. First, we pointed out earlier that variables in regexps are substituted before the regexp is evaluated:

```
    $pattern = 'Seuss';
    while (<>) {
        print if /$pattern/;
    }
```

This will print any lines containing the word `Seuss`. It is not as efficient as it could be, however, because perl has to re-evaluate `$pattern` each time through the loop. If `$pattern` won't be changing over the lifetime of the script, we can add the `//o` modifier, which directs perl to only perform variable substitutions once:

```
    #!/usr/bin/perl
    #    Improved simple_grep
    $regexp = shift;
    while (<>) {
        print if /$regexp/o;  # a good deal faster
    }
```

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'
    }
```

`m''` acts like single quotes on a regexp; all other `m` delimiters act like double 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
```

The final two modifiers `//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,
                            # $word[0] = 'cat'
                            # $word[1] = 'dog'
                            # $word[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.

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/go;
        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 and the `s///o` modifier to compile the regexp only once. As with `simple_grep`, both the `print` and the `s/$regexp/$replacement/go` use `$_` implicitly.

A modifier available specifically to search and replace is the `s///e` evaluation modifier. `s///e` wraps an `eval{...}` around the replacement string and the evaluated result 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 substitutions. `s///` in list context returns the same thing as in scalar context, i.e., the number of matches.

The split operator

The **`split`** function can also optionally use a matching operator `m//` to split a string. `split /regexp/, string, limit` splits `string` into a list of substrings and returns that list. The regexp is used to match the character sequence that the `string` is split with respect to. 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 list produced 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.

7.3 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 new features.

7.3.1 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. `\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.',
```

`\L` and `\U` converts a whole substring, delimited by `\L` or `\U` and `\E`, to lower or upper case:

```
    $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.

With the advent of 5.6.0, perl regexps can handle more than just the standard ASCII character set. Perl now supports **Unicode**, a standard for encoding the character sets from many of the world's written languages. Unicode does this by allowing characters to be more than one byte wide. Perl uses the UTF-8 encoding, in which ASCII characters are still encoded as one byte, but characters greater than `chr(127)` may be stored as two or more bytes.

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) when a matching operation will treat the string to be searched as a sequence of bytes (the old way) or as a sequence of Unicode characters (the new way). The answer to 1) is that Unicode characters greater than `chr(127)` may be represented using the `\x{hex}` notation, with `hex` a hexadecimal integer:

```
    /\x{263a}/;  # match a Unicode smiley face :)
```

Unicode characters in the range of 128-255 use two hexadecimal digits with braces: `\x{ab}`. Note that this is different than `\xab`, which is just a hexadecimal byte with no Unicode significance.

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

```
    use charnames ":full"; # use named chars with Unicode full names
    $x = "abc\N{MERCURY}def";
    $x =~ /\N{MERCURY}/;   # matches
```

One can also use short names or restrict names to a certain alphabet:

```
    use charnames ':full';
    print "\N{GREEK SMALL LETTER SIGMA} is called sigma.\n";

    use charnames ":short";
    print "\N{greek:Sigma} is an upper-case sigma.\n";

    use charnames qw(greek);
    print "\N{sigma} is Greek sigma\n";
```

A list of full names is found in the file Names.txt in the lib/perl5/5.X.X/unicore directory.

The answer to requirement 2), as of 5.6.0, is that if a regexp contains Unicode characters, the string is searched as a sequence of Unicode characters. Otherwise, the string is searched as a sequence of bytes. If the string is being searched as a sequence of Unicode characters, but matching a single byte is required, we can use the `\C` escape sequence. `\C` is a character class akin to `.` except that it matches *any* byte 0-255. So

```
    use charnames ":full"; # use named chars with Unicode full names
    $x = "a";
    $x =~ /\C/;  # matches 'a', eats one byte
    $x = "";
    $x =~ /\C/;  # doesn't match, no bytes to match
    $x = "\N{MERCURY}";  # two-byte Unicode character
    $x =~ /\C/;  # matches, but dangerous!
```

The last regexp matches, but is dangerous because the string *character* position is no longer synchronized to the string *byte* position. This generates the warning 'Malformed UTF-8 character'. The `\C` is best used for matching the binary data in strings with binary data intermixed with Unicode characters.

Let us now discuss the rest of the character classes. Just as with Unicode characters, there are named Unicode character classes represented by the `\p{name}` escape sequence. Closely associated is the `\P{name}` character class, which is the negation of the `\p{name}` class. For example, to match lower and uppercase characters,

```
    use charnames ":full"; # use named chars with Unicode full names
    $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
```

Here is the association between some Perl named classes and the traditional Unicode classes:

```
    Perl class name  Unicode class name or regular expression

    IsAlpha          /^[LM]/
    IsAlnum          /^[LMN]/
    IsASCII          $code <= 127
    IsCntrl          /^C/
    IsBlank          $code =~ /^(0020|0009)$/ || /^Z[^lp]/
    IsDigit          Nd
    IsGraph          /^([LMNPS]|Co)/
    IsLower          Ll
    IsPrint         /^([LMNPS]|Co|Zs)/
    IsPunct         /^P/
    IsSpace         /^Z/ || ($code =~ /^(0009|000A|000B|000C|000D)$/
    IsSpacePerl     /^Z/ || ($code =~ /^(0009|000A|000C|000D|0085|2028|2029)$/
    IsUpper         /^L[ut]/
    IsWord          /^[LMN]/ || $code eq "005F"
    IsXDigit        $code =~ /^00(3[0-9]|[46][1-6])$/
```

You can also use the official Unicode class names with the `\p` and `\P`, like `\p{L}` for Unicode 'letters', or `\p{Lu}` for uppercase letters, or `\P{Nd}` for non-digits. If a `name` is just one letter, the braces can be dropped. For instance, `\pM` is the character class of Unicode 'marks', for example accent marks. For the full list see *perlunicode*.

The Unicode has also been separated into various sets of charaters which you can test with `\p{In...}` (in) and `\P{In...}` (not in), for example `\p{Latin}`, `\p{Greek}`, or `\P{Katakana}`. For the full list see *perlunicode*.

`\X` is an abbreviation for a character class sequence that includes the Unicode 'combining character sequences'. A 'combining character sequence' is a base character followed by any number of combining characters. An example of a combining character is an accent. Using the Unicode full names, e.g., `A + COMBINING RING` is a combining character sequence with base character `A` and combining character `COMBINING RING` , which translates in Danish to A with the circle atop it, as in the word Angstrom. `\X` is equivalent to `\PM\pM*}`, i.e., a non-mark followed by one or more marks.

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). If `utf8` is being used, then these classes are defined the same as their corresponding perl Unicode classes: `[:upper:]` is the same as `\p{IsUpper}`, etc. The POSIX character classes, however, don't require using `utf8`. The `[:digit:]`, `[:word:]`, and `[:space:]` correspond to the familiar `\d`, `\w`, and `\s` character classes. To negate a POSIX class, put a ˆ in front of the name, so that, e.g., `[:ˆdigit:]` corresponds to `\D` and under `utf8`, `\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
    use charnames ":full";
    /\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.

7.3.2 Compiling and saving regular expressions

In Part 1 we discussed the `//o` modifier, which compiles a regexp just once. This suggests that a compiled regexp is some 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{}` and `qr˜˜`. The single quote delimiters `qr”` prevent any interpolation from taking place.

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, `simple_grep` program can be expanded into 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..$number-1);
    @compiled = map qr/$_/, @regexp;
    while ($line = <>) {
        foreach $pattern (@compiled) {
            if ($line =~ /$pattern/) {
                print $line;
                last;  # we matched, so move onto the next line
            }
        }
    }
    ^D

    % multi_grep 2 last for multi_grep
        $regexp[$_] = shift foreach (0..$number-1);
            foreach $pattern (@compiled) {
                    last;
```

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.

7.3.3 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,}?`. The rest 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.

The modifiers `//i`, `//m`, `//s`, and `//x` can also 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 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.

7.3.4 Non-capturing groupings

We noted in Part 1 that groupings `()` had two distinct functions: 1) group regexp elements together as a single unit, and 2) extract, or capture, substrings that matched the regexp in the grouping. Non-capturing groupings, denoted by `(?:regexp)`, allow the regexp to be treated as a single unit, but don't extract substrings or set matching variables `$1`, etc. 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:

```
    $x = '12a34b5';
    @num = split /(a|b)/, $x;    # @num = ('12','a','34','b','5')
    @num = split /(?:a|b)/, $x;  # @num = ('12','34','5')
```

Non-capturing groupings may also have embedded modifiers: `(?i-m:regexp)` is a non-capturing grouping that matches `regexp` case insensitively and turns off multi-line mode.

7.3.5 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 ˆ matches the beginning of the line, but doesn't eat any characters. Similarly, the word boundary anchor `\b` matches, e.g., if the character to the left is a word character and the character to the right is a non-word character, 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. ˆ 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.

7.3.6 Using independent subexpressions to prevent backtracking

The last few extended patterns in this tutorial are experimental as of 5.6.0. Play with them, use them in some code, but don't rely on them just yet for production code.

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.

7.3.7 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 two 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 second form is a bare zero width assertion `(?...)`, either a lookahead, a lookbehind, or a code assertion (discussed in the next section).

The integer 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)\2\1|\1)$' /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.

7.3.8 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.

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!";})[d]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 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 REx 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 than it is. 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 §7.3.9 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 the article 'the' in either English or 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'
                       (die|das|der)     # else, match 'die|das|der'
                    )
                   /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 with interpolating variables, perl may surprise you:

```
    $bar = 5;
    $pat = '(?{ 1 })';
    /foo(?{ $bar })bar/; # compiles ok, $bar not interpolated
    /foo(?{ 1 })$bar/;   # compile error!
    /foo${pat}bar/;      # compile error!

    $pat = qr/(?{ $foo = 1 })/;  # precompile code regexp
    /foo${pat}bar/;      # compiles ok
```

If a regexp has (1) code expressions and interpolating variables,or (2) 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;
    $pat = '(?{ 1 })';
    /foo(?{ 1 })$bar/;   # compiles ok
    /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 if a binary string `1101010010001...` has a Fibonacci spacing 0,1,1,2,3,5,... of the 1's:

```
    $s0 = 0; $s1 = 1; # initial conditions
    $x = "1101010010001000001";
    print "It is a Fibonacci sequence\n"
        if $x =~ /^1         # match an initial '1'
                    (
                       (??{'0' x $s0}) # match $s0 of '0'
                       1               # and then a '1'
                       (?{
                          $largest = $s0;   # largest seq so far
                          $s2 = $s1 + $s0;  # compute next term
                          $s0 = $s1;        # in Fibonacci sequence
                          $s1 = $s2;
                         })
                    )+   # repeat as needed
                  $      # that is all there is
                 /x;
    print "Largest sequence matched was $largest\n";
```

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 `$s0` and `$s1` are not substituted when the regexp is compiled, as happens for ordinary variables outside a code expression. Rather, the code expressions are evaluated when perl encounters them during the search for a match.

The regexp without the `//x` modifier is

```
    /^1((??{'0'x$s0})1(?{$largest=$s0;$s2=$s1+$s0$s0=$s1;$s1=$s2;}))+$/;
```

and is a great start on an Obfuscated Perl entry :-) When working with code and conditional expressions, the extended form of regexps is almost necessary in creating and debugging regexps.

7.3.9 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 '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 n the compilation list above. See Debugging regular expressions in *perldebguts* 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
```

7.4 BUGS

Code expressions, conditional expressions, and independent expressions are **experimental**. Don’t use them in production code. Yet.

7.5 SEE ALSO

This is just a tutorial. For the full story on perl regular expressions, see the *perlre* regular expressions reference page.

For more information on the matching `m//` and substitution `s///` operators, see Regexp Quote-Like Operators in *perlop*. For information on the `split` operation, see split in *perlfunc*.

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).

7.6 AUTHOR AND COPYRIGHT

7.6.1 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.

Chapter 8

perlboot

Beginner's Object-Oriented Tutorial

8.1 DESCRIPTION

If you're not familiar with objects from other languages, some of the other Perl object documentation may be a little daunting, such as *perlobj*, a basic reference in using objects, and *perltoot*, which introduces readers to the peculiarities of Perl's object system in a tutorial way.

So, let's take a different approach, presuming no prior object experience. It helps if you know about subroutines (*perlsub*), references (*perlref* et. seq.), and packages (*perlmod*), so become familiar with those first if you haven't already.

8.1.1 If we could talk to the animals...

Let's let the animals talk for a moment:

```
    sub Cow::speak {
      print "a Cow goes moooo!\n";
    }
    sub Horse::speak {
      print "a Horse goes neigh!\n";
    }
    sub Sheep::speak {
      print "a Sheep goes baaaah!\n"
    }

    Cow::speak;
    Horse::speak;
    Sheep::speak;
```

This results in:

```
    a Cow goes moooo!
    a Horse goes neigh!
    a Sheep goes baaaah!
```

Nothing spectacular here. Simple subroutines, albeit from separate packages, and called using the full package name. So let's create an entire pasture:

```
# Cow::speak, Horse::speak, Sheep::speak as before
@pasture = qw(Cow Cow Horse Sheep Sheep);
foreach $animal (@pasture) {
  &{$animal."::speak"};
}
```

This results in:

```
a Cow goes moooo!
a Cow goes moooo!
a Horse goes neigh!
a Sheep goes baaaah!
a Sheep goes baaaah!
```

Wow. That symbolic coderef de-referencing there is pretty nasty. We're counting on `no strict subs` mode, certainly not recommended for larger programs. And why was that necessary? Because the name of the package seems to be inseparable from the name of the subroutine we want to invoke within that package.

Or is it?

8.1.2 Introducing the method invocation arrow

For now, let's say that `Class->method` invokes subroutine `method` in package `Class`. (Here, "Class" is used in its "category" meaning, not its "scholastic" meaning.) That's not completely accurate, but we'll do this one step at a time. Now let's use it like so:

```
# Cow::speak, Horse::speak, Sheep::speak as before
Cow->speak;
Horse->speak;
Sheep->speak;
```

And once again, this results in:

```
a Cow goes moooo!
a Horse goes neigh!
a Sheep goes baaaah!
```

That's not fun yet. Same number of characters, all constant, no variables. But yet, the parts are separable now. Watch:

```
$a = "Cow";
$a->speak; # invokes Cow->speak
```

Ahh! Now that the package name has been parted from the subroutine name, we can use a variable package name. And this time, we've got something that works even when `use strict refs` is enabled.

8.1.3 Invoking a barnyard

Let's take that new arrow invocation and put it back in the barnyard example:

```
sub Cow::speak {
  print "a Cow goes moooo!\n";
}
sub Horse::speak {
  print "a Horse goes neigh!\n";
}
sub Sheep::speak {
  print "a Sheep goes baaaah!\n"
}
```

```
@pasture = qw(Cow Cow Horse Sheep Sheep);
foreach $animal (@pasture) {
  $animal->speak;
}
```

There! Now we have the animals all talking, and safely at that, without the use of symbolic coderefs.

But look at all that common code. Each of the `speak` routines has a similar structure: a `print` operator and a string that contains common text, except for two of the words. It'd be nice if we could factor out the commonality, in case we decide later to change it all to `says` instead of `goes`.

And we actually have a way of doing that without much fuss, but we have to hear a bit more about what the method invocation arrow is actually doing for us.

8.1.4 The extra parameter of method invocation

The invocation of:

```
Class->method(@args)
```

attempts to invoke subroutine `Class::method` as:

```
Class::method("Class", @args);
```

(If the subroutine can't be found, "inheritance" kicks in, but we'll get to that later.) This means that we get the class name as the first parameter (the only parameter, if no arguments are given). So we can rewrite the `Sheep` speaking subroutine as:

```
sub Sheep::speak {
  my $class = shift;
  print "a $class goes baaaah!\n";
}
```

And the other two animals come out similarly:

```
sub Cow::speak {
  my $class = shift;
  print "a $class goes moooo!\n";
}
sub Horse::speak {
  my $class = shift;
  print "a $class goes neigh!\n";
}
```

In each case, `$class` will get the value appropriate for that subroutine. But once again, we have a lot of similar structure. Can we factor that out even further? Yes, by calling another method in the same class.

8.1.5 Calling a second method to simplify things

Let's call out from `speak` to a helper method called `sound`. This method provides the constant text for the sound itself.

```
{ package Cow;
  sub sound { "moooo" }
  sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n"
  }
}
```

Now, when we call `Cow->speak`, we get a `$class` of `Cow` in `speak`. This in turn selects the `Cow->sound` method, which returns `moooo`. But how different would this be for the `Horse`?

```
    { package Horse;
      sub sound { "neigh" }
      sub speak {
        my $class = shift;
        print "a $class goes ", $class->sound, "!\n"
      }
    }
```

Only the name of the package and the specific sound change. So can we somehow share the definition for `speak` between the Cow and the Horse? Yes, with inheritance!

8.1.6 Inheriting the windpipes

We'll define a common subroutine package called `Animal`, with the definition for `speak`:

```
    { package Animal;
      sub speak {
        my $class = shift;
        print "a $class goes ", $class->sound, "!\n"
      }
    }
```

Then, for each animal, we say it "inherits" from `Animal`, along with the animal-specific sound:

```
    { package Cow;
      @ISA = qw(Animal);
      sub sound { "moooo" }
    }
```

Note the added `@ISA` array. We'll get to that in a minute.

But what happens when we invoke `Cow->speak` now?

First, Perl constructs the argument list. In this case, it's just `Cow`. Then Perl looks for `Cow::speak`. But that's not there, so Perl checks for the inheritance array `@Cow::ISA`. It's there, and contains the single name `Animal`.

Perl next checks for `speak` inside `Animal` instead, as in `Animal::speak`. And that's found, so Perl invokes that subroutine with the already frozen argument list.

Inside the `Animal::speak` subroutine, `$class` becomes `Cow` (the first argument). So when we get to the step of invoking `$class->sound`, it'll be looking for `Cow->sound`, which gets it on the first try without looking at `@ISA`. Success!

8.1.7 A few notes about @ISA

This magical `@ISA` variable (pronounced "is a" not "ice-uh"), has declared that `Cow` "is a" `Animal`. Note that it's an array, not a simple single value, because on rare occasions, it makes sense to have more than one parent class searched for the missing methods.

If `Animal` also had an `@ISA`, then we'd check there too. The search is recursive, depth-first, left-to-right in each `@ISA`. Typically, each `@ISA` has only one element (multiple elements means multiple inheritance and multiple headaches), so we get a nice tree of inheritance.

When we turn on `use strict`, we'll get complaints on `@ISA`, since it's not a variable containing an explicit package name, nor is it a lexical ("my") variable. We can't make it a lexical variable though (it has to belong to the package to be found by the inheritance mechanism), so there's a couple of straightforward ways to handle that.

The easiest is to just spell the package name out:

```
    @Cow::ISA = qw(Animal);
```

Or allow it as an implicitly named package variable:

```
    package Cow;
    use vars qw(@ISA);
    @ISA = qw(Animal);
```

If you're bringing in the class from outside, via an object-oriented module, you change:

```
    package Cow;
    use Animal;
    use vars qw(@ISA);
    @ISA = qw(Animal);
```

into just:

```
    package Cow;
    use base qw(Animal);
```

And that's pretty darn compact.

8.1.8 Overriding the methods

Let's add a mouse, which can barely be heard:

```
    # Animal package from before
    { package Mouse;
      @ISA = qw(Animal);
      sub sound { "squeak" }
      sub speak {
        my $class = shift;
        print "a $class goes ", $class->sound, "!\n";
        print "[but you can barely hear it!]\n";
      }
    }

    Mouse->speak;
```

which results in:

```
    a Mouse goes squeak!
    [but you can barely hear it!]
```

Here, `Mouse` has its own speaking routine, so `Mouse->speak` doesn't immediately invoke `Animal->speak`. This is known as "overriding". In fact, we didn't even need to say that a `Mouse` was an `Animal` at all, since all of the methods needed for `speak` are completely defined with `Mouse`.

But we've now duplicated some of the code from `Animal->speak`, and this can once again be a maintenance headache. So, can we avoid that? Can we say somehow that a `Mouse` does everything any other `Animal` does, but add in the extra comment? Sure!

First, we can invoke the `Animal::speak` method directly:

```
# Animal package from before
{ package Mouse;
  @ISA = qw(Animal);
  sub sound { "squeak" }
  sub speak {
    my $class = shift;
    Animal::speak($class);
    print "[but you can barely hear it!]\n";
  }
}
```

Note that we have to include the `$class` parameter (almost surely the value of `"Mouse"`) as the first parameter to `Animal::speak`, since we've stopped using the method arrow. Why did we stop? Well, if we invoke `Animal->speak` there, the first parameter to the method will be `"Animal"` not `"Mouse"`, and when time comes for it to call for the `sound`, it won't have the right class to come back to this package.

Invoking `Animal::speak` directly is a mess, however. What if `Animal::speak` didn't exist before, and was being inherited from a class mentioned in `@Animal::ISA`? Because we are no longer using the method arrow, we get one and only one chance to hit the right subroutine.

Also note that the `Animal` classname is now hardwired into the subroutine selection. This is a mess if someone maintains the code, changing `@ISA` for <Mouse> and didn't notice `Animal` there in `speak`. So, this is probably not the right way to go.

8.1.9 Starting the search from a different place

A better solution is to tell Perl to search from a higher place in the inheritance chain:

```
# same Animal as before
{ package Mouse;
  # same @ISA, &sound as before
  sub speak {
    my $class = shift;
    $class->Animal::speak;
    print "[but you can barely hear it!]\n";
  }
}
```

Ahh. This works. Using this syntax, we start with `Animal` to find `speak`, and use all of `Animal`'s inheritance chain if not found immediately. And yet the first parameter will be `$class`, so the found `speak` method will get `Mouse` as its first entry, and eventually work its way back to `Mouse::sound` for the details.

But this isn't the best solution. We still have to keep the `@ISA` and the initial search package coordinated. Worse, if `Mouse` had multiple entries in `@ISA`, we wouldn't necessarily know which one had actually defined `speak`. So, is there an even better way?

8.1.10 The SUPER way of doing things

By changing the `Animal` class to the `SUPER` class in that invocation, we get a search of all of our super classes (classes listed in `@ISA`) automatically:

```
# same Animal as before
{ package Mouse;
  # same @ISA, &sound as before
  sub speak {
    my $class = shift;
    $class->SUPER::speak;
    print "[but you can barely hear it!]\n";
  }
}
```

So, `SUPER::speak` means look in the current package's `@ISA` for `speak`, invoking the first one found. Note that it does *not* look in the `@ISA` of `$class`.

8.1.11 Where we're at so far...

So far, we've seen the method arrow syntax:

```
  Class->method(@args);
```

or the equivalent:

```
  $a = "Class";
  $a->method(@args);
```

which constructs an argument list of:

```
  ("Class", @args)
```

and attempts to invoke

```
  Class::method("Class", @Args);
```

However, if `Class::method` is not found, then `@Class::ISA` is examined (recursively) to locate a package that does indeed contain `method`, and that subroutine is invoked instead.

Using this simple syntax, we have class methods, (multiple) inheritance, overriding, and extending. Using just what we've seen so far, we've been able to factor out common code, and provide a nice way to reuse implementations with variations. This is at the core of what objects provide, but objects also provide instance data, which we haven't even begun to cover.

8.1.12 A horse is a horse, of course of course – or is it?

Let's start with the code for the `Animal` class and the `Horse` class:

```
  { package Animal;
    sub speak {
      my $class = shift;
      print "a $class goes ", $class->sound, "!\n"
    }
  }
  { package Horse;
    @ISA = qw(Animal);
    sub sound { "neigh" }
  }
```

This lets us invoke `Horse->speak` to ripple upward to `Animal::speak`, calling back to `Horse::sound` to get the specific sound, and the output of:

```
  a Horse goes neigh!
```

But all of our Horse objects would have to be absolutely identical. If I add a subroutine, all horses automatically share it. That's great for making horses the same, but how do we capture the distinctions about an individual horse? For example, suppose I want to give my first horse a name. There's got to be a way to keep its name separate from the other horses.

We can do that by drawing a new distinction, called an "instance". An "instance" is generally created by a class. In Perl, any reference can be an instance, so let's start with the simplest reference that can hold a horse's name: a scalar reference.

```
my $name = "Mr. Ed";
my $talking = \$name;
```

So now `$talking` is a reference to what will be the instance-specific data (the name). The final step in turning this into a real instance is with a special operator called `bless`:

```
bless $talking, Horse;
```

This operator stores information about the package named `Horse` into the thing pointed at by the reference. At this point, we say `$talking` is an instance of `Horse`. That is, it's a specific horse. The reference is otherwise unchanged, and can still be used with traditional dereferencing operators.

8.1.13 Invoking an instance method

The method arrow can be used on instances, as well as names of packages (classes). So, let's get the sound that `$talking` makes:

```
my $noise = $talking->sound;
```

To invoke `sound`, Perl first notes that `$talking` is a blessed reference (and thus an instance). It then constructs an argument list, in this case from just `($talking)`. (Later we'll see that arguments will take their place following the instance variable, just like with classes.)

Now for the fun part: Perl takes the class in which the instance was blessed, in this case `Horse`, and uses that to locate the subroutine to invoke the method. In this case, `Horse::sound` is found directly (without using inheritance), yielding the final subroutine invocation:

```
Horse::sound($talking)
```

Note that the first parameter here is still the instance, not the name of the class as before. We'll get `neigh` as the return value, and that'll end up as the `$noise` variable above.

If Horse::sound had not been found, we'd be wandering up the `@Horse::ISA` list to try to find the method in one of the superclasses, just as for a class method. The only difference between a class method and an instance method is whether the first parameter is an instance (a blessed reference) or a class name (a string).

8.1.14 Accessing the instance data

Because we get the instance as the first parameter, we can now access the instance-specific data. In this case, let's add a way to get at the name:

```
{ package Horse;
  @ISA = qw(Animal);
  sub sound { "neigh" }
  sub name {
    my $self = shift;
    $$self;
  }
}
```

Now we call for the name:

```
print $talking->name, " says ", $talking->sound, "\n";
```

Inside `Horse::name`, the `@_` array contains just `$talking`, which the `shift` stores into `$self`. (It's traditional to shift the first parameter off into a variable named `$self` for instance methods, so stay with that unless you have strong reasons otherwise.) Then, `$self` gets de-referenced as a scalar ref, yielding `Mr.  Ed`, and we're done with that. The result is:

```
Mr. Ed says neigh.
```

8.1.15 How to build a horse

Of course, if we constructed all of our horses by hand, we'd most likely make mistakes from time to time. We're also violating one of the properties of object-oriented programming, in that the "inside guts" of a Horse are visible. That's good if you're a veterinarian, but not if you just like to own horses. So, let's let the Horse class build a new horse:

```
{ package Horse;
  @ISA = qw(Animal);
  sub sound { "neigh" }
  sub name {
    my $self = shift;
    $$self;
  }
  sub named {
    my $class = shift;
    my $name = shift;
    bless \$name, $class;
  }
}
```

Now with the new `named` method, we can build a horse:

```
my $talking = Horse->named("Mr. Ed");
```

Notice we're back to a class method, so the two arguments to `Horse::named` are `Horse` and `Mr. Ed`. The `bless` operator not only blesses `$name`, it also returns the reference to `$name`, so that's fine as a return value. And that's how to build a horse.

We've called the constructor `named` here, so that it quickly denotes the constructor's argument as the name for this particular `Horse`. You can use different constructors with different names for different ways of "giving birth" to the object (like maybe recording its pedigree or date of birth). However, you'll find that most people coming to Perl from more limited languages use a single constructor named `new`, with various ways of interpreting the arguments to `new`. Either style is fine, as long as you document your particular way of giving birth to an object. (And you *were* going to do that, right?)

8.1.16 Inheriting the constructor

But was there anything specific to `Horse` in that method? No. Therefore, it's also the same recipe for building anything else that inherited from `Animal`, so let's put it there:

```
{ package Animal;
  sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n"
  }
  sub name {
    my $self = shift;
    $$self;
  }
  sub named {
    my $class = shift;
    my $name = shift;
    bless \$name, $class;
  }
}
{ package Horse;
  @ISA = qw(Animal);
  sub sound { "neigh" }
}
```

Ahh, but what happens if we invoke `speak` on an instance?

```
  my $talking = Horse->named("Mr. Ed");
  $talking->speak;
```

We get a debugging value:

```
  a Horse=SCALAR(0xaca42ac) goes neigh!
```

Why? Because the `Animal::speak` routine is expecting a classname as its first parameter, not an instance. When the instance is passed in, we'll end up using a blessed scalar reference as a string, and that shows up as we saw it just now.

8.1.17 Making a method work with either classes or instances

All we need is for a method to detect if it is being called on a class or called on an instance. The most straightforward way is with the `ref` operator. This returns a string (the classname) when used on a blessed reference, and `undef` when used on a string (like a classname). Let's modify the `name` method first to notice the change:

```
  sub name {
    my $either = shift;
    ref $either
      ? $$either # it's an instance, return name
      : "an unnamed $either"; # it's a class, return generic
  }
```

Here, the `?:` operator comes in handy to select either the dereference or a derived string. Now we can use this with either an instance or a class. Note that I've changed the first parameter holder to `$either` to show that this is intended:

```
  my $talking = Horse->named("Mr. Ed");
  print Horse->name, "\n"; # prints "an unnamed Horse\n"
  print $talking->name, "\n"; # prints "Mr Ed.\n"
```

and now we'll fix `speak` to use this:

```
  sub speak {
    my $either = shift;
    print $either->name, " goes ", $either->sound, "\n";
  }
```

And since `sound` already worked with either a class or an instance, we're done!

8.1.18 Adding parameters to a method

Let's train our animals to eat:

```
  { package Animal;
    sub named {
      my $class = shift;
      my $name = shift;
      bless \$name, $class;
    }
    sub name {
      my $either = shift;
      ref $either
```

```
        ? $$either # it’s an instance, return name
        : "an unnamed $either"; # it’s a class, return generic
    }
    sub speak {
      my $either = shift;
      print $either->name, " goes ", $either->sound, "\n";
    }
    sub eat {
      my $either = shift;
      my $food = shift;
      print $either->name, " eats $food.\n";
    }
  }
  { package Horse;
    @ISA = qw(Animal);
    sub sound { "neigh" }
  }
  { package Sheep;
    @ISA = qw(Animal);
    sub sound { "baaaah" }
  }
```

And now try it out:

```
  my $talking = Horse->named("Mr. Ed");
  $talking->eat("hay");
  Sheep->eat("grass");
```

which prints:

```
  Mr. Ed eats hay.
  an unnamed Sheep eats grass.
```

An instance method with parameters gets invoked with the instance, and then the list of parameters. So that first invocation is like:

```
  Animal::eat($talking, "hay");
```

8.1.19 More interesting instances

What if an instance needs more data? Most interesting instances are made of many items, each of which can in turn be a reference or even another object. The easiest way to store these is often in a hash. The keys of the hash serve as the names of parts of the object (often called "instance variables" or "member variables"), and the corresponding values are, well, the values.

But how do we turn the horse into a hash? Recall that an object was any blessed reference. We can just as easily make it a blessed hash reference as a blessed scalar reference, as long as everything that looks at the reference is changed accordingly.

Let’s make a sheep that has a name and a color:

```
  my $bad = bless { Name => "Evil", Color => "black" }, Sheep;
```

so `$bad->{Name}` has `Evil`, and `$bad->{Color}` has `black`. But we want to make `$bad->name` access the name, and that’s now messed up because it’s expecting a scalar reference. Not to worry, because that’s pretty easy to fix up:

```
## in Animal
sub name {
  my $either = shift;
  ref $either ?
    $either->{Name} :
    "an unnamed $either";
}
```

And of course `named` still builds a scalar sheep, so let's fix that as well:

```
## in Animal
sub named {
  my $class = shift;
  my $name = shift;
  my $self = { Name => $name, Color => $class->default_color };
  bless $self, $class;
}
```

What's this `default_color`? Well, if `named` has only the name, we still need to set a color, so we'll have a class-specific initial color. For a sheep, we might define it as white:

```
## in Sheep
sub default_color { "white" }
```

And then to keep from having to define one for each additional class, we'll define a "backstop" method that serves as the "default default", directly in `Animal`:

```
## in Animal
sub default_color { "brown" }
```

Now, because `name` and `named` were the only methods that referenced the "structure" of the object, the rest of the methods can remain the same, so `speak` still works as before.

8.1.20 A horse of a different color

But having all our horses be brown would be boring. So let's add a method or two to get and set the color.

```
## in Animal
sub color {
  $_[0]->{Color}
}
sub set_color {
  $_[0]->{Color} = $_[1];
}
```

Note the alternate way of accessing the arguments: `$_[0]` is used in-place, rather than with a `shift`. (This saves us a bit of time for something that may be invoked frequently.) And now we can fix that color for Mr. Ed:

```
my $talking = Horse->named("Mr. Ed");
$talking->set_color("black-and-white");
print $talking->name, " is colored ", $talking->color, "\n";
```

which results in:

```
Mr. Ed is colored black-and-white
```

8.1.21 Summary

So, now we have class methods, constructors, instance methods, instance data, and even accessors. But that's still just the beginning of what Perl has to offer. We haven't even begun to talk about accessors that double as getters and setters, destructors, indirect object notation, subclasses that add instance data, per-class data, overloading, "isa" and "can" tests, `UNIVERSAL` class, and so on. That's for the rest of the Perl documentation to cover. Hopefully, this gets you started, though.

8.2 SEE ALSO

For more information, see *perlobj* (for all the gritty details about Perl objects, now that you've seen the basics), *perltoot* (the tutorial for those who already know objects), *perltooc* (dealing with class data), *perlbot* (for some more tricks), and books such as Damian Conway's excellent *Object Oriented Perl*.

Some modules which might prove interesting are Class::Accessor, Class::Class, Class::Contract, Class::Data::Inheritable, Class::MethodMaker and Tie::SecureHash

8.3 COPYRIGHT

Copyright (c) 1999, 2000 by Randal L. Schwartz and Stonehenge Consulting Services, Inc. Permission is hereby granted to distribute this document intact with the Perl distribution, and in accordance with the licenses of the Perl distribution; derived documents must include this copyright notice intact.

Portions of this text have been derived from Perl Training materials originally appearing in the *Packages, References, Objects, and Modules* course taught by instructors for Stonehenge Consulting Services, Inc. and used with permission.

Portions of this text have been derived from materials originally appearing in *Linux Magazine* and used with permission.

Chapter 9

perltoot

Tom’s object-oriented tutorial for perl

9.1 DESCRIPTION

Object-oriented programming is a big seller these days. Some managers would rather have objects than sliced bread. Why is that? What’s so special about an object? Just what *is* an object anyway?

An object is nothing but a way of tucking away complex behaviours into a neat little easy-to-use bundle. (This is what professors call abstraction.) Smart people who have nothing to do but sit around for weeks on end figuring out really hard problems make these nifty objects that even regular people can use. (This is what professors call software reuse.) Users (well, programmers) can play with this little bundle all they want, but they aren’t to open it up and mess with the insides. Just like an expensive piece of hardware, the contract says that you void the warranty if you muck with the cover. So don’t do that.

The heart of objects is the class, a protected little private namespace full of data and functions. A class is a set of related routines that addresses some problem area. You can think of it as a user-defined type. The Perl package mechanism, also used for more traditional modules, is used for class modules as well. Objects "live" in a class, meaning that they belong to some package.

More often than not, the class provides the user with little bundles. These bundles are objects. They know whose class they belong to, and how to behave. Users ask the class to do something, like "give me an object." Or they can ask one of these objects to do something. Asking a class to do something for you is calling a *class method*. Asking an object to do something for you is calling an *object method*. Asking either a class (usually) or an object (sometimes) to give you back an object is calling a *constructor*, which is just a kind of method.

That’s all well and good, but how is an object different from any other Perl data type? Just what is an object *really*; that is, what’s its fundamental type? The answer to the first question is easy. An object is different from any other data type in Perl in one and only one way: you may dereference it using not merely string or numeric subscripts as with simple arrays and hashes, but with named subroutine calls. In a word, with *methods*.

The answer to the second question is that it’s a reference, and not just any reference, mind you, but one whose referent has been *bless*()ed into a particular class (read: package). What kind of reference? Well, the answer to that one is a bit less concrete. That’s because in Perl the designer of the class can employ any sort of reference they’d like as the underlying intrinsic data type. It could be a scalar, an array, or a hash reference. It could even be a code reference. But because of its inherent flexibility, an object is usually a hash reference.

9.2 Creating a Class

Before you create a class, you need to decide what to name it. That’s because the class (package) name governs the name of the file used to house it, just as with regular modules. Then, that class (package) should provide one or more ways to generate objects. Finally, it should provide mechanisms to allow users of its objects to indirectly manipulate these objects from a distance.

For example, let's make a simple Person class module. It gets stored in the file Person.pm. If it were called a Happy::Person class, it would be stored in the file Happy/Person.pm, and its package would become Happy::Person instead of just Person. (On a personal computer not running Unix or Plan 9, but something like Mac OS or VMS, the directory separator may be different, but the principle is the same.) Do not assume any formal relationship between modules based on their directory names. This is merely a grouping convenience, and has no effect on inheritance, variable accessibility, or anything else.

For this module we aren't going to use Exporter, because we're a well-behaved class module that doesn't export anything at all. In order to manufacture objects, a class needs to have a *constructor method*. A constructor gives you back not just a regular data type, but a brand-new object in that class. This magic is taken care of by the bless() function, whose sole purpose is to enable its referent to be used as an object. Remember: being an object really means nothing more than that methods may now be called against it.

While a constructor may be named anything you'd like, most Perl programmers seem to like to call theirs new(). However, new() is not a reserved word, and a class is under no obligation to supply such. Some programmers have also been known to use a function with the same name as the class as the constructor.

9.2.1 Object Representation

By far the most common mechanism used in Perl to represent a Pascal record, a C struct, or a C++ class is an anonymous hash. That's because a hash has an arbitrary number of data fields, each conveniently accessed by an arbitrary name of your own devising.

If you were just doing a simple struct-like emulation, you would likely go about it something like this:

```
$rec = {
    name  => "Jason",
    age   => 23,
    peers => [ "Norbert", "Rhys", "Phineas"],
};
```

If you felt like it, you could add a bit of visual distinction by up-casing the hash keys:

```
$rec = {
    NAME  => "Jason",
    AGE   => 23,
    PEERS => [ "Norbert", "Rhys", "Phineas"],
};
```

And so you could get at `$rec->{NAME}` to find "Jason", or `@{ $rec->{PEERS} }` to get at "Norbert", "Rhys", and "Phineas". (Have you ever noticed how many 23-year-old programmers seem to be named "Jason" these days? :-)

This same model is often used for classes, although it is not considered the pinnacle of programming propriety for folks from outside the class to come waltzing into an object, brazenly accessing its data members directly. Generally speaking, an object should be considered an opaque cookie that you use *object methods* to access. Visually, methods look like you're dereffing a reference using a function name instead of brackets or braces.

9.2.2 Class Interface

Some languages provide a formal syntactic interface to a class's methods, but Perl does not. It relies on you to read the documentation of each class. If you try to call an undefined method on an object, Perl won't complain, but the program will trigger an exception while it's running. Likewise, if you call a method expecting a prime number as its argument with a non-prime one instead, you can't expect the compiler to catch this. (Well, you can expect it all you like, but it's not going to happen.)

Let's suppose you have a well-educated user of your Person class, someone who has read the docs that explain the prescribed interface. Here's how they might use the Person class:

```
use Person;
```

```
    $him = Person->new();
    $him->name("Jason");
    $him->age(23);
    $him->peers( "Norbert", "Rhys", "Phineas" );

    push @All_Recs, $him;  # save object in array for later

    printf "%s is %d years old.\n", $him->name, $him->age;
    print "His peers are: ", join(", ", $him->peers), "\n";

    printf "Last rec's name is %s\n", $All_Recs[-1]->name;
```

As you can see, the user of the class doesn't know (or at least, has no business paying attention to the fact) that the object has one particular implementation or another. The interface to the class and its objects is exclusively via methods, and that's all the user of the class should ever play with.

9.2.3 Constructors and Instance Methods

Still, *someone* has to know what's in the object. And that someone is the class. It implements methods that the programmer uses to access the object. Here's how to implement the Person class using the standard hash-ref-as-an-object idiom. We'll make a class method called new() to act as the constructor, and three object methods called name(), age(), and peers() to get at per-object data hidden away in our anonymous hash.

```
    package Person;
    use strict;

    ##################################################
    ## the object constructor (simplistic version)  ##
    ##################################################
    sub new {
        my $self  = {};
        $self->{NAME}   = undef;
        $self->{AGE}    = undef;
        $self->{PEERS}  = [];
        bless($self);           # but see below
        return $self;
    }

    ##############################################
    ## methods to access per-object data        ##
    ##                                          ##
    ## With args, they set the value.  Without  ##
    ## any, they only retrieve it/them.         ##
    ##############################################

    sub name {
        my $self = shift;
        if (@_) { $self->{NAME} = shift }
        return $self->{NAME};
    }

    sub age {
        my $self = shift;
        if (@_) { $self->{AGE} = shift }
        return $self->{AGE};
    }
    sub peers {
        my $self = shift;
        if (@_) { @{ $self->{PEERS} } = @_ }
        return @{ $self->{PEERS} };
    }

    1;  # so the require or use succeeds
```

We've created three methods to access an object's data, name(), age(), and peers(). These are all substantially similar. If called with an argument, they set the appropriate field; otherwise they return the value held by that field, meaning the value of that hash key.

9.2.4 Planning for the Future: Better Constructors

Even though at this point you may not even know what it means, someday you're going to worry about inheritance. (You can safely ignore this for now and worry about it later if you'd like.) To ensure that this all works out smoothly, you must use the double-argument form of bless(). The second argument is the class into which the referent will be blessed. By not assuming our own class as the default second argument and instead using the class passed into us, we make our constructor inheritable.

```
    sub new {
        my $class = shift;
        my $self  = {};
        $self->{NAME}   = undef;
        $self->{AGE}    = undef;
        $self->{PEERS}  = [];
        bless ($self, $class);
        return $self;
    }
```

That's about all there is for constructors. These methods bring objects to life, returning neat little opaque bundles to the user to be used in subsequent method calls.

9.2.5 Destructors

Every story has a beginning and an end. The beginning of the object's story is its constructor, explicitly called when the object comes into existence. But the ending of its story is the *destructor*, a method implicitly called when an object leaves this life. Any per-object clean-up code is placed in the destructor, which must (in Perl) be called DESTROY.

If constructors can have arbitrary names, then why not destructors? Because while a constructor is explicitly called, a destructor is not. Destruction happens automatically via Perl's garbage collection (GC) system, which is a quick but somewhat lazy reference-based GC system. To know what to call, Perl insists that the destructor be named DESTROY. Perl's notion of the right time to call a destructor is not well-defined currently, which is why your destructors should not rely on when they are called.

Why is DESTROY in all caps? Perl on occasion uses purely uppercase function names as a convention to indicate that the function will be automatically called by Perl in some way. Others that are called implicitly include BEGIN, END, AUTOLOAD, plus all methods used by tied objects, described in *perltie*.

In really good object-oriented programming languages, the user doesn't care when the destructor is called. It just happens when it's supposed to. In low-level languages without any GC at all, there's no way to depend on this happening at the right time, so the programmer must explicitly call the destructor to clean up memory and state, crossing their fingers that it's the right time to do so. Unlike C++, an object destructor is nearly never needed in Perl, and even when it is, explicit invocation is uncalled for. In the case of our Person class, we don't need a destructor because Perl takes care of simple matters like memory deallocation.

The only situation where Perl's reference-based GC won't work is when there's a circularity in the data structure, such as:

```
    $this->{WHATEVER} = $this;
```

In that case, you must delete the self-reference manually if you expect your program not to leak memory. While admittedly error-prone, this is the best we can do right now. Nonetheless, rest assured that when your program is finished, its objects' destructors are all duly called. So you are guaranteed that an object *eventually* gets properly destroyed, except in the unique case of a program that never exits. (If you're running Perl embedded in another application, this full GC pass happens a bit more frequently–whenever a thread shuts down.)

9.2.6 Other Object Methods

The methods we've talked about so far have either been constructors or else simple "data methods", interfaces to data stored in the object. These are a bit like an object's data members in the C++ world, except that strangers don't access them as data. Instead, they should only access the object's data indirectly via its methods. This is an important rule: in Perl, access to an object's data should *only* be made through methods.

Perl doesn't impose restrictions on who gets to use which methods. The public-versus-private distinction is by convention, not syntax. (Well, unless you use the Alias module described below in Data Members as Variables.) Occasionally you'll see method names beginning or ending with an underscore or two. This marking is a convention indicating that the methods are private to that class alone and sometimes to its closest acquaintances, its immediate subclasses. But this distinction is not enforced by Perl itself. It's up to the programmer to behave.

There's no reason to limit methods to those that simply access data. Methods can do anything at all. The key point is that they're invoked against an object or a class. Let's say we'd like object methods that do more than fetch or set one particular field.

```
    sub exclaim {
        my $self = shift;
        return sprintf "Hi, I'm %s, age %d, working with %s",
            $self->{NAME}, $self->{AGE}, join(", ", @{$self->{PEERS}});
    }
```

Or maybe even one like this:

```
    sub happy_birthday {
        my $self = shift;
        return ++$self->{AGE};
    }
```

Some might argue that one should go at these this way:

```
    sub exclaim {
        my $self = shift;
        return sprintf "Hi, I'm %s, age %d, working with %s",
            $self->name, $self->age, join(", ", $self->peers);
    }

    sub happy_birthday {
        my $self = shift;
        return $self->age( $self->age() + 1 );
    }
```

But since these methods are all executing in the class itself, this may not be critical. There are tradeoffs to be made. Using direct hash access is faster (about an order of magnitude faster, in fact), and it's more convenient when you want to interpolate in strings. But using methods (the external interface) internally shields not just the users of your class but even you yourself from changes in your data representation.

9.3 Class Data

What about "class data", data items common to each object in a class? What would you want that for? Well, in your Person class, you might like to keep track of the total people alive. How do you implement that?

You *could* make it a global variable called $Person::Census. But about only reason you'd do that would be if you *wanted* people to be able to get at your class data directly. They could just say $Person::Census and play around with it. Maybe this is ok in your design scheme. You might even conceivably want to make it an exported variable. To be exportable, a variable must be a (package) global. If this were a traditional module rather than an object-oriented one, you might do that.

While this approach is expected in most traditional modules, it's generally considered rather poor form in most object modules. In an object module, you should set up a protective veil to separate interface from implementation. So provide a class method to access class data just as you provide object methods to access object data.

So, you *could* still keep $Census as a package global and rely upon others to honor the contract of the module and therefore not play around with its implementation. You could even be supertricky and make $Census a tied object as described in *perltie*, thereby intercepting all accesses.

But more often than not, you just want to make your class data a file-scoped lexical. To do so, simply put this at the top of the file:

```
    my $Census = 0;
```

Even though the scope of a my() normally expires when the block in which it was declared is done (in this case the whole file being required or used), Perl's deep binding of lexical variables guarantees that the variable will not be deallocated, remaining accessible to functions declared within that scope. This doesn't work with global variables given temporary values via local(), though.

Irrespective of whether you leave $Census a package global or make it instead a file-scoped lexical, you should make these changes to your Person::new() constructor:

```
    sub new {
        my $class = shift;
        my $self  = {};
        $Census++;
        $self->{NAME}   = undef;
        $self->{AGE}    = undef;
        $self->{PEERS}  = [];
        bless ($self, $class);
        return $self;
    }

    sub population {
        return $Census;
    }
```

Now that we've done this, we certainly do need a destructor so that when Person is destroyed, the $Census goes down. Here's how this could be done:

```
    sub DESTROY { --$Census }
```

Notice how there's no memory to deallocate in the destructor? That's something that Perl takes care of for you all by itself.

Alternatively, you could use the Class::Data::Inheritable module from CPAN.

9.3.1 Accessing Class Data

It turns out that this is not really a good way to go about handling class data. A good scalable rule is that *you must never reference class data directly from an object method*. Otherwise you aren't building a scalable, inheritable class. The object must be the rendezvous point for all operations, especially from an object method. The globals (class data) would in some sense be in the "wrong" package in your derived classes. In Perl, methods execute in the context of the class they were defined in, *not* that of the object that triggered them. Therefore, namespace visibility of package globals in methods is unrelated to inheritance.

Got that? Maybe not. Ok, let's say that some other class "borrowed" (well, inherited) the DESTROY method as it was defined above. When those objects are destroyed, the original $Census variable will be altered, not the one in the new class's package namespace. Perhaps this is what you want, but probably it isn't.

Here's how to fix this. We'll store a reference to the data in the value accessed by the hash key "_CENSUS". Why the underscore? Well, mostly because an initial underscore already conveys strong feelings of magicalness to a C programmer. It's really just a mnemonic device to remind ourselves that this field is special and not to be used as a public data member in the same way that NAME, AGE, and PEERS are. (Because we've been developing this code under the strict pragma, prior to perl version 5.004 we'll have to quote the field name.)

```
    sub new {
        my $class = shift;
        my $self  = {};
        $self->{NAME}     = undef;
        $self->{AGE}      = undef;
        $self->{PEERS}    = [];
        # "private" data
        $self->{"_CENSUS"} = \$Census;
        bless ($self, $class);
        ++ ${ $self->{"_CENSUS"} };
        return $self;
    }

    sub population {
        my $self = shift;
        if (ref $self) {
            return ${ $self->{"_CENSUS"} };
        } else {
            return $Census;
        }
    }

    sub DESTROY {
        my $self = shift;
        -- ${ $self->{"_CENSUS"} };
    }
```

9.3.2 Debugging Methods

It's common for a class to have a debugging mechanism. For example, you might want to see when objects are created or destroyed. To do that, add a debugging variable as a file-scoped lexical. For this, we'll pull in the standard Carp module to emit our warnings and fatal messages. That way messages will come out with the caller's filename and line number instead of our own; if we wanted them to be from our own perspective, we'd just use die() and warn() directly instead of croak() and carp() respectively.

```
    use Carp;
    my $Debugging = 0;
```

Now add a new class method to access the variable.

```
    sub debug {
        my $class = shift;
        if (ref $class)  { confess "Class method called as object method" }
        unless (@_ == 1) { confess "usage: CLASSNAME->debug(level)" }
        $Debugging = shift;
    }
```

Now fix up DESTROY to murmur a bit as the moribund object expires:

```
    sub DESTROY {
        my $self = shift;
        if ($Debugging) { carp "Destroying $self " . $self->name }
        -- ${ $self->{"_CENSUS"} };
    }
```

One could conceivably make a per-object debug state. That way you could call both of these:

```
    Person->debug(1);   # entire class
    $him->debug(1);     # just this object
```

To do so, we need our debugging method to be a "bimodal" one, one that works on both classes *and* objects. Therefore, adjust the debug() and DESTROY methods as follows:

```
    sub debug {
        my $self = shift;
        confess "usage: thing->debug(level)"    unless @_ == 1;
        my $level = shift;
        if (ref($self))  {
            $self->{"_DEBUG"} = $level;         # just myself
        } else {
            $Debugging        = $level;         # whole class
        }
    }

    sub DESTROY {
        my $self = shift;
        if ($Debugging || $self->{"_DEBUG"}) {
            carp "Destroying $self " . $self->name;
        }
        -- ${ $self->{"_CENSUS"} };
    }
```

What happens if a derived class (which we'll call Employee) inherits methods from this Person base class? Then `Employee->debug()`, when called as a class method, manipulates $Person::Debugging not $Employee::Debugging.

9.3.3 Class Destructors

The object destructor handles the death of each distinct object. But sometimes you want a bit of cleanup when the entire class is shut down, which currently only happens when the program exits. To make such a *class destructor*, create a function in that class's package named END. This works just like the END function in traditional modules, meaning that it gets called whenever your program exits unless it execs or dies of an uncaught signal. For example,

```
    sub END {
        if ($Debugging) {
            print "All persons are going away now.\n";
        }
    }
```

When the program exits, all the class destructors (END functions) are be called in the opposite order that they were loaded in (LIFO order).

9.3.4 Documenting the Interface

And there you have it: we've just shown you the *implementation* of this Person class. Its *interface* would be its documentation. Usually this means putting it in pod ("plain old documentation") format right there in the same file. In our Person example, we would place the following docs anywhere in the Person.pm file. Even though it looks mostly like code, it's not. It's embedded documentation such as would be used by the pod2man, pod2html, or pod2text programs. The Perl compiler ignores pods entirely, just as the translators ignore code. Here's an example of some pods describing the informal interface:

```
    =head1 NAME

    Person - class to implement people

    =head1 SYNOPSIS

     use Person;

     #################
     # class methods #
     #################
     $ob    = Person->new;
     $count = Person->population;

     #######################
     # object data methods #
     #######################

     ### get versions ###
         $who   = $ob->name;
         $years = $ob->age;
         @pals  = $ob->peers;

     ### set versions ###
         $ob->name("Jason");
         $ob->age(23);
         $ob->peers( "Norbert", "Rhys", "Phineas" );

     ########################
     # other object methods #
     ########################

     $phrase = $ob->exclaim;
     $ob->happy_birthday;

    =head1 DESCRIPTION

    The Person class implements dah dee dah dee dah....
```

That's all there is to the matter of interface versus implementation. A programmer who opens up the module and plays around with all the private little shiny bits that were safely locked up behind the interface contract has voided the warranty, and you shouldn't worry about their fate.

9.4 Aggregation

Suppose you later want to change the class to implement better names. Perhaps you'd like to support both given names (called Christian names, irrespective of one's religion) and family names (called surnames), plus nicknames and titles. If users of your Person class have been properly accessing it through its documented interface, then you can easily change the underlying implementation. If they haven't, then they lose and it's their fault for breaking the contract and voiding their warranty.

To do this, we'll make another class, this one called Fullname. What's the Fullname class look like? To answer that question, you have to first figure out how you want to use it. How about we use it this way:

```
    $him = Person->new();
    $him->fullname->title("St");
    $him->fullname->christian("Thomas");
    $him->fullname->surname("Aquinas");
    $him->fullname->nickname("Tommy");
    printf "His normal name is %s\n", $him->name;
    printf "But his real name is %s\n", $him->fullname->as_string;
```

Ok. To do this, we'll change Person::new() so that it supports a full name field this way:

```
    sub new {
        my $class = shift;
        my $self  = {};
        $self->{FULLNAME} = Fullname->new();
        $self->{AGE}      = undef;
        $self->{PEERS}    = [];
        $self->{"_CENSUS"} = \$Census;
        bless ($self, $class);
        ++ ${ $self->{"_CENSUS"} };
        return $self;
    }

    sub fullname {
        my $self = shift;
        return $self->{FULLNAME};
    }
```

Then to support old code, define Person::name() this way:

```
    sub name {
        my $self = shift;
        return $self->{FULLNAME}->nickname(@_)
          ||   $self->{FULLNAME}->christian(@_);
    }
```

Here's the Fullname class. We'll use the same technique of using a hash reference to hold data fields, and methods by the appropriate name to access them:

```
    package Fullname;
    use strict;

    sub new {
        my $class = shift;
        my $self  = {
            TITLE       => undef,
            CHRISTIAN   => undef,
            SURNAME     => undef,
            NICK        => undef,
        };
        bless ($self, $class);
        return $self;
    }

    sub christian {
        my $self = shift;
        if (@_) { $self->{CHRISTIAN} = shift }
        return $self->{CHRISTIAN};
    }

    sub surname {
        my $self = shift;
        if (@_) { $self->{SURNAME} = shift }
        return $self->{SURNAME};
    }

    sub nickname {
        my $self = shift;
        if (@_) { $self->{NICK} = shift }
        return $self->{NICK};
    }

    sub title {
        my $self = shift;
        if (@_) { $self->{TITLE} = shift }
        return $self->{TITLE};
    }

    sub as_string {
        my $self = shift;
        my $name = join(" ", @$self{'CHRISTIAN', 'SURNAME'});
        if ($self->{TITLE}) {
            $name = $self->{TITLE} . " " . $name;
        }
        return $name;
    }

    1;
```

Finally, here’s the test program:

```
    #!/usr/bin/perl -w
    use strict;
    use Person;
    sub END { show_census() }

    sub show_census ()  {
        printf "Current population: %d\n", Person->population;
    }

    Person->debug(1);
    show_census();

    my $him = Person->new();

    $him->fullname->christian("Thomas");
    $him->fullname->surname("Aquinas");
    $him->fullname->nickname("Tommy");
    $him->fullname->title("St");
    $him->age(1);

    printf "%s is really %s.\n", $him->name, $him->fullname->as_string;
    printf "%s's age: %d.\n", $him->name, $him->age;
    $him->happy_birthday;
    printf "%s's age: %d.\n", $him->name, $him->age;

    show_census();
```

9.5 Inheritance

Object-oriented programming systems all support some notion of inheritance. Inheritance means allowing one class to piggy-back on top of another one so you don't have to write the same code again and again. It's about software reuse, and therefore related to Laziness, the principal virtue of a programmer. (The import/export mechanisms in traditional modules are also a form of code reuse, but a simpler one than the true inheritance that you find in object modules.)

Sometimes the syntax of inheritance is built into the core of the language, and sometimes it's not. Perl has no special syntax for specifying the class (or classes) to inherit from. Instead, it's all strictly in the semantics. Each package can have a variable called @ISA, which governs (method) inheritance. If you try to call a method on an object or class, and that method is not found in that object's package, Perl then looks to @ISA for other packages to go looking through in search of the missing method.

Like the special per-package variables recognized by Exporter (such as @EXPORT, @EXPORT_OK, @EXPORT_FAIL, %EXPORT_TAGS, and $VERSION), the @ISA array *must* be a package-scoped global and not a file-scoped lexical created via my(). Most classes have just one item in their @ISA array. In this case, we have what's called "single inheritance", or SI for short.

Consider this class:

```
    package Employee;
    use Person;
    @ISA = ("Person");
    1;
```

Not a lot to it, eh? All it's doing so far is loading in another class and stating that this one will inherit methods from that other class if need be. We have given it none of its own methods. We rely upon an Employee to behave just like a Person.

Setting up an empty class like this is called the "empty subclass test"; that is, making a derived class that does nothing but inherit from a base class. If the original base class has been designed properly, then the new derived class can be used as a drop-in replacement for the old one. This means you should be able to write a program like this:

```
    use Employee;
    my $empl = Employee->new();
    $empl->name("Jason");
    $empl->age(23);
    printf "%s is age %d.\n", $empl->name, $empl->age;
```

By proper design, we mean always using the two-argument form of bless(), avoiding direct access of global data, and not exporting anything. If you look back at the Person::new() function we defined above, we were careful to do that. There's a bit of package data used in the constructor, but the reference to this is stored on the object itself and all other methods access package data via that reference, so we should be ok.

What do we mean by the Person::new() function – isn't that actually a method? Well, in principle, yes. A method is just a function that expects as its first argument a class name (package) or object (blessed reference). Person::new() is the function that both the `Person->new()` method and the `Employee->new()` method end up calling. Understand that while a method call looks a lot like a function call, they aren't really quite the same, and if you treat them as the same, you'll very soon be left with nothing but broken programs. First, the actual underlying calling conventions are different: method calls get an extra argument. Second, function calls don't do inheritance, but methods do.

```
        Method Call             Resulting Function Call
        -----------             ------------------------
        Person->new()           Person::new("Person")
        Employee->new()         Person::new("Employee")
```

So don't use function calls when you mean to call a method.

If an employee is just a Person, that's not all too very interesting. So let's add some other methods. We'll give our employee data fields to access their salary, their employee ID, and their start date.

If you're getting a little tired of creating all these nearly identical methods just to get at the object's data, do not despair. Later, we'll describe several different convenience mechanisms for shortening this up. Meanwhile, here's the straight-forward way:

```
    sub salary {
        my $self = shift;
        if (@_) { $self->{SALARY} = shift }
        return $self->{SALARY};
    }

    sub id_number {
        my $self = shift;
        if (@_) { $self->{ID} = shift }
        return $self->{ID};
    }

    sub start_date {
        my $self = shift;
        if (@_) { $self->{START_DATE} = shift }
        return $self->{START_DATE};
    }
```

9.5.1 Overridden Methods

What happens when both a derived class and its base class have the same method defined? Well, then you get the derived class's version of that method. For example, let's say that we want the peers() method called on an employee to act a bit differently. Instead of just returning the list of peer names, let's return slightly different strings. So doing this:

```
    $empl->peers("Peter", "Paul", "Mary");
    printf "His peers are: %s\n", join(", ", $empl->peers);
```

will produce:

```
    His peers are: PEON=PETER, PEON=PAUL, PEON=MARY
```

To do this, merely add this definition into the Employee.pm file:

```
    sub peers {
        my $self = shift;
        if (@_) { @{ $self->{PEERS} } = @_ }
        return map { "PEON=\U$_" } @{ $self->{PEERS} };
    }
```

There, we've just demonstrated the high-falutin' concept known in certain circles as *polymorphism*. We've taken on the form and behaviour of an existing object, and then we've altered it to suit our own purposes. This is a form of Laziness. (Getting polymorphed is also what happens when the wizard decides you'd look better as a frog.)

Every now and then you'll want to have a method call trigger both its derived class (also known as "subclass") version as well as its base class (also known as "superclass") version. In practice, constructors and destructors are likely to want to do this, and it probably also makes sense in the debug() method we showed previously.

To do this, add this to Employee.pm:

```
    use Carp;
    my $Debugging = 0;

    sub debug {
        my $self = shift;
        confess "usage: thing->debug(level)"    unless @_ == 1;
        my $level = shift;
        if (ref($self))  {
            $self->{"_DEBUG"} = $level;
        } else {
            $Debugging = $level;            # whole class
        }
        Person::debug($self, $Debugging);   # don't really do this
    }
```

As you see, we turn around and call the Person package's debug() function. But this is far too fragile for good design. What if Person doesn't have a debug() function, but is inheriting *its* debug() method from elsewhere? It would have been slightly better to say

```
    Person->debug($Debugging);
```

But even that's got too much hard-coded. It's somewhat better to say

```
    $self->Person::debug($Debugging);
```

Which is a funny way to say to start looking for a debug() method up in Person. This strategy is more often seen on overridden object methods than on overridden class methods.

There is still something a bit off here. We've hard-coded our superclass's name. This in particular is bad if you change which classes you inherit from, or add others. Fortunately, the pseudoclass SUPER comes to the rescue here.

```
    $self->SUPER::debug($Debugging);
```

This way it starts looking in my class's @ISA. This only makes sense from *within* a method call, though. Don't try to access anything in SUPER:: from anywhere else, because it doesn't exist outside an overridden method call. Note that `SUPER` refers to the superclass of the current package, *not* to the superclass of `$self`.

Things are getting a bit complicated here. Have we done anything we shouldn't? As before, one way to test whether we're designing a decent class is via the empty subclass test. Since we already have an Employee class that we're trying to check, we'd better get a new empty subclass that can derive from Employee. Here's one:

```
    package Boss;
    use Employee;        # :-)
    @ISA = qw(Employee);
```

And here's the test program:

```
    #!/usr/bin/perl -w
    use strict;
    use Boss;
    Boss->debug(1);

    my $boss = Boss->new();

    $boss->fullname->title("Don");
    $boss->fullname->surname("Pichon Alvarez");
    $boss->fullname->christian("Federico Jesus");
    $boss->fullname->nickname("Fred");

    $boss->age(47);
    $boss->peers("Frank", "Felipe", "Faust");

    printf "%s is age %d.\n", $boss->fullname->as_string, $boss->age;
    printf "His peers are: %s\n", join(", ", $boss->peers);
```

Running it, we see that we're still ok. If you'd like to dump out your object in a nice format, somewhat like the way the 'x' command works in the debugger, you could use the Data::Dumper module from CPAN this way:

```
    use Data::Dumper;
    print "Here's the boss:\n";
    print Dumper($boss);
```

Which shows us something like this:

```
    Here's the boss:
    $VAR1 = bless( {
         _CENSUS => \1,
         FULLNAME => bless( {
                              TITLE => 'Don',
                              SURNAME => 'Pichon Alvarez',
                              NICK => 'Fred',
                              CHRISTIAN => 'Federico Jesus'
                            }, 'Fullname' ),
         AGE => 47,
         PEERS => [
                    'Frank',
                    'Felipe',
                    'Faust'
                  ]
       }, 'Boss' );
```

Hm.... something's missing there. What about the salary, start date, and ID fields? Well, we never set them to anything, even undef, so they don't show up in the hash's keys. The Employee class has no new() method of its own, and the new() method in Person doesn't know about Employees. (Nor should it: proper OO design dictates that a subclass be allowed to know about its immediate superclass, but never vice-versa.) So let's fix up Employee::new() this way:

```
    sub new {
        my $class = shift;
        my $self  = $class->SUPER::new();
        $self->{SALARY}        = undef;
        $self->{ID}            = undef;
        $self->{START_DATE}    = undef;
        bless ($self, $class);          # reconsecrate
        return $self;
    }
```

Now if you dump out an Employee or Boss object, you'll find that new fields show up there now.

9.5.2 Multiple Inheritance

Ok, at the risk of confusing beginners and annoying OO gurus, it's time to confess that Perl's object system includes that controversial notion known as multiple inheritance, or MI for short. All this means is that rather than having just one parent class who in turn might itself have a parent class, etc., that you can directly inherit from two or more parents. It's true that some uses of MI can get you into trouble, although hopefully not quite so much trouble with Perl as with dubiously-OO languages like C++.

The way it works is actually pretty simple: just put more than one package name in your @ISA array. When it comes time for Perl to go finding methods for your object, it looks at each of these packages in order. Well, kinda. It's actually a fully recursive, depth-first order. Consider a bunch of @ISA arrays like this:

```
    @First::ISA    = qw( Alpha );
    @Second::ISA   = qw( Beta );
    @Third::ISA    = qw( First Second );
```

If you have an object of class Third:

```
    my $ob = Third->new();
    $ob->spin();
```

How do we find a spin() method (or a new() method for that matter)? Because the search is depth-first, classes will be looked up in the following order: Third, First, Alpha, Second, and Beta.

In practice, few class modules have been seen that actually make use of MI. One nearly always chooses simple containership of one class within another over MI. That's why our Person object *contained* a Fullname object. That doesn't mean it *was* one.

However, there is one particular area where MI in Perl is rampant: borrowing another class's class methods. This is rather common, especially with some bundled "objectless" classes, like Exporter, DynaLoader, AutoLoader, and SelfLoader. These classes do not provide constructors; they exist only so you may inherit their class methods. (It's not entirely clear why inheritance was done here rather than traditional module importation.)

For example, here is the POSIX module's @ISA:

```
    package POSIX;
    @ISA = qw(Exporter DynaLoader);
```

The POSIX module isn't really an object module, but then, neither are Exporter or DynaLoader. They're just lending their classes' behaviours to POSIX.

Why don't people use MI for object methods much? One reason is that it can have complicated side-effects. For one thing, your inheritance graph (no longer a tree) might converge back to the same base class. Although Perl guards against recursive inheritance, merely having parents who are related to each other via a common ancestor, incestuous though it sounds, is not forbidden. What if in our Third class shown above we wanted its new() method to also call both overridden constructors in its two parent classes? The SUPER notation would only find the first one. Also, what about if the Alpha and Beta classes both had a common ancestor, like Nought? If you kept climbing up the inheritance tree calling overridden methods, you'd end up calling Nought::new() twice, which might well be a bad idea.

9.5.3 UNIVERSAL: The Root of All Objects

Wouldn’t it be convenient if all objects were rooted at some ultimate base class? That way you could give every object common methods without having to go and add it to each and every @ISA. Well, it turns out that you can. You don’t see it, but Perl tacitly and irrevocably assumes that there’s an extra element at the end of @ISA: the class UNIVERSAL. In version 5.003, there were no predefined methods there, but you could put whatever you felt like into it.

However, as of version 5.004 (or some subversive releases, like 5.003_08), UNIVERSAL has some methods in it already. These are builtin to your Perl binary, so they don’t take any extra time to load. Predefined methods include isa(), can(), and VERSION(). isa() tells you whether an object or class "is" another one without having to traverse the hierarchy yourself:

```
$has_io = $fd->isa("IO::Handle");
$itza_handle = IO::Socket->isa("IO::Handle");
```

The can() method, called against that object or class, reports back whether its string argument is a callable method name in that class. In fact, it gives you back a function reference to that method:

```
$his_print_method = $obj->can('as_string');
```

Finally, the VERSION method checks whether the class (or the object’s class) has a package global called $VERSION that’s high enough, as in:

```
 Some_Module->VERSION(3.0);
 $his_vers = $ob->VERSION();
```

However, we don’t usually call VERSION ourselves. (Remember that an all uppercase function name is a Perl convention that indicates that the function will be automatically used by Perl in some way.) In this case, it happens when you say

```
 use Some_Module 3.0;
```

If you wanted to add version checking to your Person class explained above, just add this to Person.pm:

```
 our $VERSION = '1.1';
```

and then in Employee.pm you can say

```
 use Person 1.1;
```

And it would make sure that you have at least that version number or higher available. This is not the same as loading in that exact version number. No mechanism currently exists for concurrent installation of multiple versions of a module. Lamentably.

9.6 Alternate Object Representations

Nothing requires objects to be implemented as hash references. An object can be any sort of reference so long as its referent has been suitably blessed. That means scalar, array, and code references are also fair game.

A scalar would work if the object has only one datum to hold. An array would work for most cases, but makes inheritance a bit dodgy because you have to invent new indices for the derived classes.

9.6.1 Arrays as Objects

If the user of your class honors the contract and sticks to the advertised interface, then you can change its underlying interface if you feel like it. Here's another implementation that conforms to the same interface specification. This time we'll use an array reference instead of a hash reference to represent the object.

```
package Person;
use strict;

my($NAME, $AGE, $PEERS) = ( 0 .. 2 );

############################################
## the object constructor (array version) ##
############################################
sub new {
    my $self = [];
    $self->[$NAME]   = undef;  # this is unnecessary
    $self->[$AGE]    = undef;  # as is this
    $self->[$PEERS]  = [];     # but this isn't, really
    bless($self);
    return $self;
}

sub name {
    my $self = shift;
    if (@_) { $self->[$NAME] = shift }
    return $self->[$NAME];
}

sub age {
    my $self = shift;
    if (@_) { $self->[$AGE] = shift }
    return $self->[$AGE];
}

sub peers {
    my $self = shift;
    if (@_) { @{ $self->[$PEERS] } = @_ }
    return @{ $self->[$PEERS] };
}

1;  # so the require or use succeeds
```

You might guess that the array access would be a lot faster than the hash access, but they're actually comparable. The array is a *little* bit faster, but not more than ten or fifteen percent, even when you replace the variables above like $AGE with literal numbers, like 1. A bigger difference between the two approaches can be found in memory use. A hash representation takes up more memory than an array representation because you have to allocate memory for the keys as well as for the values. However, it really isn't that bad, especially since as of version 5.004, memory is only allocated once for a given hash key, no matter how many hashes have that key. It's expected that sometime in the future, even these differences will fade into obscurity as more efficient underlying representations are devised.

Still, the tiny edge in speed (and somewhat larger one in memory) is enough to make some programmers choose an array representation for simple classes. There's still a little problem with scalability, though, because later in life when you feel like creating subclasses, you'll find that hashes just work out better.

9.6.2 Closures as Objects

Using a code reference to represent an object offers some fascinating possibilities. We can create a new anonymous function (closure) who alone in all the world can see the object's data. This is because we put the data into an anonymous hash that's lexically visible only to the closure we create, bless, and return as the object. This object's methods turn around and call the closure as a regular subroutine call, passing it the field we want to affect. (Yes, the double-function call is slow, but if you wanted fast, you wouldn't be using objects at all, eh? :-)

Use would be similar to before:

```
use Person;
$him = Person->new();
$him->name("Jason");
$him->age(23);
$him->peers( [ "Norbert", "Rhys", "Phineas" ] );
printf "%s is %d years old.\n", $him->name, $him->age;
print "His peers are: ", join(", ", @{$him->peers}), "\n";
```

but the implementation would be radically, perhaps even sublimely different:

```
package Person;

sub new {
     my $class  = shift;
     my $self = {
        NAME  => undef,
        AGE   => undef,
        PEERS => [],
     };
     my $closure = sub {
        my $field = shift;
        if (@_) { $self->{$field} = shift }
        return    $self->{$field};
    };
    bless($closure, $class);
    return $closure;
}

sub name   { &{ $_[0] }("NAME",  @_[ 1 .. $#_ ] ) }
sub age    { &{ $_[0] }("AGE",   @_[ 1 .. $#_ ] ) }
sub peers  { &{ $_[0] }("PEERS", @_[ 1 .. $#_ ] ) }

1;
```

Because this object is hidden behind a code reference, it's probably a bit mysterious to those whose background is more firmly rooted in standard procedural or object-based programming languages than in functional programming languages whence closures derive. The object created and returned by the new() method is itself not a data reference as we've seen before. It's an anonymous code reference that has within it access to a specific version (lexical binding and instantiation) of the object's data, which are stored in the private variable $self. Although this is the same function each time, it contains a different version of $self.

When a method like `$him->name("Jason")` is called, its implicit zeroth argument is the invoking object–just as it is with all method calls. But in this case, it's our code reference (something like a function pointer in C++, but with deep binding of lexical variables). There's not a lot to be done with a code reference beyond calling it, so that's just what we do when we say `&{$_[0]}`. This is just a regular function call, not a method call. The initial argument is the string "NAME", and any remaining arguments are whatever had been passed to the method itself.

Once we're executing inside the closure that had been created in new(), the $self hash reference suddenly becomes visible. The closure grabs its first argument ("NAME" in this case because that's what the name() method passed it), and uses that string to subscript into the private hash hidden in its unique version of $self.

Nothing under the sun will allow anyone outside the executing method to be able to get at this hidden data. Well, nearly nothing. You *could* single step through the program using the debugger and find out the pieces while you're in the method, but everyone else is out of luck.

There, if that doesn't excite the Scheme folks, then I just don't know what will. Translation of this technique into C++, Java, or any other braindead-static language is left as a futile exercise for aficionados of those camps.

You could even add a bit of nosiness via the caller() function and make the closure refuse to operate unless called via its own package. This would no doubt satisfy certain fastidious concerns of programming police and related puritans.

If you were wondering when Hubris, the third principle virtue of a programmer, would come into play, here you have it. (More seriously, Hubris is just the pride in craftsmanship that comes from having written a sound bit of well-designed code.)

9.7 AUTOLOAD: Proxy Methods

Autoloading is a way to intercept calls to undefined methods. An autoload routine may choose to create a new function on the fly, either loaded from disk or perhaps just eval()ed right there. This define-on-the-fly strategy is why it's called autoloading.

But that's only one possible approach. Another one is to just have the autoloaded method itself directly provide the requested service. When used in this way, you may think of autoloaded methods as "proxy" methods.

When Perl tries to call an undefined function in a particular package and that function is not defined, it looks for a function in that same package called AUTOLOAD. If one exists, it's called with the same arguments as the original function would have had. The fully-qualified name of the function is stored in that package's global variable $AUTOLOAD. Once called, the function can do anything it would like, including defining a new function by the right name, and then doing a really fancy kind of `goto` right to it, erasing itself from the call stack.

What does this have to do with objects? After all, we keep talking about functions, not methods. Well, since a method is just a function with an extra argument and some fancier semantics about where it's found, we can use autoloading for methods, too. Perl doesn't start looking for an AUTOLOAD method until it has exhausted the recursive hunt up through @ISA, though. Some programmers have even been known to define a UNIVERSAL::AUTOLOAD method to trap unresolved method calls to any kind of object.

9.7.1 Autoloaded Data Methods

You probably began to get a little suspicious about the duplicated code way back earlier when we first showed you the Person class, and then later the Employee class. Each method used to access the hash fields looked virtually identical. This should have tickled that great programming virtue, Impatience, but for the time, we let Laziness win out, and so did nothing. Proxy methods can cure this.

Instead of writing a new function every time we want a new data field, we'll use the autoload mechanism to generate (actually, mimic) methods on the fly. To verify that we're accessing a valid member, we will check against an `_permitted` (pronounced "under-permitted") field, which is a reference to a file-scoped lexical (like a C file static) hash of permitted fields in this record called %fields. Why the underscore? For the same reason as the _CENSUS field we once used: as a marker that means "for internal use only".

Here's what the module initialization code and class constructor will look like when taking this approach:

```
package Person;
use Carp;
our $AUTOLOAD;  # it's a package global

my %fields = (
    name        => undef,
    age         => undef,
    peers       => undef,
);
```

```
    sub new {
        my $class = shift;
        my $self  = {
            _permitted => \%fields,
            %fields,
        };
        bless $self, $class;
        return $self;
    }
```

If we wanted our record to have default values, we could fill those in where current we have `undef` in the %fields hash.

Notice how we saved a reference to our class data on the object itself? Remember that it's important to access class data through the object itself instead of having any method reference %fields directly, or else you won't have a decent inheritance.

The real magic, though, is going to reside in our proxy method, which will handle all calls to undefined methods for objects of class Person (or subclasses of Person). It has to be called AUTOLOAD. Again, it's all caps because it's called for us implicitly by Perl itself, not by a user directly.

```
    sub AUTOLOAD {
        my $self = shift;
        my $type = ref($self)
                    or croak "$self is not an object";

        my $name = $AUTOLOAD;
        $name =~ s/.*://;   # strip fully-qualified portion

        unless (exists $self->{_permitted}->{$name} ) {
            croak "Can't access `$name' field in class $type";
        }

        if (@_) {
            return $self->{$name} = shift;
        } else {
            return $self->{$name};
        }
    }
```

Pretty nifty, eh? All we have to do to add new data fields is modify %fields. No new functions need be written.

I could have avoided the `_permitted` field entirely, but I wanted to demonstrate how to store a reference to class data on the object so you wouldn't have to access that class data directly from an object method.

9.7.2 Inherited Autoloaded Data Methods

But what about inheritance? Can we define our Employee class similarly? Yes, so long as we're careful enough.

Here's how to be careful:

```
    package Employee;
    use Person;
    use strict;
    our @ISA = qw(Person);

    my %fields = (
        id          => undef,
        salary      => undef,
    );
    sub new {
        my $class = shift;
        my $self  = $class->SUPER::new();
        my($element);
        foreach $element (keys %fields) {
            $self->{_permitted}->{$element} = $fields{$element};
        }
        @{$self}{keys %fields} = values %fields;
        return $self;
    }
```

Once we've done this, we don't even need to have an AUTOLOAD function in the Employee package, because we'll grab Person's version of that via inheritance, and it will all work out just fine.

9.8 Metaclassical Tools

Even though proxy methods can provide a more convenient approach to making more struct-like classes than tediously coding up data methods as functions, it still leaves a bit to be desired. For one thing, it means you have to handle bogus calls that you don't mean to trap via your proxy. It also means you have to be quite careful when dealing with inheritance, as detailed above.

Perl programmers have responded to this by creating several different class construction classes. These metaclasses are classes that create other classes. A couple worth looking at are Class::Struct and Alias. These and other related metaclasses can be found in the modules directory on CPAN.

9.8.1 Class::Struct

One of the older ones is Class::Struct. In fact, its syntax and interface were sketched out long before perl5 even solidified into a real thing. What it does is provide you a way to "declare" a class as having objects whose fields are of a specific type. The function that does this is called, not surprisingly enough, struct(). Because structures or records are not base types in Perl, each time you want to create a class to provide a record-like data object, you yourself have to define a new() method, plus separate data-access methods for each of that record's fields. You'll quickly become bored with this process. The Class::Struct::struct() function alleviates this tedium.

Here's a simple example of using it:

```
    use Class::Struct qw(struct);
    use Jobbie;  # user-defined; see below

    struct 'Fred' => {
        one        => '$',
        many       => '@',
        profession => 'Jobbie',  # does not call Jobbie->new()
    };

    $ob = Fred->new(profession => Jobbie->new());
    $ob->one("hmmmm");

    $ob->many(0, "here");
    $ob->many(1, "you");
    $ob->many(2, "go");
    print "Just set: ", $ob->many(2), "\n";

    $ob->profession->salary(10_000);
```

You can declare types in the struct to be basic Perl types, or user-defined types (classes). User types will be initialized by calling that class's new() method.

Take care that the `Jobbie` object is not created automatically by the `Fred` class's new() method, so you should specify a `Jobbie` object when you create an instance of `Fred`.

Here's a real-world example of using struct generation. Let's say you wanted to override Perl's idea of gethostbyname() and gethostbyaddr() so that they would return objects that acted like C structures. We don't care about high-falutin' OO gunk. All we want is for these objects to act like structs in the C sense.

```
use Socket;
use Net::hostent;
$h = gethostbyname("perl.com");  # object return
printf "perl.com's real name is %s, address %s\n",
    $h->name, inet_ntoa($h->addr);
```

Here's how to do this using the Class::Struct module. The crux is going to be this call:

```
struct 'Net::hostent' => [          # note bracket
    name       => '$',
    aliases    => '@',
    addrtype   => '$',
    'length'   => '$',
    addr_list  => '@',
 ];
```

Which creates object methods of those names and types. It even creates a new() method for us.

We could also have implemented our object this way:

```
struct 'Net::hostent' => {          # note brace
    name       => '$',
    aliases    => '@',
    addrtype   => '$',
    'length'   => '$',
    addr_list  => '@',
 };
```

and then Class::Struct would have used an anonymous hash as the object type, instead of an anonymous array. The array is faster and smaller, but the hash works out better if you eventually want to do inheritance. Since for this struct-like object we aren't planning on inheritance, this time we'll opt for better speed and size over better flexibility.

Here's the whole implementation:

```
package Net::hostent;
use strict;

BEGIN {
    use Exporter   ();
    our @EXPORT      = qw(gethostbyname gethostbyaddr gethost);
    our @EXPORT_OK   = qw(
                           $h_name         @h_aliases
                           $h_addrtype     $h_length
                           @h_addr_list    $h_addr
                       );
    our %EXPORT_TAGS = ( FIELDS => [ @EXPORT_OK, @EXPORT ] );
}
our @EXPORT_OK;
```

```
    # Class::Struct forbids use of @ISA
    sub import { goto &Exporter::import }

    use Class::Struct qw(struct);
    struct 'Net::hostent' => [
       name        => '$',
       aliases     => '@',
       addrtype    => '$',
       'length'    => '$',
       addr_list   => '@',
    ];

    sub addr { shift->addr_list->[0] }

    sub populate (@) {
        return unless @_;
        my $hob = new();  # Class::Struct made this!
        $h_name     =    $hob->[0]              = $_[0];
        @h_aliases  = @{ $hob->[1] } = split ' ', $_[1];
        $h_addrtype =    $hob->[2]              = $_[2];
        $h_length   =    $hob->[3]              = $_[3];
        $h_addr     =                             $_[4];
        @h_addr_list = @{ $hob->[4] } =         @_[ (4 .. $#_) ];
        return $hob;
    }

    sub gethostbyname ($)  { populate(CORE::gethostbyname(shift)) }

    sub gethostbyaddr ($;$) {
        my ($addr, $addrtype);
        $addr = shift;
        require Socket unless @_;
        $addrtype = @_ ? shift : Socket::AF_INET();
        populate(CORE::gethostbyaddr($addr, $addrtype))
    }

    sub gethost($) {
        if ($_[0] =~ /^\d+(?:\.\d+(?:\.\d+(?:\.\d+)?)?)?$/) {
           require Socket;
           &gethostbyaddr(Socket::inet_aton(shift));
        } else {
           &gethostbyname;
        }
    }

    1;
```

We've snuck in quite a fair bit of other concepts besides just dynamic class creation, like overriding core functions, import/export bits, function prototyping, short-cut function call via `&whatever`, and function replacement with `goto &whatever`. These all mostly make sense from the perspective of a traditional module, but as you can see, we can also use them in an object module.

You can look at other object-based, struct-like overrides of core functions in the 5.004 release of Perl in File::stat, Net::hostent, Net::netent, Net::protoent, Net::servent, Time::gmtime, Time::localtime, User::grent, and User::pwent. These modules have a final component that's all lowercase, by convention reserved for compiler pragmas, because they affect the compilation and change a builtin function. They also have the type names that a C programmer would most expect.

9.8.2 Data Members as Variables

If you're used to C++ objects, then you're accustomed to being able to get at an object's data members as simple variables from within a method. The Alias module provides for this, as well as a good bit more, such as the possibility of private methods that the object can call but folks outside the class cannot.

Here's an example of creating a Person using the Alias module. When you update these magical instance variables, you automatically update value fields in the hash. Convenient, eh?

```
package Person;

# this is the same as before...
sub new {
     my $class = shift;
     my $self = {
        NAME  => undef,
        AGE   => undef,
        PEERS => [],
    };
    bless($self, $class);
    return $self;
}

use Alias qw(attr);
our ($NAME, $AGE, $PEERS);

sub name {
    my $self = attr shift;
    if (@_) { $NAME = shift; }
    return    $NAME;
}

sub age {
    my $self = attr shift;
    if (@_) { $AGE = shift; }
    return    $AGE;
}

sub peers {
    my $self = attr shift;
    if (@_) { @PEERS = @_; }
    return    @PEERS;
}

sub exclaim {
    my $self = attr shift;
    return sprintf "Hi, I'm %s, age %d, working with %s",
        $NAME, $AGE, join(", ", @PEERS);
}

sub happy_birthday {
    my $self = attr shift;
    return ++$AGE;
}
```

The need for the `our` declaration is because what Alias does is play with package globals with the same name as the fields. To use globals while `use strict` is in effect, you have to predeclare them. These package variables are localized to the block enclosing the attr() call just as if you'd used a local() on them. However, that means that they're still considered global variables with temporary values, just as with any other local().

It would be nice to combine Alias with something like Class::Struct or Class::MethodMaker.

9.9 NOTES

9.9.1 Object Terminology

In the various OO literature, it seems that a lot of different words are used to describe only a few different concepts. If you're not already an object programmer, then you don't need to worry about all these fancy words. But if you are, then you might like to know how to get at the same concepts in Perl.

For example, it's common to call an object an *instance* of a class and to call those objects' methods *instance methods*. Data fields peculiar to each object are often called *instance data* or *object attributes*, and data fields common to all members of that class are *class data*, *class attributes*, or *static data members*.

Also, *base class*, *generic class*, and *superclass* all describe the same notion, whereas *derived class*, *specific class*, and *subclass* describe the other related one.

C++ programmers have *static methods* and *virtual methods*, but Perl only has *class methods* and *object methods*. Actually, Perl only has methods. Whether a method gets used as a class or object method is by usage only. You could accidentally call a class method (one expecting a string argument) on an object (one expecting a reference), or vice versa.

From the C++ perspective, all methods in Perl are virtual. This, by the way, is why they are never checked for function prototypes in the argument list as regular builtin and user-defined functions can be.

Because a class is itself something of an object, Perl's classes can be taken as describing both a "class as meta-object" (also called *object factory*) philosophy and the "class as type definition" (*declaring* behaviour, not *defining* mechanism) idea. C++ supports the latter notion, but not the former.

9.10 SEE ALSO

The following manpages will doubtless provide more background for this one: *perlmod*, *perlref*, *perlobj*, *perlbot*, *perltie*, and *overload*.

perlboot is a kinder, gentler introduction to object-oriented programming.

perltooc provides more detail on class data.

Some modules which might prove interesting are Class::Accessor, Class::Class, Class::Contract, Class::Data::Inheritable, Class::MethodMaker and Tie::SecureHash

9.11 AUTHOR AND COPYRIGHT

Copyright (c) 1997, 1998 Tom Christiansen All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

9.12 COPYRIGHT

9.12.1 Acknowledgments

Thanks to Larry Wall, Roderick Schertler, Gurusamy Sarathy, Dean Roehrich, Raphael Manfredi, Brent Halsey, Greg Bacon, Brad Appleton, and many others for their helpful comments.

Chapter 10

perltooc

Tom’s OO Tutorial for Class Data in Perl

10.1 DESCRIPTION

When designing an object class, you are sometimes faced with the situation of wanting common state shared by all objects of that class. Such *class attributes* act somewhat like global variables for the entire class, but unlike program-wide globals, class attributes have meaning only to the class itself.

Here are a few examples where class attributes might come in handy:

- to keep a count of the objects you’ve created, or how many are still extant.
- to extract the name or file descriptor for a logfile used by a debugging method.
- to access collective data, like the total amount of cash dispensed by all ATMs in a network in a given day.
- to access the last object created by a class, or the most accessed object, or to retrieve a list of all objects.

Unlike a true global, class attributes should not be accessed directly. Instead, their state should be inspected, and perhaps altered, only through the mediated access of *class methods*. These class attributes accessor methods are similar in spirit and function to accessors used to manipulate the state of instance attributes on an object. They provide a clear firewall between interface and implementation.

You should allow access to class attributes through either the class name or any object of that class. If we assume that $an_object is of type Some_Class, and the &Some_Class::population_count method accesses class attributes, then these two invocations should both be possible, and almost certainly equivalent.

```
    Some_Class->population_count()
    $an_object->population_count()
```

The question is, where do you store the state which that method accesses? Unlike more restrictive languages like C++, where these are called static data members, Perl provides no syntactic mechanism to declare class attributes, any more than it provides a syntactic mechanism to declare instance attributes. Perl provides the developer with a broad set of powerful but flexible features that can be uniquely crafted to the particular demands of the situation.

A class in Perl is typically implemented in a module. A module consists of two complementary feature sets: a package for interfacing with the outside world, and a lexical file scope for privacy. Either of these two mechanisms can be used to implement class attributes. That means you get to decide whether to put your class attributes in package variables or to put them in lexical variables.

And those aren’t the only decisions to make. If you choose to use package variables, you can make your class attribute accessor methods either ignorant of inheritance or sensitive to it. If you choose lexical variables, you can elect to permit access to them from anywhere in the entire file scope, or you can limit direct data access exclusively to the methods implementing those attributes.

10.2 Class Data in a Can

One of the easiest ways to solve a hard problem is to let someone else do it for you! In this case, Class::Data::Inheritable (available on a CPAN near you) offers a canned solution to the class data problem using closures. So before you wade into this document, consider having a look at that module.

10.3 Class Data as Package Variables

Because a class in Perl is really just a package, using package variables to hold class attributes is the most natural choice. This makes it simple for each class to have its own class attributes. Let's say you have a class called Some_Class that needs a couple of different attributes that you'd like to be global to the entire class. The simplest thing to do is to use package variables like $Some_Class::CData1 and $Some_Class::CData2 to hold these attributes. But we certainly don't want to encourage outsiders to touch those data directly, so we provide methods to mediate access.

In the accessor methods below, we'll for now just ignore the first argument–that part to the left of the arrow on method invocation, which is either a class name or an object reference.

```
    package Some_Class;
    sub CData1 {
        shift;  # XXX: ignore calling class/object
        $Some_Class::CData1 = shift if @_;
        return $Some_Class::CData1;
    }
    sub CData2 {
        shift;  # XXX: ignore calling class/object
        $Some_Class::CData2 = shift if @_;
        return $Some_Class::CData2;
    }
```

This technique is highly legible and should be completely straightforward to even the novice Perl programmer. By fully qualifying the package variables, they stand out clearly when reading the code. Unfortunately, if you misspell one of these, you've introduced an error that's hard to catch. It's also somewhat disconcerting to see the class name itself hard-coded in so many places.

Both these problems can be easily fixed. Just add the `use strict` pragma, then pre-declare your package variables. (The `our` operator will be new in 5.6, and will work for package globals just like `my` works for scoped lexicals.)

```
    package Some_Class;
    use strict;
    our($CData1, $CData2);      # our() is new to perl5.6
    sub CData1 {
        shift;  # XXX: ignore calling class/object
        $CData1 = shift if @_;
        return $CData1;
    }
    sub CData2 {
        shift;  # XXX: ignore calling class/object
        $CData2 = shift if @_;
        return $CData2;
    }
```

As with any other global variable, some programmers prefer to start their package variables with capital letters. This helps clarity somewhat, but by no longer fully qualifying the package variables, their significance can be lost when reading the code. You can fix this easily enough by choosing better names than were used here.

10.3.1 Putting All Your Eggs in One Basket

Just as the mindless enumeration of accessor methods for instance attributes grows tedious after the first few (see *perltoot*), so too does the repetition begin to grate when listing out accessor methods for class data. Repetition runs counter to the primary virtue of a programmer: Laziness, here manifesting as that innate urge every programmer feels to factor out duplicate code whenever possible.

Here's what to do. First, make just one hash to hold all class attributes.

```
    package Some_Class;
    use strict;
    our %ClassData = (          # our() is new to perl5.6
        CData1 => "",
        CData2 => "",
    );
```

Using closures (see *perlref*) and direct access to the package symbol table (see *perlmod*), now clone an accessor method for each key in the %ClassData hash. Each of these methods is used to fetch or store values to the specific, named class attribute.

```
    for my $datum (keys %ClassData) {
        no strict "refs";       # to register new methods in package
        *$datum = sub {
            shift;      # XXX: ignore calling class/object
            $ClassData{$datum} = shift if @_;
            return $ClassData{$datum};
        }
    }
```

It's true that you could work out a solution employing an &AUTOLOAD method, but this approach is unlikely to prove satisfactory. Your function would have to distinguish between class attributes and object attributes; it could interfere with inheritance; and it would have to careful about DESTROY. Such complexity is uncalled for in most cases, and certainly in this one.

You may wonder why we're rescinding strict refs for the loop. We're manipulating the package's symbol table to introduce new function names using symbolic references (indirect naming), which the strict pragma would otherwise forbid. Normally, symbolic references are a dodgy notion at best. This isn't just because they can be used accidentally when you aren't meaning to. It's also because for most uses to which beginning Perl programmers attempt to put symbolic references, we have much better approaches, like nested hashes or hashes of arrays. But there's nothing wrong with using symbolic references to manipulate something that is meaningful only from the perspective of the package symbol table, like method names or package variables. In other words, when you want to refer to the symbol table, use symbol references.

Clustering all the class attributes in one place has several advantages. They're easy to spot, initialize, and change. The aggregation also makes them convenient to access externally, such as from a debugger or a persistence package. The only possible problem is that we don't automatically know the name of each class's class object, should it have one. This issue is addressed below in §10.3.3.

10.3.2 Inheritance Concerns

Suppose you have an instance of a derived class, and you access class data using an inherited method call. Should that end up referring to the base class's attributes, or to those in the derived class? How would it work in the earlier examples? The derived class inherits all the base class's methods, including those that access class attributes. But what package are the class attributes stored in?

The answer is that, as written, class attributes are stored in the package into which those methods were compiled. When you invoke the &CData1 method on the name of the derived class or on one of that class's objects, the version shown above is still run, so you'll access $Some_Class::CData1–or in the method cloning version, `$Some_Class::ClassData{CData1}`.

Think of these class methods as executing in the context of their base class, not in that of their derived class. Sometimes this is exactly what you want. If Feline subclasses Carnivore, then the population of Carnivores in the world should go up when a new Feline is born. But what if you wanted to figure out how many Felines you have apart from Carnivores? The current approach doesn't support that.

You'll have to decide on a case-by-case basis whether it makes any sense for class attributes to be package-relative. If you want it to be so, then stop ignoring the first argument to the function. Either it will be a package name if the method was invoked directly on a class name, or else it will be an object reference if the method was invoked on an object reference. In the latter case, the ref() function provides the class of that object.

```
package Some_Class;
sub CData1 {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    my $varname = $class . "::CData1";
    no strict "refs";       # to access package data symbolically
    $$varname = shift if @_;
    return $$varname;
}
```

And then do likewise for all other class attributes (such as CData2, etc.) that you wish to access as package variables in the invoking package instead of the compiling package as we had previously.

Once again we temporarily disable the strict references ban, because otherwise we couldn't use the fully-qualified symbolic name for the package global. This is perfectly reasonable: since all package variables by definition live in a package, there's nothing wrong with accessing them via that package's symbol table. That's what it's there for (well, somewhat).

What about just using a single hash for everything and then cloning methods? What would that look like? The only difference would be the closure used to produce new method entries for the class's symbol table.

```
no strict "refs";
*$datum = sub {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    my $varname = $class . "::ClassData";
    $varname->{$datum} = shift if @_;
    return $varname->{$datum};
}
```

10.3.3 The Eponymous Meta-Object

It could be argued that the %ClassData hash in the previous example is neither the most imaginative nor the most intuitive of names. Is there something else that might make more sense, be more useful, or both?

As it happens, yes, there is. For the "class meta-object", we'll use a package variable of the same name as the package itself. Within the scope of a package Some_Class declaration, we'll use the eponymously named hash %Some_Class as that class's meta-object. (Using an eponymously named hash is somewhat reminiscent of classes that name their constructors eponymously in the Python or C++ fashion. That is, class Some_Class would use &Some_Class::Some_Class as a constructor, probably even exporting that name as well. The StrNum class in Recipe 13.14 in *The Perl Cookbook* does this, if you're looking for an example.)

This predictable approach has many benefits, including having a well-known identifier to aid in debugging, transparent persistence, or checkpointing. It's also the obvious name for monadic classes and translucent attributes, discussed later.

Here's an example of such a class. Notice how the name of the hash storing the meta-object is the same as the name of the package used to implement the class.

```
package Some_Class;
use strict;
```

```
    # create class meta-object using that most perfect of names
    our %Some_Class = (         # our() is new to perl5.6
        CData1 => "",
        CData2 => "",
    );

    # this accessor is calling-package-relative
    sub CData1 {
        my $obclass = shift;
        my $class   = ref($obclass) || $obclass;
        no strict "refs";       # to access eponymous meta-object
        $class->{CData1} = shift if @_;
        return $class->{CData1};
    }

    # but this accessor is not
    sub CData2 {
        shift;                  # XXX: ignore calling class/object
        no strict "refs";       # to access eponymous meta-object
        __PACKAGE__ -> {CData2} = shift if @_;
        return __PACKAGE__ -> {CData2};
    }
```

In the second accessor method, the __PACKAGE__ notation was used for two reasons. First, to avoid hardcoding the literal package name in the code in case we later want to change that name. Second, to clarify to the reader that what matters here is the package currently being compiled into, not the package of the invoking object or class. If the long sequence of non-alphabetic characters bothers you, you can always put the __PACKAGE__ in a variable first.

```
    sub CData2 {
        shift;                  # XXX: ignore calling class/object
        no strict "refs";       # to access eponymous meta-object
        my $class = __PACKAGE__;
        $class->{CData2} = shift if @_;
        return $class->{CData2};
    }
```

Even though we're using symbolic references for good not evil, some folks tend to become unnerved when they see so many places with strict ref checking disabled. Given a symbolic reference, you can always produce a real reference (the reverse is not true, though). So we'll create a subroutine that does this conversion for us. If invoked as a function of no arguments, it returns a reference to the compiling class's eponymous hash. Invoked as a class method, it returns a reference to the eponymous hash of its caller. And when invoked as an object method, this function returns a reference to the eponymous hash for whatever class the object belongs to.

```
    package Some_Class;
    use strict;

    our %Some_Class = (         # our() is new to perl5.6
        CData1 => "",
        CData2 => "",
    );

    # tri-natured: function, class method, or object method
    sub _classobj {
        my $obclass = shift || __PACKAGE__;
        my $class   = ref($obclass) || $obclass;
        no strict "refs";   # to convert sym ref to real one
        return \%$class;
    }
    for my $datum (keys %{ _classobj() } ) {
        # turn off strict refs so that we can
        # register a method in the symbol table
        no strict "refs";
        *$datum = sub {
            use strict "refs";
            my $self = shift->_classobj();
            $self->{$datum} = shift if @_;
            return $self->{$datum};
        }
    }
```

10.3.4 Indirect References to Class Data

A reasonably common strategy for handling class attributes is to store a reference to each package variable on the object itself. This is a strategy you've probably seen before, such as in *perltoot* and *perlbot*, but there may be variations in the example below that you haven't thought of before.

```
    package Some_Class;
    our($CData1, $CData2);              # our() is new to perl5.6

    sub new {
        my $obclass = shift;
        return bless my $self = {
            ObData1 => "",
            ObData2 => "",
            CData1  => \$CData1,
            CData2  => \$CData2,
        } => (ref $obclass || $obclass);
    }

    sub ObData1 {
        my $self = shift;
        $self->{ObData1} = shift if @_;
        return $self->{ObData1};
    }

    sub ObData2 {
        my $self = shift;
        $self->{ObData2} = shift if @_;
        return $self->{ObData2};
    }

    sub CData1 {
        my $self = shift;
        my $dataref = ref $self
                        ? $self->{CData1}
                        : \$CData1;
        $$dataref = shift if @_;
        return $$dataref;
    }

    sub CData2 {
        my $self = shift;
        my $dataref = ref $self
```

```
                        ? $self->{CData2}
                        : \$CData2;
        $$dataref = shift if @_;
        return $$dataref;
    }
```

As written above, a derived class will inherit these methods, which will consequently access package variables in the base class's package. This is not necessarily expected behavior in all circumstances. Here's an example that uses a variable meta-object, taking care to access the proper package's data.

```
        package Some_Class;
        use strict;

        our %Some_Class = (     # our() is new to perl5.6
            CData1 => "",
            CData2 => "",
        );

        sub _classobj {
            my $self  = shift;
            my $class = ref($self) || $self;
            no strict "refs";
            # get (hard) ref to eponymous meta-object
            return \%$class;
        }

        sub new {
            my $obclass  = shift;
            my $classobj = $obclass->_classobj();
            bless my $self = {
                ObData1 => "",
                ObData2 => "",
                CData1  => \$classobj->{CData1},
                CData2  => \$classobj->{CData2},
            } => (ref $obclass || $obclass);
            return $self;
        }

        sub ObData1 {
            my $self = shift;
            $self->{ObData1} = shift if @_;
            return $self->{ObData1};
        }

        sub ObData2 {
            my $self = shift;
            $self->{ObData2} = shift if @_;
            return $self->{ObData2};
        }

        sub CData1 {
            my $self = shift;
            $self = $self->_classobj() unless ref $self;
            my $dataref = $self->{CData1};
            $$dataref = shift if @_;
            return $$dataref;
        }
        sub CData2 {
            my $self = shift;
            $self = $self->_classobj() unless ref $self;
            my $dataref = $self->{CData2};
            $$dataref = shift if @_;
            return $$dataref;
        }
```

Not only are we now strict refs clean, using an eponymous meta-object seems to make the code cleaner. Unlike the previous version, this one does something interesting in the face of inheritance: it accesses the class meta-object in the invoking class instead of the one into which the method was initially compiled.

You can easily access data in the class meta-object, making it easy to dump the complete class state using an external mechanism such as when debugging or implementing a persistent class. This works because the class meta-object is a package variable, has a well-known name, and clusters all its data together. (Transparent persistence is not always feasible, but it's certainly an appealing idea.)

There's still no check that object accessor methods have not been invoked on a class name. If strict ref checking is enabled, you'd blow up. If not, then you get the eponymous meta-object. What you do with–or about–this is up to you. The next two sections demonstrate innovative uses for this powerful feature.

10.3.5 Monadic Classes

Some of the standard modules shipped with Perl provide class interfaces without any attribute methods whatsoever. The most commonly used module not numbered amongst the pragmata, the Exporter module, is a class with neither constructors nor attributes. Its job is simply to provide a standard interface for modules wishing to export part of their namespace into that of their caller. Modules use the Exporter's &import method by setting their inheritance list in their package's @ISA array to mention "Exporter". But class Exporter provides no constructor, so you can't have several instances of the class. In fact, you can't have any–it just doesn't make any sense. All you get is its methods. Its interface contains no statefulness, so state data is wholly superfluous.

Another sort of class that pops up from time to time is one that supports a unique instance. Such classes are called *monadic classes*, or less formally, *singletons* or *highlander classes*.

If a class is monadic, where do you store its state, that is, its attributes? How do you make sure that there's never more than one instance? While you could merely use a slew of package variables, it's a lot cleaner to use the eponymously named hash. Here's a complete example of a monadic class:

```
    package Cosmos;
    %Cosmos = ();

    # accessor method for "name" attribute
    sub name {
        my $self = shift;
        $self->{name} = shift if @_;
        return $self->{name};
    }

    # read-only accessor method for "birthday" attribute
    sub birthday {
        my $self = shift;
        die "can't reset birthday" if @_;  # XXX: croak() is better
        return $self->{birthday};
    }

    # accessor method for "stars" attribute
    sub stars {
        my $self = shift;
        $self->{stars} = shift if @_;
        return $self->{stars};
    }
    # oh my - one of our stars just went out!
    sub supernova {
        my $self = shift;
        my $count = $self->stars();
        $self->stars($count - 1) if $count > 0;
    }

    # constructor/initializer method - fix by reboot
    sub bigbang {
        my $self = shift;
        %$self = (
            name         => "the world according to tchrist",
            birthday     => time(),
            stars        => 0,
        );
        return $self;       # yes, it's probably a class.  SURPRISE!
    }

    # After the class is compiled, but before any use or require
    # returns, we start off the universe with a bang.
    __PACKAGE__ -> bigbang();
```

Hold on, that doesn't look like anything special. Those attribute accessors look no different than they would if this were a regular class instead of a monadic one. The crux of the matter is there's nothing that says that $self must hold a reference to a blessed object. It merely has to be something you can invoke methods on. Here the package name itself, Cosmos, works as an object. Look at the &supernova method. Is that a class method or an object method? The answer is that static analysis cannot reveal the answer. Perl doesn't care, and neither should you. In the three attribute methods, `%$self` is really accessing the %Cosmos package variable.

If like Stephen Hawking, you posit the existence of multiple, sequential, and unrelated universes, then you can invoke the &bigbang method yourself at any time to start everything all over again. You might think of &bigbang as more of an initializer than a constructor, since the function doesn't allocate new memory; it only initializes what's already there. But like any other constructor, it does return a scalar value to use for later method invocations.

Imagine that some day in the future, you decide that one universe just isn't enough. You could write a new class from scratch, but you already have an existing class that does what you want–except that it's monadic, and you want more than just one cosmos.

That's what code reuse via subclassing is all about. Look how short the new code is:

```
    package Multiverse;
    use Cosmos;
    @ISA = qw(Cosmos);

    sub new {
        my $protoverse = shift;
        my $class      = ref($protoverse) || $protoverse;
        my $self       = {};
        return bless($self, $class)->bigbang();
    }
    1;
```

Because we were careful to be good little creators when we designed our Cosmos class, we can now reuse it without touching a single line of code when it comes time to write our Multiverse class. The same code that worked when invoked as a class method continues to work perfectly well when invoked against separate instances of a derived class.

The astonishing thing about the Cosmos class above is that the value returned by the &bigbang "constructor" is not a reference to a blessed object at all. It's just the class's own name. A class name is, for virtually all intents and purposes, a perfectly acceptable object. It has state, behavior, and identity, the three crucial components of an object system. It even manifests inheritance, polymorphism, and encapsulation. And what more can you ask of an object?

To understand object orientation in Perl, it's important to recognize the unification of what other programming languages might think of as class methods and object methods into just plain methods. "Class methods" and "object methods" are distinct only in the compartmentalizing mind of the Perl programmer, not in the Perl language itself.

Along those same lines, a constructor is nothing special either, which is one reason why Perl has no pre-ordained name for them. "Constructor" is just an informal term loosely used to describe a method that returns a scalar value that you can make further method calls against. So long as it's either a class name or an object reference, that's good enough. It doesn't even have to be a reference to a brand new object.

You can have as many–or as few–constructors as you want, and you can name them whatever you care to. Blindly and obediently using new() for each and every constructor you ever write is to speak Perl with such a severe C++ accent that you do a disservice to both languages. There's no reason to insist that each class have but one constructor, or that a constructor be named new(), or that a constructor be used solely as a class method and not an object method.

The next section shows how useful it can be to further distance ourselves from any formal distinction between class method calls and object method calls, both in constructors and in accessor methods.

10.3.6 Translucent Attributes

A package's eponymous hash can be used for more than just containing per-class, global state data. It can also serve as a sort of template containing default settings for object attributes. These default settings can then be used in constructors for initialization of a particular object. The class's eponymous hash can also be used to implement *translucent attributes*. A translucent attribute is one that has a class-wide default. Each object can set its own value for the attribute, in which case `$object->attribute()` returns that value. But if no value has been set, then `$object->attribute()` returns the class-wide default.

We'll apply something of a copy-on-write approach to these translucent attributes. If you're just fetching values from them, you get translucency. But if you store a new value to them, that new value is set on the current object. On the other hand, if you use the class as an object and store the attribute value directly on the class, then the meta-object's value changes, and later fetch operations on objects with uninitialized values for those attributes will retrieve the meta-object's new values. Objects with their own initialized values, however, won't see any change.

Let's look at some concrete examples of using these properties before we show how to implement them. Suppose that a class named Some_Class had a translucent data attribute called "color". First you set the color in the meta-object, then you create three objects using a constructor that happens to be named &spawn.

```
    use Vermin;
    Vermin->color("vermilion");

    $ob1 = Vermin->spawn();     # so that's where Jedi come from
    $ob2 = Vermin->spawn();
    $ob3 = Vermin->spawn();

    print $obj3->color();       # prints "vermilion"
```

Each of these objects' colors is now "vermilion", because that's the meta-object's value for that attribute, and these objects do not have individual color values set.

Changing the attribute on one object has no effect on other objects previously created.

```
    $ob3->color("chartreuse");
    print $ob3->color();        # prints "chartreuse"
    print $ob1->color();        # prints "vermilion", translucently
```

If you now use $ob3 to spawn off another object, the new object will take the color its parent held, which now happens to be "chartreuse". That's because the constructor uses the invoking object as its template for initializing attributes. When that invoking object is the class name, the object used as a template is the eponymous meta-object. When the invoking object is a reference to an instantiated object, the &spawn constructor uses that existing object as a template.

```
    $ob4 = $ob3->spawn();       # $ob3 now template, not %Vermin
    print $ob4->color();        # prints "chartreuse"
```

Any actual values set on the template object will be copied to the new object. But attributes undefined in the template object, being translucent, will remain undefined and consequently translucent in the new one as well.

Now let's change the color attribute on the entire class:

```
    Vermin->color("azure");
    print $ob1->color();        # prints "azure"
    print $ob2->color();        # prints "azure"
    print $ob3->color();        # prints "chartreuse"
    print $ob4->color();        # prints "chartreuse"
```

That color change took effect only in the first pair of objects, which were still translucently accessing the meta-object's values. The second pair had per-object initialized colors, and so didn't change.

One important question remains. Changes to the meta-object are reflected in translucent attributes in the entire class, but what about changes to discrete objects? If you change the color of $ob3, does the value of $ob4 see that change? Or vice-versa. If you change the color of $ob4, does then the value of $ob3 shift?

```
    $ob3->color("amethyst");
    print $ob3->color();        # prints "amethyst"
    print $ob4->color();        # hmm: "chartreuse" or "amethyst"?
```

While one could argue that in certain rare cases it should, let's not do that. Good taste aside, we want the answer to the question posed in the comment above to be "chartreuse", not "amethyst". So we'll treat these attributes similar to the way process attributes like environment variables, user and group IDs, or the current working directory are treated across a fork(). You can change only yourself, but you will see those changes reflected in your unspawned children. Changes to one object will propagate neither up to the parent nor down to any existing child objects. Those objects made later, however, will see the changes.

If you have an object with an actual attribute value, and you want to make that object's attribute value translucent again, what do you do? Let's design the class so that when you invoke an accessor method with `undef` as its argument, that attribute returns to translucency.

```
    $ob4->color(undef);         # back to "azure"
```

Here's a complete implementation of Vermin as described above.

```
    package Vermin;

    # here's the class meta-object, eponymously named.
    # it holds all class attributes, and also all instance attributes
    # so the latter can be used for both initialization
    # and translucency.

    our %Vermin = (             # our() is new to perl5.6
        PopCount => 0,          # capital for class attributes
        color    => "beige",    # small for instance attributes
    );

    # constructor method
    # invoked as class method or object method
    sub spawn {
        my $obclass = shift;
        my $class   = ref($obclass) || $obclass;
        my $self = {};
        bless($self, $class);
        $class->{PopCount}++;
        # init fields from invoking object, or omit if
        # invoking object is the class to provide translucency
        %$self = %$obclass if ref $obclass;
        return $self;
    }

    # translucent accessor for "color" attribute
    # invoked as class method or object method
    sub color {
        my $self  = shift;
        my $class = ref($self) || $self;

        # handle class invocation
        unless (ref $self) {
            $class->{color} = shift if @_;
            return $class->{color}
        }

        # handle object invocation
        $self->{color} = shift if @_;
        if (defined $self->{color}) {  # not exists!
            return $self->{color};
        } else {
            return $class->{color};
        }
    }

    # accessor for "PopCount" class attribute
    # invoked as class method or object method
    # but uses object solely to locate meta-object
    sub population {
        my $obclass = shift;
        my $class   = ref($obclass) || $obclass;
        return $class->{PopCount};
    }

    # instance destructor
    # invoked only as object method
    sub DESTROY {
        my $self  = shift;
        my $class = ref $self;
        $class->{PopCount}--;
    }
```

Here are a couple of helper methods that might be convenient. They aren't accessor methods at all. They're used to detect accessibility of data attributes. The &is_translucent method determines whether a particular object attribute is coming from the meta-object. The &has_attribute method detects whether a class implements a particular property at all. It could also be used to distinguish undefined properties from non-existent ones.

```
    # detect whether an object attribute is translucent
    # (typically?) invoked only as object method
    sub is_translucent {
        my($self, $attr)  = @_;
        return !defined $self->{$attr};
    }
    # test for presence of attribute in class
    # invoked as class method or object method
    sub has_attribute {
        my($self, $attr)  = @_;
        my $class = ref($self) || $self;
        return exists $class->{$attr};
    }
```

If you prefer to install your accessors more generically, you can make use of the upper-case versus lower-case convention to register into the package appropriate methods cloned from generic closures.

```
    for my $datum (keys %{ +__PACKAGE__ }) {
        *$datum = ($datum =~ /^[A-Z]/)
            ? sub {  # install class accessor
                    my $obclass = shift;
                    my $class   = ref($obclass) || $obclass;
                    return $class->{$datum};
                  }
            : sub { # install translucent accessor
                    my $self  = shift;
                    my $class = ref($self) || $self;
                    unless (ref $self) {
                        $class->{$datum} = shift if @_;
                        return $class->{$datum}
                    }
                    $self->{$datum} = shift if @_;
                    return defined $self->{$datum}
                        ? $self  -> {$datum}
                        : $class -> {$datum}
                  }
    }
```

Translations of this closure-based approach into C++, Java, and Python have been left as exercises for the reader. Be sure to send us mail as soon as you're done.

10.4 Class Data as Lexical Variables

10.4.1 Privacy and Responsibility

Unlike conventions used by some Perl programmers, in the previous examples, we didn't prefix the package variables used for class attributes with an underscore, nor did we do so for the names of the hash keys used for instance attributes. You don't need little markers on data names to suggest nominal privacy on attribute variables or hash keys, because these are **already** notionally private! Outsiders have no business whatsoever playing with anything within a class save through the mediated access of its documented interface; in other words, through method invocations. And not even through just any method, either. Methods that begin with an underscore are traditionally considered off-limits outside the class. If outsiders skip the documented method interface to poke around the internals of your class and end up breaking something, that's not your fault–it's theirs.

Perl believes in individual responsibility rather than mandated control. Perl respects you enough to let you choose your own preferred level of pain, or of pleasure. Perl believes that you are creative, intelligent, and capable of making your own decisions–and fully expects you to take complete responsibility for your own actions. In a perfect world, these admonitions alone would suffice, and everyone would be intelligent, responsible, happy, and creative. And careful. One probably shouldn't forget careful, and that's a good bit harder to expect. Even Einstein would take wrong turns by accident and end up lost in the wrong part of town.

Some folks get the heebie-jeebies when they see package variables hanging out there for anyone to reach over and alter them. Some folks live in constant fear that someone somewhere might do something wicked. The solution to that problem is simply to fire the wicked, of course. But unfortunately, it's not as simple as all that. These cautious types are also afraid that they or others will do something not so much wicked as careless, whether by accident or out of desperation. If we fire everyone who ever gets careless, pretty soon there won't be anybody left to get any work done.

Whether it's needless paranoia or sensible caution, this uneasiness can be a problem for some people. We can take the edge off their discomfort by providing the option of storing class attributes as lexical variables instead of as package variables. The my() operator is the source of all privacy in Perl, and it is a powerful form of privacy indeed.

It is widely perceived, and indeed has often been written, that Perl provides no data hiding, that it affords the class designer no privacy nor isolation, merely a rag-tag assortment of weak and unenforcible social conventions instead. This perception is demonstrably false and easily disproven. In the next section, we show how to implement forms of privacy that are far stronger than those provided in nearly any other object-oriented language.

10.4.2 File-Scoped Lexicals

A lexical variable is visible only through the end of its static scope. That means that the only code able to access that variable is code residing textually below the my() operator through the end of its block if it has one, or through the end of the current file if it doesn't.

Starting again with our simplest example given at the start of this document, we replace our() variables with my() versions.

```
package Some_Class;
my($CData1, $CData2);   # file scope, not in any package
sub CData1 {
    shift;  # XXX: ignore calling class/object
    $CData1 = shift if @_;
    return $CData1;
}
sub CData2 {
    shift;  # XXX: ignore calling class/object
    $CData2 = shift if @_;
    return $CData2;
}
```

So much for that old $Some_Class::CData1 package variable and its brethren! Those are gone now, replaced with lexicals. No one outside the scope can reach in and alter the class state without resorting to the documented interface. Not even subclasses or superclasses of this one have unmediated access to $CData1. They have to invoke the &CData1 method against Some_Class or an instance thereof, just like anybody else.

To be scrupulously honest, that last statement assumes you haven't packed several classes together into the same file scope, nor strewn your class implementation across several different files. Accessibility of those variables is based uniquely on the static file scope. It has nothing to do with the package. That means that code in a different file but the same package (class) could not access those variables, yet code in the same file but a different package (class) could. There are sound reasons why we usually suggest a one-to-one mapping between files and packages and modules and classes. You don't have to stick to this suggestion if you really know what you're doing, but you're apt to confuse yourself otherwise, especially at first.

If you'd like to aggregate your class attributes into one lexically scoped, composite structure, you're perfectly free to do so.

```
package Some_Class;
my %ClassData = (
    CData1 => "",
    CData2 => "",
);
sub CData1 {
    shift;  # XXX: ignore calling class/object
    $ClassData{CData1} = shift if @_;
```

```
        return $ClassData{CData1};
    }
    sub CData2 {
        shift;  # XXX: ignore calling class/object
        $ClassData{CData2} = shift if @_;
        return $ClassData{CData2};
    }
```

To make this more scalable as other class attributes are added, we can again register closures into the package symbol table to create accessor methods for them.

```
    package Some_Class;
    my %ClassData = (
        CData1 => "",
        CData2 => "",
    );
    for my $datum (keys %ClassData) {
        no strict "refs";
        *$datum = sub {
            shift;      # XXX: ignore calling class/object
            $ClassData{$datum} = shift if @_;
            return $ClassData{$datum};
        };
    }
```

Requiring even your own class to use accessor methods like anybody else is probably a good thing. But demanding and expecting that everyone else, be they subclass or superclass, friend or foe, will all come to your object through mediation is more than just a good idea. It's absolutely critical to the model. Let there be in your mind no such thing as "public" data, nor even "protected" data, which is a seductive but ultimately destructive notion. Both will come back to bite at you. That's because as soon as you take that first step out of the solid position in which all state is considered completely private, save from the perspective of its own accessor methods, you have violated the envelope. And, having pierced that encapsulating envelope, you shall doubtless someday pay the price when future changes in the implementation break unrelated code. Considering that avoiding this infelicitous outcome was precisely why you consented to suffer the slings and arrows of obsequious abstraction by turning to object orientation in the first place, such breakage seems unfortunate in the extreme.

10.4.3 More Inheritance Concerns

Suppose that Some_Class were used as a base class from which to derive Another_Class. If you invoke a &CData method on the derived class or on an object of that class, what do you get? Would the derived class have its own state, or would it piggyback on its base class's versions of the class attributes?

The answer is that under the scheme outlined above, the derived class would **not** have its own state data. As before, whether you consider this a good thing or a bad one depends on the semantics of the classes involved.

The cleanest, sanest, simplest way to address per-class state in a lexical is for the derived class to override its base class's version of the method that accesses the class attributes. Since the actual method called is the one in the object's derived class if this exists, you automatically get per-class state this way. Any urge to provide an unadvertised method to sneak out a reference to the %ClassData hash should be strenuously resisted.

As with any other overridden method, the implementation in the derived class always has the option of invoking its base class's version of the method in addition to its own. Here's an example:

```
    package Another_Class;
    @ISA = qw(Some_Class);

    my %ClassData = (
        CData1 => "",
    );
```

```
sub CData1 {
    my($self, $newvalue) = @_;
    if (@_ > 1) {
        # set locally first
        $ClassData{CData1} = $newvalue;

        # then pass the buck up to the first
        # overridden version, if there is one
        if ($self->can("SUPER::CData1")) {
            $self->SUPER::CData1($newvalue);
        }
    }
    return $ClassData{CData1};
}
```

Those dabbling in multiple inheritance might be concerned about there being more than one override.

```
for my $parent (@ISA) {
    my $methname = $parent . "::CData1";
    if ($self->can($methname)) {
        $self->$methname($newvalue);
    }
}
```

Because the &UNIVERSAL::can method returns a reference to the function directly, you can use this directly for a significant performance improvement:

```
for my $parent (@ISA) {
    if (my $coderef = $self->can($parent . "::CData1")) {
        $self->$coderef($newvalue);
    }
}
```

10.4.4 Locking the Door and Throwing Away the Key

As currently implemented, any code within the same scope as the file-scoped lexical %ClassData can alter that hash directly. Is that ok? Is it acceptable or even desirable to allow other parts of the implementation of this class to access class attributes directly?

That depends on how careful you want to be. Think back to the Cosmos class. If the &supernova method had directly altered $Cosmos::Stars or `$Cosmos::Cosmos{stars}`, then we wouldn't have been able to reuse the class when it came to inventing a Multiverse. So letting even the class itself access its own class attributes without the mediating intervention of properly designed accessor methods is probably not a good idea after all.

Restricting access to class attributes from the class itself is usually not enforcible even in strongly object-oriented languages. But in Perl, you can.

Here's one way:

```
package Some_Class;

{  # scope for hiding $CData1
    my $CData1;
    sub CData1 {
        shift;      # XXX: unused
        $CData1 = shift if @_;
        return $CData1;
    }
}
```

```
{  # scope for hiding $CData2
    my $CData2;
    sub CData2 {
        shift;      # XXX: unused
        $CData2 = shift if @_;
        return $CData2;
    }
}
```

No one–absolutely no one–is allowed to read or write the class attributes without the mediation of the managing accessor method, since only that method has access to the lexical variable it's managing. This use of mediated access to class attributes is a form of privacy far stronger than most OO languages provide.

The repetition of code used to create per-datum accessor methods chafes at our Laziness, so we'll again use closures to create similar methods.

```
package Some_Class;

{  # scope for ultra-private meta-object for class attributes
    my %ClassData = (
        CData1 => "",
        CData2 => "",
    );

    for my $datum (keys %ClassData ) {
        no strict "refs";
        *$datum = sub {
            use strict "refs";
            my ($self, $newvalue) = @_;
            $ClassData{$datum} = $newvalue if @_ > 1;
            return $ClassData{$datum};
        }
    }

}
```

The closure above can be modified to take inheritance into account using the &UNIVERSAL::can method and SUPER as shown previously.

10.4.5 Translucency Revisited

The Vermin class demonstrates translucency using a package variable, eponymously named %Vermin, as its meta-object. If you prefer to use absolutely no package variables beyond those necessary to appease inheritance or possibly the Exporter, this strategy is closed to you. That's too bad, because translucent attributes are an appealing technique, so it would be valuable to devise an implementation using only lexicals.

There's a second reason why you might wish to avoid the eponymous package hash. If you use class names with double-colons in them, you would end up poking around somewhere you might not have meant to poke.

```
package Vermin;
$class = "Vermin";
$class->{PopCount}++;
# accesses $Vermin::Vermin{PopCount}

package Vermin::Noxious;
$class = "Vermin::Noxious";
$class->{PopCount}++;
# accesses $Vermin::Noxious{PopCount}
```

In the first case, because the class name had no double-colons, we got the hash in the current package. But in the second case, instead of getting some hash in the current package, we got the hash %Noxious in the Vermin package. (The noxious vermin just invaded another package and sprayed their data around it. :-) Perl doesn't support relative packages in its naming conventions, so any double-colons trigger a fully-qualified lookup instead of just looking in the current package.

In practice, it is unlikely that the Vermin class had an existing package variable named %Noxious that you just blew away. If you're still mistrustful, you could always stake out your own territory where you know the rules, such as using Eponymous::Vermin::Noxious or Hieronymus::Vermin::Boschious or Leave_Me_Alone::Vermin::Noxious as class names instead. Sure, it's in theory possible that someone else has a class named Eponymous::Vermin with its own %Noxious hash, but this kind of thing is always true. There's no arbiter of package names. It's always the case that globals like @Cwd::ISA would collide if more than one class uses the same Cwd package.

If this still leaves you with an uncomfortable twinge of paranoia, we have another solution for you. There's nothing that says that you have to have a package variable to hold a class meta-object, either for monadic classes or for translucent attributes. Just code up the methods so that they access a lexical instead.

Here's another implementation of the Vermin class with semantics identical to those given previously, but this time using no package variables.

```
    package Vermin;

    # Here's the class meta-object, eponymously named.
    # It holds all class data, and also all instance data
    # so the latter can be used for both initialization
    # and translucency.  it's a template.
    my %ClassData = (
        PopCount => 0,          # capital for class attributes
        color    => "beige",    # small for instance attributes
    );

    # constructor method
    # invoked as class method or object method
    sub spawn {
        my $obclass = shift;
        my $class   = ref($obclass) || $obclass;
        my $self = {};
        bless($self, $class);
        $ClassData{PopCount}++;
        # init fields from invoking object, or omit if
        # invoking object is the class to provide translucency
        %$self = %$obclass if ref $obclass;
        return $self;
    }

    # translucent accessor for "color" attribute
    # invoked as class method or object method
    sub color {
        my $self  = shift;

        # handle class invocation
        unless (ref $self) {
            $ClassData{color} = shift if @_;
            return $ClassData{color}
        }
        # handle object invocation
        $self->{color} = shift if @_;
        if (defined $self->{color}) {  # not exists!
            return $self->{color};
        } else {
            return $ClassData{color};
        }
    }

    # class attribute accessor for "PopCount" attribute
    # invoked as class method or object method
    sub population {
        return $ClassData{PopCount};
    }

    # instance destructor; invoked only as object method
    sub DESTROY {
        $ClassData{PopCount}--;
    }

    # detect whether an object attribute is translucent
    # (typically?) invoked only as object method
    sub is_translucent {
        my($self, $attr)  = @_;
        $self = \%ClassData if !ref $self;
        return !defined $self->{$attr};
    }

    # test for presence of attribute in class
    # invoked as class method or object method
    sub has_attribute {
        my($self, $attr)  = @_;
        return exists $ClassData{$attr};
    }
```

10.5 NOTES

Inheritance is a powerful but subtle device, best used only after careful forethought and design. Aggregation instead of inheritance is often a better approach.

You can't use file-scoped lexicals in conjunction with the SelfLoader or the AutoLoader, because they alter the lexical scope in which the module's methods wind up getting compiled.

The usual mealy-mouthed package-mungeing doubtless applies to setting up names of object attributes. For example, `$self->{ObData1}` should probably be `$self->{ __PACKAGE__ .  "_ObData1" }`, but that would just confuse the examples.

10.6 SEE ALSO

perltoot, *perlobj*, *perlmod*, and *perlbot*.

The Tie::SecureHash and Class::Data::Inheritable modules from CPAN are worth checking out.

10.7 AUTHOR AND COPYRIGHT

10.8 ACKNOWLEDGEMENTS

Russ Allbery, Jon Orwant, Randy Ray, Larry Rosler, Nat Torkington, and Stephen Warren all contributed suggestions and corrections to this piece. Thanks especially to Damian Conway for his ideas and feedback, and without whose indirect prodding I might never have taken the time to show others how much Perl has to offer in the way of objects once you start thinking outside the tiny little box that today's "popular" object-oriented languages enforce.

10.9 HISTORY

Last edit: Sun Feb 4 20:50:28 EST 2001

Chapter 11

perlbot

Bag'o Object Tricks (the BOT)

11.1 DESCRIPTION

The following collection of tricks and hints is intended to whet curious appetites about such things as the use of instance variables and the mechanics of object and class relationships. The reader is encouraged to consult relevant textbooks for discussion of Object Oriented definitions and methodology. This is not intended as a tutorial for object-oriented programming or as a comprehensive guide to Perl's object oriented features, nor should it be construed as a style guide. If you're looking for tutorials, be sure to read *perlboot*, *perltoot*, and *perltooc*.

The Perl motto still holds: There's more than one way to do it.

11.2 OO SCALING TIPS

1. Do not attempt to verify the type of $self. That'll break if the class is inherited, when the type of $self is valid but its package isn't what you expect. See rule 5.

2. If an object-oriented (OO) or indirect-object (IO) syntax was used, then the object is probably the correct type and there's no need to become paranoid about it. Perl isn't a paranoid language anyway. If people subvert the OO or IO syntax then they probably know what they're doing and you should let them do it. See rule 1.

3. Use the two-argument form of bless(). Let a subclass use your constructor. See INHERITING A CONSTRUCTOR.

4. The subclass is allowed to know things about its immediate superclass, the superclass is allowed to know nothing about a subclass.

5. Don't be trigger happy with inheritance. A "using", "containing", or "delegation" relationship (some sort of aggregation, at least) is often more appropriate. See OBJECT RELATIONSHIPS, USING RELATIONSHIP WITH SDBM, and §11.12.

6. The object is the namespace. Make package globals accessible via the object. This will remove the guess work about the symbol's home package. See CLASS CONTEXT AND THE OBJECT.

7. IO syntax is certainly less noisy, but it is also prone to ambiguities that can cause difficult-to-find bugs. Allow people to use the sure-thing OO syntax, even if you don't like it.

8. Do not use function-call syntax on a method. You're going to be bitten someday. Someone might move that method into a superclass and your code will be broken. On top of that you're feeding the paranoia in rule 2.

9. Don't assume you know the home package of a method. You're making it difficult for someone to override that method. See THINKING OF CODE REUSE.

11.3 INSTANCE VARIABLES

An anonymous array or anonymous hash can be used to hold instance variables. Named parameters are also demonstrated.

```
        package Foo;

        sub new {
                my $type = shift;
                my %params = @_;
                my $self = {};
                $self->{'High'} = $params{'High'};
                $self->{'Low'}  = $params{'Low'};
                bless $self, $type;
        }

        package Bar;

        sub new {
                my $type = shift;
                my %params = @_;
                my $self = [];
                $self->[0] = $params{'Left'};
                $self->[1] = $params{'Right'};
                bless $self, $type;
        }

        package main;

        $a = Foo->new( 'High' => 42, 'Low' => 11 );
        print "High=$a->{'High'}\n";
        print "Low=$a->{'Low'}\n";

        $b = Bar->new( 'Left' => 78, 'Right' => 40 );
        print "Left=$b->[0]\n";
        print "Right=$b->[1]\n";
```

11.4 SCALAR INSTANCE VARIABLES

An anonymous scalar can be used when only one instance variable is needed.

```
        package Foo;

        sub new {
                my $type = shift;
                my $self;
                $self = shift;
                bless \$self, $type;
        }

        package main;

        $a = Foo->new( 42 );
        print "a=$$a\n";
```

11.5 INSTANCE VARIABLE INHERITANCE

This example demonstrates how one might inherit instance variables from a superclass for inclusion in the new class. This requires calling the superclass's constructor and adding one's own instance variables to the new object.

```
package Bar;

sub new {
        my $type = shift;
        my $self = {};
        $self->{'buz'} = 42;
        bless $self, $type;
}

package Foo;
@ISA = qw( Bar );

sub new {
        my $type = shift;
        my $self = Bar->new;
        $self->{'biz'} = 11;
        bless $self, $type;
}

package main;

$a = Foo->new;
print "buz = ", $a->{'buz'}, "\n";
print "biz = ", $a->{'biz'}, "\n";
```

11.6 OBJECT RELATIONSHIPS

The following demonstrates how one might implement "containing" and "using" relationships between objects.

```
package Bar;

sub new {
        my $type = shift;
        my $self = {};
        $self->{'buz'} = 42;
        bless $self, $type;
}

package Foo;

sub new {
        my $type = shift;
        my $self = {};
        $self->{'Bar'} = Bar->new;
        $self->{'biz'} = 11;
        bless $self, $type;
}

package main;

$a = Foo->new;
print "buz = ", $a->{'Bar'}->{'buz'}, "\n";
print "biz = ", $a->{'biz'}, "\n";
```

11.7 OVERRIDING SUPERCLASS METHODS

The following example demonstrates how to override a superclass method and then call the overridden method. The **SUPER** pseudo-class allows the programmer to call an overridden superclass method without actually knowing where that method is defined.

```
package Buz;
sub goo { print "here’s the goo\n" }

package Bar; @ISA = qw( Buz );
sub google { print "google here\n" }

package Baz;
sub mumble { print "mumbling\n" }

package Foo;
@ISA = qw( Bar Baz );

sub new {
        my $type = shift;
        bless [], $type;
}
sub grr { print "grumble\n" }
sub goo {
        my $self = shift;
        $self->SUPER::goo();
}
sub mumble {
        my $self = shift;
        $self->SUPER::mumble();
}
sub google {
        my $self = shift;
        $self->SUPER::google();
}

package main;

$foo = Foo->new;
$foo->mumble;
$foo->grr;
$foo->goo;
$foo->google;
```

Note that `SUPER` refers to the superclasses of the current package (`Foo`), not to the superclasses of `$self`.

11.8 USING RELATIONSHIP WITH SDBM

This example demonstrates an interface for the SDBM class. This creates a "using" relationship between the SDBM class and the new class Mydbm.

```
package Mydbm;
```

```
        require SDBM_File;
        require Tie::Hash;
        @ISA = qw( Tie::Hash );

        sub TIEHASH {
            my $type = shift;
            my $ref  = SDBM_File->new(@_);
            bless {'dbm' => $ref}, $type;
        }
        sub FETCH {
            my $self = shift;
            my $ref  = $self->{'dbm'};
            $ref->FETCH(@_);
        }
        sub STORE {
            my $self = shift;
            if (defined $_[0]){
                my $ref = $self->{'dbm'};
                $ref->STORE(@_);
            } else {
                die "Cannot STORE an undefined key in Mydbm\n";
            }
        }

        package main;
        use Fcntl qw( O_RDWR O_CREAT );

        tie %foo, "Mydbm", "Sdbm", O_RDWR|O_CREAT, 0640;
        $foo{'bar'} = 123;
        print "foo-bar = $foo{'bar'}\n";

        tie %bar, "Mydbm", "Sdbm2", O_RDWR|O_CREAT, 0640;
        $bar{'Cathy'} = 456;
        print "bar-Cathy = $bar{'Cathy'}\n";
```

11.9 THINKING OF CODE REUSE

One strength of Object-Oriented languages is the ease with which old code can use new code. The following examples will demonstrate first how one can hinder code reuse and then how one can promote code reuse.

This first example illustrates a class which uses a fully-qualified method call to access the "private" method BAZ(). The second example will show that it is impossible to override the BAZ() method.

```
        package FOO;

        sub new {
                my $type = shift;
                bless {}, $type;
        }
        sub bar {
                my $self = shift;
                $self->FOO::private::BAZ;
        }

        package FOO::private;
```

```
    sub BAZ {
            print "in BAZ\n";
    }

    package main;

    $a = FOO->new;
    $a->bar;
```

Now we try to override the BAZ() method. We would like FOO::bar() to call GOOP::BAZ(), but this cannot happen because FOO::bar() explicitly calls FOO::private::BAZ().

```
    package FOO;

    sub new {
            my $type = shift;
            bless {}, $type;
    }
    sub bar {
            my $self = shift;
            $self->FOO::private::BAZ;
    }

    package FOO::private;

    sub BAZ {
            print "in BAZ\n";
    }

    package GOOP;
    @ISA = qw( FOO );
    sub new {
            my $type = shift;
            bless {}, $type;
    }

    sub BAZ {
            print "in GOOP::BAZ\n";
    }

    package main;

    $a = GOOP->new;
    $a->bar;
```

To create reusable code we must modify class FOO, flattening class FOO::private. The next example shows a reusable class FOO which allows the method GOOP::BAZ() to be used in place of FOO::BAZ().

```
    package FOO;

    sub new {
            my $type = shift;
            bless {}, $type;
    }
    sub bar {
            my $self = shift;
            $self->BAZ;
    }
```

```
        sub BAZ {
                print "in BAZ\n";
        }

        package GOOP;
        @ISA = qw( FOO );

        sub new {
                my $type = shift;
                bless {}, $type;
        }
        sub BAZ {
                print "in GOOP::BAZ\n";
        }

        package main;

        $a = GOOP->new;
        $a->bar;
```

11.10 CLASS CONTEXT AND THE OBJECT

Use the object to solve package and class context problems. Everything a method needs should be available via the object or should be passed as a parameter to the method.

A class will sometimes have static or global data to be used by the methods. A subclass may want to override that data and replace it with new data. When this happens the superclass may not know how to find the new copy of the data.

This problem can be solved by using the object to define the context of the method. Let the method look in the object for a reference to the data. The alternative is to force the method to go hunting for the data ("Is it in my class, or in a subclass? Which subclass?"), and this can be inconvenient and will lead to hackery. It is better just to let the object tell the method where that data is located.

```
        package Bar;

        %fizzle = ( 'Password' => 'XYZZY' );

        sub new {
                my $type = shift;
                my $self = {};
                $self->{'fizzle'} = \%fizzle;
                bless $self, $type;
        }

        sub enter {
                my $self = shift;

                # Don't try to guess if we should use %Bar::fizzle
                # or %Foo::fizzle.  The object already knows which
                # we should use, so just ask it.
                #
                my $fizzle = $self->{'fizzle'};

                print "The word is ", $fizzle->{'Password'}, "\n";
        }
        package Foo;
        @ISA = qw( Bar );

        %fizzle = ( 'Password' => 'Rumple' );

        sub new {
                my $type = shift;
                my $self = Bar->new;
                $self->{'fizzle'} = \%fizzle;
                bless $self, $type;
        }

        package main;

        $a = Bar->new;
        $b = Foo->new;
        $a->enter;
        $b->enter;
```

11.11 INHERITING A CONSTRUCTOR

An inheritable constructor should use the second form of bless() which allows blessing directly into a specified class. Notice in this example that the object will be a BAR not a FOO, even though the constructor is in class FOO.

```
        package FOO;

        sub new {
                my $type = shift;
                my $self = {};
                bless $self, $type;
        }

        sub baz {
                print "in FOO::baz()\n";
        }

        package BAR;
        @ISA = qw(FOO);

        sub baz {
                print "in BAR::baz()\n";
        }

        package main;

        $a = BAR->new;
        $a->baz;
```

11.12 DELEGATION

Some classes, such as SDBM_File, cannot be effectively subclassed because they create foreign objects. Such a class can be extended with some sort of aggregation technique such as the "using" relationship mentioned earlier or by delegation.

The following example demonstrates delegation using an AUTOLOAD() function to perform message-forwarding. This will allow the Mydbm object to behave exactly like an SDBM_File object. The Mydbm class could now extend the behavior by adding custom FETCH() and STORE() methods, if this is desired.

```
        package Mydbm;

        require SDBM_File;
        require Tie::Hash;
        @ISA = qw(Tie::Hash);

        sub TIEHASH {
                my $type = shift;
                my $ref = SDBM_File->new(@_);
                bless {'delegate' => $ref};
        }

        sub AUTOLOAD {
                my $self = shift;

                # The Perl interpreter places the name of the
                # message in a variable called $AUTOLOAD.

                # DESTROY messages should never be propagated.
                return if $AUTOLOAD =~ /::DESTROY$/;

                # Remove the package name.
                $AUTOLOAD =~ s/^Mydbm:://;

                # Pass the message to the delegate.
                $self->{'delegate'}->$AUTOLOAD(@_);
        }

        package main;
        use Fcntl qw( O_RDWR O_CREAT );

        tie %foo, "Mydbm", "adbm", O_RDWR|O_CREAT, 0640;
        $foo{'bar'} = 123;
        print "foo-bar = $foo{'bar'}\n";
```

11.13 SEE ALSO

perlboot, *perltoot*, *perltooc*.

Chapter 12

perlstyle

Perl style guide

12.1 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. 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.
- Be consistent.
- Be nice.

Chapter 13

perlcheat

Perl 5 Cheat Sheet

13.1 DESCRIPTION

This 'cheat sheet' is a handy reference, meant for beginning Perl programmers. Not everything is mentioned, but 194 features may already be overwhelming.

13.1.1 The sheet

```
  CONTEXTS  SIGILS             ARRAYS        HASHES
  void      $scalar   whole:   @array        %hash
  scalar    @array    slice:   @array[0, 2]  @hash{'a', 'b'}
  list      %hash     element: $array[0]     $hash{'a'}
            &sub
            *glob    SCALAR VALUES
                     number, string, reference, glob, undef
  REFERENCES
  \     references      $$foo[1]       aka $foo->[1]
  $@%&* dereference     $$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
                          NUMBERS vs STRINGS  LINKS
  OPERATOR PRECEDENCE     =          =        perl.plover.com
  ->                      +          .        search.cpan.org
  ++ --                   == !=      eq ne         cpan.org
  **                      < > <= >=  lt gt le ge   pm.org
  ! ~ \ u+ u-             <=>        cmp           tpj.com
  =~ !~                                            perldoc.com
  * / % x                 SYNTAX
  + - .                   for    (LIST) { }, for (a;b;c) { }
  << >>                   while  ( ) { }, until ( ) { }
  named uops              if     ( ) { } elsif ( ) { } else { }
  < > <= >= lt gt le ge   unless ( ) { } elsif ( ) { } else { }
  == != <=> eq ne cmp     for equals foreach (ALWAYS)
  &
  | ^              REGEX METACHARS            REGEX MODIFIERS
  &&               ^     string begin         /i case insens.
  ||               $     str. end (before \n) /m line based ^$
```

```
.. ...            +     one or more          /s . includes \n
?:                *     zero or more         /x ign. wh.space
= += -= *= etc.   ?     zero or one          /g global
, =>              {3,7} repeat in range
list ops          ()    capture          REGEX CHARCLASSES
not               (?:)  no capture       .  == [^\n]
and               []    character class  \s == [\x20\f\t\r\n]
or xor            |     alternation      \w == [A-Za-z0-9_]
                  \b    word boundary    \d == [0-9]
                  \z    string end       \S, \W and \D negate
DO
use strict;        DON'T            LINKS
use warnings;      "$foo"           perl.com
my $var;           $$variable_name  perlmonks.org
open() or die $!;  `$userinput`     use.perl.org
use Modules;       /$userinput/     perl.apache.org
                                    parrotcode.org
FUNCTION RETURN LISTS
stat      localtime    caller         SPECIAL VARIABLES
 0 dev    0 second     0 package      $_    default variable
 1 ino    1 minute     1 filename     $0    program name
 2 mode   2 hour       2 line         $/    input separator
 3 nlink  3 day        3 subroutine   $\    output separator
 4 uid    4 month-1    4 hasargs      $|    autoflush
 5 gid    5 year-1900  5 wantarray    $!    sys/libcall error
 6 rdev   6 weekday    6 evaltext     $@    eval error
 7 size   7 yearday    7 is_require   $$    process ID
 8 atime  8 is_dst     8 hints        $.    line number
 9 mtime               9 bitmask      @ARGV command line args
10 ctime  just use                    @INC  include paths
11 blksz  POSIX::      3..9 only      @_    subroutine args
12 blcks  strftime!    with EXPR      %ENV  environment
```

13.2 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.

13.3 AUTHOR

Juerd Waalboer <juerd@cpan.org>, with the help of many Perl Monks.

13.4 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
```

Chapter 14

perltrap

Perl traps for the unwary

14.1 DESCRIPTION

The biggest trap of all is forgetting to `use warnings` or use the **-w** switch; see *perllexwarn* and *perlrun*. 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*.

14.1.1 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 $RS), as though they were in **awk**; see *perlvar* for details.
- Semicolons are required after all simple statements in Perl (except at the end of a block). Newline is not a statement delimiter.
- Curly brackets are required on `if`s and `while`s.
- Variables begin with "$", "@" 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 *perlvar*.

- $<*digit*> does not refer to fields–it refers to substrings matched by the last match pattern.
- The print() statement does not add field and record separators unless you set `$,` 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.

14.1.2 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 Loop Control in *perlsyn*.
- There's no switch statement. (But it's easy to build one on the fly, see Basic BLOCKs and Switch Statements in *perlsyn*)
- 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.

14.1.3 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.

14.1.4 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.

14.1.5 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 *perldata* 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 *perlop* and *perlsub*.
- 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 *perlform* 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.

14.1.6 Perl4 to Perl5 Traps

Practicing Perl4 Programmers should take note of the following Perl4-to-Perl5 specific traps.
They're crudely ordered according to the following list:

Discontinuance, Deprecation, and BugFix traps

Anything that's been fixed as a perl4 bug, removed as a perl4 feature or deprecated as a perl4 feature with the intent to encourage usage of some other perl5 feature.

Parsing Traps

Traps that appear to stem from the new parser.

Numerical Traps

Traps having to do with numerical or mathematical operators.

General data type traps

Traps involving perl standard data types.

Context Traps - scalar, list contexts

Traps related to context within lists, scalar statements/declarations.

Precedence Traps

Traps related to the precedence of parsing, evaluation, and execution of code.

General Regular Expression Traps using s///, etc.

Traps related to the use of pattern matching.

Subroutine, Signal, Sorting Traps

Traps related to the use of signals and signal handlers, general subroutines, and sorting, along with sorting subroutines.

OS Traps

OS-specific traps.

DBM Traps

Traps specific to the use of `dbmopen()`, and specific dbm implementations.

Unclassified Traps

Everything else.

If you find an example of a conversion trap that is not listed here, please submit it to <*perlbug@perl.org*> for inclusion. Also note that at least some of these can be caught with the `use warnings` pragma or the **-w** switch.

14.1.7 Discontinuance, Deprecation, and BugFix traps

Anything that has been discontinued, deprecated, or fixed as a bug from perl4.

- Discontinuance

  Symbols starting with "_" are no longer forced into package main, except for `$_` itself (and `@_`, etc.).

  ```
  package test;
  $_legacy = 1;

  package main;
  print "\$_legacy is ",$_legacy,"\n";

  # perl4 prints: $_legacy is 1
  # perl5 prints: $_legacy is
  ```

- Deprecation

  Double-colon is now a valid package separator in a variable name. Thus these behave differently in perl4 vs. perl5, because the packages don't exist.

  ```
  $a=1;$b=2;$c=3;$var=4;
  print "$a::$b::$c ";
  print "$var::abc::xyz\n";

  # perl4 prints: 1::2::3 4::abc::xyz
  # perl5 prints: 3
  ```

  Given that `::` is now the preferred package delimiter, it is debatable whether this should be classed as a bug or not. (The older package delimiter, ' ,is used here)

  ```
  $x = 10 ;
  print "x=${'x}\n" ;

  # perl4 prints: x=10
  # perl5 prints: Can't find string terminator "'" anywhere before EOF
  ```

  You can avoid this problem, and remain compatible with perl4, if you always explicitly include the package name:

  ```
  $x = 10 ;
  print "x=${main'x}\n" ;
  ```

  Also see precedence traps, for parsing `$:`.

- BugFix

  The second and third arguments of `splice()` are now evaluated in scalar context (as the Camel says) rather than list context.

  ```
  sub sub1{return(0,2) }          # return a 2-element list
  sub sub2{ return(1,2,3)}        # return a 3-element list
  @a1 = ("a","b","c","d","e");
  @a2 = splice(@a1,&sub1,&sub2);
  print join(' ',@a2),"\n";

  # perl4 prints: a b
  # perl5 prints: c d e
  ```

- Discontinuance

  You can't do a `goto` into a block that is optimized away. Darn.

  ```
  goto marker1;

  for(1){
  marker1:
      print "Here I is!\n";
  }

  # perl4 prints: Here I is!
  # perl5 errors: Can't "goto" into the middle of a foreach loop
  ```

- Discontinuance

  It is no longer syntactically legal to use whitespace as the name of a variable, or as a delimiter for any kind of quote construct. Double darn.

  ```
  $a = ("foo bar");
  $b = q baz ;
  print "a is $a, b is $b\n";

  # perl4 prints: a is foo bar, b is baz
  # perl5 errors: Bareword found where operator expected
  ```

- Discontinuance

  The archaic while/if BLOCK BLOCK syntax is no longer supported.

  ```
  if { 1 } {
      print "True!";
  }
  else {
      print "False!";
  }

  # perl4 prints: True!
  # perl5 errors: syntax error at test.pl line 1, near "if {"
  ```

- BugFix

  The ** operator now binds more tightly than unary minus. It was documented to work this way before, but didn't.

  ```
  print -4**2,"\n";

  # perl4 prints: 16
  # perl5 prints: -16
  ```

- Discontinuance

  The meaning of `foreach{}` has changed slightly when it is iterating over a list which is not an array. This used to assign the list to a temporary array, but no longer does so (for efficiency). This means that you'll now be iterating over the actual values, not over copies of the values. Modifications to the loop variable can change the original values.

```
@list = ('ab','abc','bcd','def');
foreach $var (grep(/ab/,@list)){
    $var = 1;
}
print (join(':',@list));

# perl4 prints: ab:abc:bcd:def
# perl5 prints: 1:1:bcd:def
```

To retain Perl4 semantics you need to assign your list explicitly to a temporary array and then iterate over that. For example, you might need to change

```
foreach $var (grep(/ab/,@list)){
```

to

```
foreach $var (@tmp = grep(/ab/,@list)){
```

Otherwise changing $var will clobber the values of @list. (This most often happens when you use `$_` for the loop variable, and call subroutines in the loop that don't properly localize `$_`.)

- Discontinuance

  `split` with no arguments now behaves like `split ' '` (which doesn't return an initial null field if $_ starts with whitespace), it used to behave like `split /\s+/` (which does).

  ```
  $_ = ' hi mom';
  print join(':', split);

  # perl4 prints: :hi:mom
  # perl5 prints: hi:mom
  ```

- BugFix

  Perl 4 would ignore any text which was attached to an **-e** switch, always taking the code snippet from the following arg. Additionally, it would silently accept an **-e** switch without a following arg. Both of these behaviors have been fixed.

  ```
  perl -e'print "attached to -e"' 'print "separate arg"'

  # perl4 prints: separate arg
  # perl5 prints: attached to -e

  perl -e

  # perl4 prints:
  # perl5 dies: No code specified for -e.
  ```

- Discontinuance

  In Perl 4 the return value of `push` was undocumented, but it was actually the last value being pushed onto the target list. In Perl 5 the return value of `push` is documented, but has changed, it is the number of elements in the resulting list.

  ```
  @x = ('existing');
  print push(@x, 'first new', 'second new');
  ```

```
    # perl4 prints: second new
    # perl5 prints: 3
```

- Deprecation

  Some error messages will be different.

- Discontinuance

  In Perl 4, if in list context the delimiters to the first argument of `split()` were `??`, the result would be placed in `@_` as well as being returned. Perl 5 has more respect for your subroutine arguments.

- Discontinuance

  Some bugs may have been inadvertently removed. :-)

14.1.8 Parsing Traps

Perl4-to-Perl5 traps from having to do with parsing.

- Parsing

  Note the space between . and =

  ```
      $string . = "more string";
      print $string;

      # perl4 prints: more string
      # perl5 prints: syntax error at - line 1, near ". ="
  ```

- Parsing

  Better parsing in perl 5

  ```
      sub foo {}
      &foo
      print("hello, world\n");

      # perl4 prints: hello, world
      # perl5 prints: syntax error
  ```

- Parsing

  "if it looks like a function, it is a function" rule.

  ```
    print
      ($foo == 1) ? "is one\n" : "is zero\n";

      # perl4 prints: is zero
      # perl5 warns: "Useless use of a constant in void context" if using -w
  ```

- Parsing

  String interpolation of the `$#array` construct differs when braces are to used around the name.

  ```
      @a = (1..3);
      print "${#a}";

      # perl4 prints: 2
      # perl5 fails with syntax error
  ```

```
@ = (1..3);
print "$#{a}";

# perl4 prints: {a}
# perl5 prints: 2
```

- Parsing

  When perl sees `map {` (or `grep {`), it has to guess whether the `{` starts a BLOCK or a hash reference. If it guesses wrong, it will report a syntax error near the `}` and the missing (or unexpected) comma.

  Use unary + before `{` on a hash reference, and unary + applied to the first thing in a BLOCK (after `{`), for perl to guess right all the time. (See map in *perlfunc*.)

14.1.9 Numerical Traps

Perl4-to-Perl5 traps having to do with numerical operators, operands, or output from same.

- Numerical

  Formatted output and significant digits. In general, Perl 5 tries to be more precise. For example, on a Solaris Sparc:

  ```
  print 7.373504 - 0, "\n";
  printf "%20.18f\n", 7.373504 - 0;

  # Perl4 prints:
  7.3750399999999996141
  7.375039999999999614

  # Perl5 prints:
  7.373504
  7.375039999999999614
  ```

  Notice how the first result looks better in Perl 5.

  Your results may vary, since your floating point formatting routines and even floating point format may be slightly different.

- Numerical

  This specific item has been deleted. It demonstrated how the auto-increment operator would not catch when a number went over the signed int limit. Fixed in version 5.003_04. But always be wary when using large integers. If in doubt:

  ```
  use Math::BigInt;
  ```

- Numerical

  Assignment of return values from numeric equality tests does not work in perl5 when the test evaluates to false (0). Logical tests now return a null, instead of 0

  ```
  $p = ($test == 1);
  print $p,"\n";

  # perl4 prints: 0
  # perl5 prints:
  ```

  Also see §14.1.13 for another example of this new feature...

- Bitwise string ops

  When bitwise operators which can operate upon either numbers or strings (`& | ^ ~`) are given only strings as arguments, perl4 would treat the operands as bitstrings so long as the program contained a call to the `vec()` function. perl5 treats the string operands as bitstrings. (See Bitwise String Operators in *perlop* for more details.)

  ```
  $fred = "10";
  $barney = "12";
  $betty = $fred & $barney;
  print "$betty\n";
  # Uncomment the next line to change perl4's behavior
  # ($dummy) = vec("dummy", 0, 0);

  # Perl4 prints:
  8

  # Perl5 prints:
  10

  # If vec() is used anywhere in the program, both print:
  10
  ```

14.1.10 General data type traps

Perl4-to-Perl5 traps involving most data-types, and their usage within certain expressions and/or context.

- (Arrays)

  Negative array subscripts now count from the end of the array.

  ```
  @a = (1, 2, 3, 4, 5);
  print "The third element of the array is $a[3] also expressed as $a[-2] \n";

  # perl4 prints: The third element of the array is 4 also expressed as
  # perl5 prints: The third element of the array is 4 also expressed as 4
  ```

- (Arrays)

  Setting `$#array` lower now discards array elements, and makes them impossible to recover.

  ```
  @a = (a,b,c,d,e);
  print "Before: ",join('',@a);
  $#a =1;
  print ", After: ",join('',@a);
  $#a =3;
  print ", Recovered: ",join('',@a),"\n";

  # perl4 prints: Before: abcde, After: ab, Recovered: abcd
  # perl5 prints: Before: abcde, After: ab, Recovered: ab
  ```

- (Hashes)

  Hashes get defined before use

  ```
  local($s,@a,%h);
  die "scalar \$s defined" if defined($s);
  die "array \@a defined" if defined(@a);
  die "hash \%h defined" if defined(%h);
  ```

```
    # perl4 prints:
    # perl5 dies: hash %h defined
```

Perl will now generate a warning when it sees defined(@a) and defined(%h).

- (Globs)

  glob assignment from variable to variable will fail if the assigned variable is localized subsequent to the assignment

  ```
  @a = ("This is Perl 4");
  *b = *a;
  local(@a);
  print @b,"\n";

  # perl4 prints: This is Perl 4
  # perl5 prints:
  ```

- (Globs)

  Assigning `undef` to a glob has no effect in Perl 5. In Perl 4 it undefines the associated scalar (but may have other side effects including SEGVs). Perl 5 will also warn if `undef` is assigned to a typeglob. (Note that assigning `undef` to a typeglob is different than calling the `undef` function on a typeglob (`undef *foo`), which has quite a few effects.

  ```
  $foo = "bar";
  *foo = undef;
  print $foo;

  # perl4 prints:
  # perl4 warns: "Use of uninitialized variable" if using -w
  # perl5 prints: bar
  # perl5 warns: "Undefined value assigned to typeglob" if using -w
  ```

- (Scalar String)

  Changes in unary negation (of strings) This change effects both the return value and what it does to auto(magic)increment.

  ```
  $x = "aaa";
  print ++$x," : ";
  print -$x," : ";
  print ++$x,"\n";

  # perl4 prints: aab : -0 : 1
  # perl5 prints: aab : -aab : aac
  ```

- (Constants)

  perl 4 lets you modify constants:

  ```
  $foo = "x";
  &mod($foo);
  for ($x = 0; $x < 3; $x++) {
      &mod("a");
  }
  sub mod {
      print "before: $_[0]";
      $_[0] = "m";
      print "  after: $_[0]\n";
  }
  ```

```
# perl4:
# before: x  after: m
# before: a  after: m
# before: m  after: m
# before: m  after: m

# Perl5:
# before: x  after: m
# Modification of a read-only value attempted at foo.pl line 12.
# before: a
```

- (Scalars)

  The behavior is slightly different for:

```
print "$x", defined $x

# perl 4: 1
# perl 5: <no output, $x is not called into existence>
```

- (Variable Suicide)

  Variable suicide behavior is more consistent under Perl 5. Perl5 exhibits the same behavior for hashes and scalars, that perl4 exhibits for only scalars.

```
$aGlobal{ "aKey" } = "global value";
print "MAIN:", $aGlobal{"aKey"}, "\n";
$GlobalLevel = 0;
&test( *aGlobal );

sub test {
    local( *theArgument ) = @_;
    local( %aNewLocal ); # perl 4 != 5.001l,m
    $aNewLocal{"aKey"} = "this should never appear";
    print "SUB: ", $theArgument{"aKey"}, "\n";
    $aNewLocal{"aKey"} = "level $GlobalLevel";   # what should print
    $GlobalLevel++;
    if( $GlobalLevel<4 ) {
        &test( *aNewLocal );
    }
}

# Perl4:
# MAIN:global value
# SUB: global value
# SUB: level 0
# SUB: level 1
# SUB: level 2

# Perl5:
# MAIN:global value
# SUB: global value
# SUB: this should never appear
# SUB: this should never appear
# SUB: this should never appear
```

14.1.11 Context Traps - scalar, list contexts

- (list context)

  The elements of argument lists for formats are now evaluated in list context. This means you can interpolate list values now.

  ```
  @fmt = ("foo","bar","baz");
  format STDOUT=
  @<<<<< @||||| @>>>>>
  @fmt;
  .
  write;

  # perl4 errors:  Please use commas to separate fields in file
  # perl5 prints: foo     bar      baz
  ```

- (scalar context)

  The `caller()` function now returns a false value in a scalar context if there is no caller. This lets library files determine if they're being required.

  ```
  caller() ? (print "You rang?\n") : (print "Got a 0\n");

  # perl4 errors: There is no caller
  # perl5 prints: Got a 0
  ```

- (scalar context)

  The comma operator in a scalar context is now guaranteed to give a scalar context to its arguments.

  ```
  @y= ('a','b','c');
  $x = (1, 2, @y);
  print "x = $x\n";

  # Perl4 prints:  x = c   # Thinks list context interpolates list
  # Perl5 prints:  x = 3   # Knows scalar uses length of list
  ```

- (list, builtin)

  `sprintf()` is prototyped as ($;@), so its first argument is given scalar context. Thus, if passed an array, it will probably not do what you want, unlike Perl 4:

  ```
  @z = ('%s%s', 'foo', 'bar');
  $x = sprintf(@z);
  print $x;

  # perl4 prints: foobar
  # perl5 prints: 3
  ```

  `printf()` works the same as it did in Perl 4, though:

  ```
  @z = ('%s%s', 'foo', 'bar');
  printf STDOUT (@z);

  # perl4 prints: foobar
  # perl5 prints: foobar
  ```

14.1.12 Precedence Traps

Perl4-to-Perl5 traps involving precedence order.

Perl 4 has almost the same precedence rules as Perl 5 for the operators that they both have. Perl 4 however, seems to have had some inconsistencies that made the behavior differ from what was documented.

- Precedence

  LHS vs. RHS of any assignment operator. LHS is evaluated first in perl4, second in perl5; this can affect the relationship between side-effects in sub-expressions.

  ```
  @arr = ( 'left', 'right' );
  $a{shift @arr} = shift @arr;
  print join( ' ', keys %a );

  # perl4 prints: left
  # perl5 prints: right
  ```

- Precedence

  These are now semantic errors because of precedence:

  ```
  @list = (1,2,3,4,5);
  %map = ("a",1,"b",2,"c",3,"d",4);
  $n = shift @list + 2;   # first item in list plus 2
  print "n is $n, ";
  $m = keys %map + 2;     # number of items in hash plus 2
  print "m is $m\n";

  # perl4 prints: n is 3, m is 6
  # perl5 errors and fails to compile
  ```

- Precedence

  The precedence of assignment operators is now the same as the precedence of assignment. Perl 4 mistakenly gave them the precedence of the associated operator. So you now must parenthesize them in expressions like

  ```
  /foo/ ? ($a += 2) : ($a -= 2);
  ```

  Otherwise

  ```
  /foo/ ? $a += 2 : $a -= 2
  ```

  would be erroneously parsed as

  ```
  (/foo/ ? $a += 2 : $a) -= 2;
  ```

  On the other hand,

  ```
  $a += /foo/ ? 1 : 2;
  ```

  now works as a C programmer would expect.

- Precedence

  ```
  open FOO || die;
  ```

is now incorrect. You need parentheses around the filehandle. Otherwise, perl5 leaves the statement as its default precedence:

```
open(FOO || die);

# perl4 opens or dies
# perl5 opens FOO, dying only if 'FOO' is false, i.e. never
```

- Precedence

  perl4 gives the special variable, `$:` precedence, where perl5 treats `$::` as main `package`

```
$a = "x"; print "$::a";

# perl 4 prints: -:a
# perl 5 prints: x
```

- Precedence

  perl4 had buggy precedence for the file test operators vis-a-vis the assignment operators. Thus, although the precedence table for perl4 leads one to believe `-e $foo .= "q"` should parse as `((-e $foo) .= "q")`, it actually parses as `(-e ($foo .= "q"))`. In perl5, the precedence is as documented.

```
-e $foo .= "q"

# perl4 prints: no output
# perl5 prints: Can't modify -e in concatenation
```

- Precedence

  In perl4, keys(), each() and values() were special high-precedence operators that operated on a single hash, but in perl5, they are regular named unary operators. As documented, named unary operators have lower precedence than the arithmetic and concatenation operators `+ - .`, but the perl4 variants of these operators actually bind tighter than `+ - .`. Thus, for:

```
%foo = 1..10;
print keys %foo - 1

# perl4 prints: 4
# perl5 prints: Type of arg 1 to keys must be hash (not subtraction)
```

  The perl4 behavior was probably more useful, if less consistent.

14.1.13 General Regular Expression Traps using s///, etc.

All types of RE traps.

- Regular Expression

  `s'$lhs'$rhs'` now does no interpolation on either side. It used to interpolate $lhs but not $rhs. (And still does not match a literal '$' in string)

```
$a=1;$b=2;
$string = '1 2 $a $b';
$string =~ s'$a'$b';
print $string,"\n";
```

```
    # perl4 prints: $b 2 $a $b
    # perl5 prints: 1 2 $a $b
```

- Regular Expression

  m//g now attaches its state to the searched string rather than the regular expression. (Once the scope of a block is left for the sub, the state of the searched string is lost)

```
    $_ = "ababab";
    while(m/ab/g){
        &doit("blah");
    }
    sub doit{local($_) = shift; print "Got $_ "}

    # perl4 prints: Got blah Got blah Got blah Got blah
    # perl5 prints: infinite loop blah...
```

- Regular Expression

  Currently, if you use the m//o qualifier on a regular expression within an anonymous sub, *all* closures generated from that anonymous sub will use the regular expression as it was compiled when it was used the very first time in any such closure. For instance, if you say

```
    sub build_match {
        my($left,$right) = @_;
        return sub { $_[0] =~ /$left stuff $right/o; };
    }
    $good = build_match('foo','bar');
    $bad = build_match('baz','blarch');
    print $good->('foo stuff bar') ? "ok\n" : "not ok\n";
    print $bad->('baz stuff blarch') ? "ok\n" : "not ok\n";
    print $bad->('foo stuff bar') ? "not ok\n" : "ok\n";
```

  For most builds of Perl5, this will print: ok not ok not ok

  build_match() will always return a sub which matches the contents of $left and $right as they were the *first* time that build_match() was called, not as they are in the current call.

- Regular Expression

  If no parentheses are used in a match, Perl4 sets $+ to the whole match, just like $&. Perl5 does not.

```
    "abcdef" =~ /b.*e/;
    print "\$+ = $+\n";

    # perl4 prints: bcde
    # perl5 prints:
```

- Regular Expression

  substitution now returns the null string if it fails

```
    $string = "test";
    $value = ($string =~ s/foo//);
    print $value, "\n";

    # perl4 prints: 0
    # perl5 prints:
```

Also see Numerical Traps for another example of this new feature.

- Regular Expression

  `s‘lhs‘rhs‘` (using backticks) is now a normal substitution, with no backtick expansion

```
$string = "";
$string =~ s‘^‘hostname‘;
print $string, "\n";

# perl4 prints: <the local hostname>
# perl5 prints: hostname
```

- Regular Expression

  Stricter parsing of variables used in regular expressions

```
s/^([^$grpc]*$grpc[$opt$plus$rep]?)//o;

# perl4: compiles w/o error
# perl5: with Scalar found where operator expected ..., near "$opt$plus"
```

  an added component of this example, apparently from the same script, is the actual value of the s’d string after the substitution. `[$opt]` is a character class in perl4 and an array subscript in perl5

```
$grpc = ’a’;
$opt  = ’r’;
$_ = ’bar’;
s/^([^$grpc]*$grpc[$opt]?)/foo/;
print ;

# perl4 prints: foo
# perl5 prints: foobar
```

- Regular Expression

  Under perl5, `m?x?` matches only once, like `?x?`. Under perl4, it matched repeatedly, like `/x/` or `m!x!`.

```
$test = "once";
sub match { $test =~ m?once?; }
&match();
if( &match() ) {
    # m?x? matches more then once
    print "perl4\n";
} else {
    # m?x? matches only once
    print "perl5\n";
}

# perl4 prints: perl4
# perl5 prints: perl5
```

- Regular Expression

  Unlike in Ruby, failed matches in Perl do not reset the match variables ($1, $2, ..., `$‘`, ...).

14.1.14 Subroutine, Signal, Sorting Traps

The general group of Perl4-to-Perl5 traps having to do with Signals, Sorting, and their related subroutines, as well as general subroutine traps. Includes some OS-Specific traps.

- (Signals)

  Barewords that used to look like strings to Perl will now look like subroutine calls if a subroutine by that name is defined before the compiler sees them.

```
sub SeeYa { warn"Hasta la vista, baby!" }
$SIG{'TERM'} = SeeYa;
print "SIGTERM is now $SIG{'TERM'}\n";

# perl4 prints: SIGTERM is now main'SeeYa
# perl5 prints: SIGTERM is now main::1 (and warns "Hasta la vista, baby!")
```

  Use **-w** to catch this one

- (Sort Subroutine)

  reverse is no longer allowed as the name of a sort subroutine.

```
sub reverse{ print "yup "; $a <=> $b }
print sort reverse (2,1,3);

# perl4 prints: yup yup 123
# perl5 prints: 123
# perl5 warns (if using -w): Ambiguous call resolved as CORE::reverse()
```

- warn() won't let you specify a filehandle.

  Although it _always_ printed to STDERR, warn() would let you specify a filehandle in perl4. With perl5 it does not.

```
warn STDERR "Foo!";

# perl4 prints: Foo!
# perl5 prints: String found where operator expected
```

14.1.15 OS Traps

- (SysV)

  Under HPUX, and some other SysV OSes, one had to reset any signal handler, within the signal handler function, each time a signal was handled with perl4. With perl5, the reset is now done correctly. Any code relying on the handler _not_ being reset will have to be reworked.

  Since version 5.002, Perl uses sigaction() under SysV.

```
sub gotit {
    print "Got @_... ";
}
$SIG{'INT'} = 'gotit';
```

```
$| = 1;
$pid = fork;
if ($pid) {
    kill('INT', $pid);
    sleep(1);
    kill('INT', $pid);
} else {
    while (1) {sleep(10);}
}

# perl4 (HPUX) prints: Got INT...
# perl5 (HPUX) prints: Got INT... Got INT...
```

- (SysV)

  Under SysV OSes, `seek()` on a file opened to append >> now does the right thing w.r.t. the fopen() manpage. e.g., - When a file is opened for append, it is impossible to overwrite information already in the file.

```
open(TEST,">>seek.test");
$start = tell TEST ;
foreach(1 .. 9){
    print TEST "$_ ";
}
$end = tell TEST ;
seek(TEST,$start,0);
print TEST "18 characters here";

# perl4 (solaris) seek.test has: 18 characters here
# perl5 (solaris) seek.test has: 1 2 3 4 5 6 7 8 9 18 characters here
```

14.1.16 Interpolation Traps

Perl4-to-Perl5 traps having to do with how things get interpolated within certain expressions, statements, contexts, or whatever.

- Interpolation

  @ now always interpolates an array in double-quotish strings.

```
print "To: someone@somewhere.com\n";

# perl4 prints: To:someone@somewhere.com
# perl < 5.6.1, error : In string, @somewhere now must be written as \@somewhere
# perl >= 5.6.1, warning : Possible unintended interpolation of @somewhere in string
```

- Interpolation

  Double-quoted strings may no longer end with an unescaped $.

```
$foo = "foo$";
print "foo is $foo\n";

# perl4 prints: foo is foo$
# perl5 errors: Final $ should be \$ or $name
```

Note: perl5 DOES NOT error on the terminating @ in $bar

- Interpolation

  Perl now sometimes evaluates arbitrary expressions inside braces that occur within double quotes (usually when the opening brace is preceded by `$` or `@`).

```
@www = "buz";
$foo = "foo";
$bar = "bar";
sub foo { return "bar" };
print "|@{w.w.w}|${main'foo}|";

# perl4 prints: |@{w.w.w}|foo|
# perl5 prints: |buz|bar|
```

  Note that you can `use strict;` to ward off such trappiness under perl5.

- Interpolation

  The construct "this is $$x" used to interpolate the pid at that point, but now tries to dereference $x. `$$` by itself still works fine, however.

```
$s = "a reference";
$x = *s;
print "this is $$x\n";

# perl4 prints: this is XXXx   (XXX is the current pid)
# perl5 prints: this is a reference
```

- Interpolation

  Creation of hashes on the fly with `eval "EXPR"` now requires either both `$`'s to be protected in the specification of the hash name, or both curlies to be protected. If both curlies are protected, the result will be compatible with perl4 and perl5. This is a very common practice, and should be changed to use the block form of `eval{}` if possible.

```
$hashname = "foobar";
$key = "baz";
$value = 1234;
eval "\$$hashname{'$key'} = q|$value|";
(defined($foobar{'baz'})) ?  (print "Yup") : (print "Nope");

# perl4 prints: Yup
# perl5 prints: Nope
```

  Changing

```
eval "\$$hashname{'$key'} = q|$value|";
```

  to

```
eval "\$\$hashname{'$key'} = q|$value|";
```

  causes the following result:

```
# perl4 prints: Nope
# perl5 prints: Yup
```

  or, changing to

```
    eval "\$$hashname\{'$key'\} = q|$value|";
```

causes the following result:

```
    # perl4 prints: Yup
    # perl5 prints: Yup
    # and is compatible for both versions
```

- Interpolation

  perl4 programs which unconsciously rely on the bugs in earlier perl versions.

```
    perl -e '$bar=q/not/; print "This is $foo{$bar} perl5"'

    # perl4 prints: This is not perl5
    # perl5 prints: This is perl5
```

- Interpolation

  You also have to be careful about array and hash brackets during interpolation.

```
    print "$foo["

    perl 4 prints: [
    perl 5 prints: syntax error

    print "$foo{"

    perl 4 prints: {
    perl 5 prints: syntax error
```

  Perl 5 is expecting to find an index or key name following the respective brackets, as well as an ending bracket of the appropriate type. In order to mimic the behavior of Perl 4, you must escape the bracket like so.

```
    print "$foo\[";
    print "$foo\{";
```

- Interpolation

  Similarly, watch out for:

```
    $foo = "baz";
    print "\$$foo{bar}\n";

    # perl4 prints: $baz{bar}
    # perl5 prints: $
```

  Perl 5 is looking for `$foo{bar}` which doesn't exist, but perl 4 is happy just to expand $foo to "baz" by itself. Watch out for this especially in `eval`'s.

- Interpolation

  `qq()` string passed to `eval`

```
    eval qq(
        foreach \$y (keys %\$x\) {
            \$count++;
        }
    );

    # perl4 runs this ok
    # perl5 prints: Can't find string terminator ")"
```

14.1.17 DBM Traps

General DBM traps.

- DBM

  Existing dbm databases created under perl4 (or any other dbm/ndbm tool) may cause the same script, run under perl5, to fail. The build of perl5 must have been linked with the same dbm/ndbm as the default for `dbmopen()` to function properly without `tie`'ing to an extension dbm implementation.

  ```
  dbmopen (%dbm, "file", undef);
  print "ok\n";

  # perl4 prints: ok
  # perl5 prints: ok (IFF linked with -ldbm or -lndbm)
  ```

- DBM

  Existing dbm databases created under perl4 (or any other dbm/ndbm tool) may cause the same script, run under perl5, to fail. The error generated when exceeding the limit on the key/value size will cause perl5 to exit immediately.

  ```
  dbmopen(DB, "testdb",0600) || die "couldn't open db! $!";
  $DB{'trap'} = "x" x 1024;  # value too large for most dbm/ndbm
  print "YUP\n";

  # perl4 prints:
  dbm store returned -1, errno 28, key "trap" at - line 3.
  YUP

  # perl5 prints:
  dbm store returned -1, errno 28, key "trap" at - line 3.
  ```

14.1.18 Unclassified Traps

Everything else.

- `require`/`do` trap using returned value

  If the file doit.pl has:

  ```
  sub foo {
      $rc = do "./do.pl";
      return 8;
  }
  print &foo, "\n";
  ```

  And the do.pl file has the following single line:

  ```
  return 3;
  ```

  Running doit.pl gives the following:

  ```
  # perl 4 prints: 3 (aborts the subroutine early)
  # perl 5 prints: 8
  ```

  Same behavior if you replace `do` with `require`.

- `split` on empty string with LIMIT specified

  ```
  $string = '';
  @list = split(/foo/, $string, 2)
  ```

  Perl4 returns a one element list containing the empty string but Perl5 returns an empty list.

As always, if any of these are ever officially declared as bugs, they'll be fixed and removed.

Chapter 15

perldebtut

Perl debugging tutorial

15.1 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.

15.2 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.

15.3 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.

15.4 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-commital, 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 *perlref* and *perlreftut*

15.5 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 ($arg =~ /^\-(c|f)((\-|\+)*\d+(\.\d+)*)$/) {
        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);
        $out =~ s/^((\-|\+)*\d+)\.0+$/$1/;
        print "$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 'd' or 'D'.

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.

15.6 Placeholder for a, w, t, T

Actions, watch variables, stack traces etc.: on the TODO list.

```
a

w

t

T
```

15.7 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 *perlre*, *perlretut*, and to decode the mysterious labels (BOL and CURLYN, etc. above), see *perldebguts*.

15.8 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, *perlvar* 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 *perlopentut* and *perlfaq8*.

15.9 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.

15.10 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).

15.11 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 (*perldebug* is probably the next place to go), and of course, experiment.

15.12 SEE ALSO

perldebug, *perldebguts*, *perldiag*, *dprofpp*, *perlrun*

15.13 AUTHOR

Richard Foley <richard@rfi.net> Copyright (c) 2000

15.14 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>

Chapter 16

perlfaq

Frequently asked questions about Perl ($Date: 2003/01/31 17:37:17 $)

16.1 DESCRIPTION

The perlfaq is divided into several documents based on topics. A table of contents is at the end of this document.

16.1.1 Where to get the perlfaq

Extracts of the perlfaq are posted regularly to comp.lang.perl.misc. It is available on many web sites: http://www.perldoc.com/ and http://faq.perl.org/

16.1.2 How to contribute to the perlfaq

You may mail corrections, additions, and suggestions to perlfaq-workers@perl.org . This alias should not be used to *ask* FAQs. It's for fixing the current FAQ. Send questions to the comp.lang.perl.misc newsgroup. You can view the source tree at http://cvs.perl.org/cvsweb/perlfaq/ (which is outside of the main Perl source tree). The CVS repository notes all changes to the FAQ.

16.1.3 What will happen if you mail your Perl programming problems to the authors

Your questions will probably go unread, unless they're suggestions of new questions to add to the FAQ, in which case they should have gone to the perlfaq-workers@perl.org instead.

You should have read section 2 of this faq. There you would have learned that comp.lang.perl.misc is the appropriate place to go for free advice. If your question is really important and you require a prompt and correct answer, you should hire a consultant.

16.2 Credits

The original perlfaq was written by Tom Christiansen, then expanded by collaboration between Tom and Nathan Torkington. The current document is maintained by the perlfaq-workers (perlfaq-workers@perl.org). Several people have contributed answers, corrections, and comments.

16.3 Author and Copyright Information

Copyright (c) 1997-2003 Tom Christiansen, Nathan Torkington, and other contributors noted in the answers.

All rights reserved.

16.3.1 Bundled Distributions

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in these files are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

16.3.2 Disclaimer

This information is offered in good faith and in the hope that it may be of use, but is not guaranteed to be correct, up to date, or suitable for any particular purpose whatsoever. The authors accept no liability in respect of this information or its use.

16.4 Table of Contents

16.5 The Questions

16.5.1 *perlfaq1*: General Questions About Perl

Very general, high-level questions about Perl.

- What is Perl?
- Who supports Perl? Who develops it? Why is it free?
- Which version of Perl should I use?
- What are perl4 and perl5?
- What is perl6?
- How stable is Perl?
- Is Perl difficult to learn?
- How does Perl compare with other languages like Java, Python, REXX, Scheme, or Tcl?
- Can I do [task] in Perl?

- When shouldn't I program in Perl?
- What's the difference between "perl" and "Perl"?
- Is it a Perl program or a Perl script?
- What is a JAPH?
- Where can I get a list of Larry Wall witticisms?
- How can I convince my sysadmin/supervisor/employees to use version 5/5.6.1/Perl instead of some other language?

16.5.2 *perlfaq2*: Obtaining and Learning about Perl

Where to find source and documentation for Perl, support, and related matters.

- What machines support Perl? Where do I get it?
- How can I get a binary version of Perl?
- I don't have a C compiler on my system. How can I compile perl?
- I copied the Perl binary from one machine to another, but scripts don't work.
- I grabbed the sources and tried to compile but gdbm/dynamic loading/malloc/linking/... failed. How do I make it work?
- What modules and extensions are available for Perl? What is CPAN? What does CPAN/src/... mean?
- Is there an ISO or ANSI certified version of Perl?
- Where can I get information on Perl?
- What are the Perl newsgroups on Usenet? Where do I post questions?
- Where should I post source code?
- Perl Books
- Perl in Magazines
- Perl on the Net: FTP and WWW Access
- What mailing lists are there for Perl?
- Archives of comp.lang.perl.misc
- Where can I buy a commercial version of Perl?
- Where do I send bug reports?
- What is perl.com? Perl Mongers? pm.org? perl.org? cpan.org?

16.5.3 *perlfaq3*: Programming Tools

Programmer tools and programming support.

- How do I do (anything)?
- How can I use Perl interactively?
- Is there a Perl shell?
- How do I find which modules are installed on my system?
- How do I debug my Perl programs?
- How do I profile my Perl programs?
- How do I cross-reference my Perl programs?
- Is there a pretty-printer (formatter) for Perl?
- Is there a ctags for Perl?
- Is there an IDE or Windows Perl Editor?
- Where can I get Perl macros for vi?
- Where can I get perl-mode for emacs?
- How can I use curses with Perl?
- How can I use X or Tk with Perl?
- How can I generate simple menus without using CGI or Tk?
- How can I make my Perl program run faster?
- How can I make my Perl program take less memory?
- Is it safe to return a reference to local or lexical data?
- How can I free an array or hash so my program shrinks?
- How can I make my CGI script more efficient?
- How can I hide the source for my Perl program?
- How can I compile my Perl program into byte code or C?
- How can I compile Perl into Java?
- How can I get `#!perl` to work on [MS-DOS,NT,...]?
- Can I write useful Perl programs on the command line?
- Why don’t Perl one-liners work on my DOS/Mac/VMS system?
- Where can I learn about CGI or Web programming in Perl?
- Where can I learn about object-oriented Perl programming?
- Where can I learn about linking C with Perl? [h2xs, xsubpp]
- I’ve read perlembed, perlguts, etc., but I can’t embed perl in my C program; what am I doing wrong?
- When I tried to run my script, I got this message. What does it mean?
- What’s MakeMaker?

16.5.4 *perlfaq4*: Data Manipulation

Manipulating numbers, dates, strings, arrays, hashes, and miscellaneous data issues.

- Why am I getting long decimals (eg, 19.9499999999999) instead of the numbers I should be getting (eg, 19.95)?
- Why isn't my octal data interpreted correctly?
- Does Perl have a round() function? What about ceil() and floor()? Trig functions?
- How do I convert between numeric representations?
- Why doesn't & work the way I want it to?
- How do I multiply matrices?
- How do I perform an operation on a series of integers?
- How can I output Roman numerals?
- Why aren't my random numbers random?
- How do I get a random number between X and Y?
- How do I find the day or week of the year?
- How do I find the current century or millennium?
- How can I compare two dates and find the difference?
- How can I take a string and turn it into epoch seconds?
- How can I find the Julian Day?
- How do I find yesterday's date?
- Does Perl have a Year 2000 problem? Is Perl Y2K compliant?
- How do I validate input?
- How do I unescape a string?
- How do I remove consecutive pairs of characters?
- How do I expand function calls in a string?
- How do I find matching/nesting anything?
- How do I reverse a string?
- How do I expand tabs in a string?
- How do I reformat a paragraph?
- How can I access or change N characters of a string?
- How do I change the Nth occurrence of something?
- How can I count the number of occurrences of a substring within a string?
- How do I capitalize all the words on one line?
- How can I split a [character] delimited string except when inside [character]?
- How do I strip blank space from the beginning/end of a string?
- How do I pad a string with blanks or pad a number with zeroes?

- How do I extract selected columns from a string?
- How do I find the soundex value of a string?
- How can I expand variables in text strings?
- What's wrong with always quoting "$vars"?
- Why don't my <<HERE documents work?
- What is the difference between a list and an array?
- What is the difference between $array[1] and @array[1]?
- How can I remove duplicate elements from a list or array?
- How can I tell whether a certain element is contained in a list or array?
- How do I compute the difference of two arrays? How do I compute the intersection of two arrays?
- How do I test whether two arrays or hashes are equal?
- How do I find the first array element for which a condition is true?
- How do I handle linked lists?
- How do I handle circular lists?
- How do I shuffle an array randomly?
- How do I process/modify each element of an array?
- How do I select a random element from an array?
- How do I permute N elements of a list?
- How do I sort an array by (anything)?
- How do I manipulate arrays of bits?
- Why does defined() return true on empty arrays and hashes?
- How do I process an entire hash?
- What happens if I add or remove keys from a hash while iterating over it?
- How do I look up a hash element by value?
- How can I know how many entries are in a hash?
- How do I sort a hash (optionally by value instead of key)?
- How can I always keep my hash sorted?
- What's the difference between "delete" and "undef" with hashes?
- Why don't my tied hashes make the defined/exists distinction?
- How do I reset an each() operation part-way through?
- How can I get the unique keys from two hashes?
- How can I store a multidimensional array in a DBM file?
- How can I make my hash remember the order I put elements into it?
- Why does passing a subroutine an undefined element in a hash create it?

- How can I make the Perl equivalent of a C structure/C++ class/hash or array of hashes or arrays?
- How can I use a reference as a hash key?
- How do I handle binary data correctly?
- How do I determine whether a scalar is a number/whole/integer/float?
- How do I keep persistent data across program calls?
- How do I print out or copy a recursive data structure?
- How do I define methods for every class/object?
- How do I verify a credit card checksum?
- How do I pack arrays of doubles or floats for XS code?

16.5.5 *perlfaq5*: Files and Formats

I/O and the "f" issues: filehandles, flushing, formats, and footers.

- How do I flush/unbuffer an output filehandle? Why must I do this?
- How do I change one line in a file/delete a line in a file/insert a line in the middle of a file/append to the beginning of a file?
- How do I count the number of lines in a file?
- How can I use Perl's `-i` option from within a program?
- How do I make a temporary file name?
- How can I manipulate fixed-record-length files?
- How can I make a filehandle local to a subroutine? How do I pass filehandles between subroutines? How do I make an array of filehandles?
- How can I use a filehandle indirectly?
- How can I set up a footer format to be used with write()?
- How can I write() into a string?
- How can I output my numbers with commas added?
- How can I translate tildes (˜) in a filename?
- How come when I open a file read-write it wipes it out?
- Why do I sometimes get an "Argument list too long" when I use <*>?
- Is there a leak/bug in glob()?
- How can I open a file with a leading ">" or trailing blanks?
- How can I reliably rename a file?
- How can I lock a file?
- Why can't I just open(FH, ">file.lock")?
- I still don't get locking. I just want to increment the number in the file. How can I do this?
- All I want to do is append a small amount of text to the end of a file. Do I still have to use locking?

- How do I randomly update a binary file?
- How do I get a file's timestamp in perl?
- How do I set a file's timestamp in perl?
- How do I print to more than one file at once?
- How can I read in an entire file all at once?
- How can I read in a file by paragraphs?
- How can I read a single character from a file? From the keyboard?
- How can I tell whether there's a character waiting on a filehandle?
- How do I do a `tail -f` in perl?
- How do I dup() a filehandle in Perl?
- How do I close a file descriptor by number?
- Why can't I use "C:\temp\foo" in DOS paths? What doesn't ‘C:\temp\foo.exe‘ work?
- Why doesn't glob("*.*") get all the files?
- Why does Perl let me delete read-only files? Why does `-i` clobber protected files? Isn't this a bug in Perl?
- How do I select a random line from a file?
- Why do I get weird spaces when I print an array of lines?

16.5.6 *perlfaq6*: Regular Expressions

Pattern matching and regular expressions.

- How can I hope to use regular expressions without creating illegible and unmaintainable code?
- I'm having trouble matching over more than one line. What's wrong?
- How can I pull out lines between two patterns that are themselves on different lines?
- I put a regular expression into $/ but it didn't work. What's wrong?
- How do I substitute case insensitively on the LHS while preserving case on the RHS?
- How can I make `\w` match national character sets?
- How can I match a locale-smart version of `/[a-zA-Z]/`?
- How can I quote a variable to use in a regex?
- What is `/o` really for?
- How do I use a regular expression to strip C style comments from a file?
- Can I use Perl regular expressions to match balanced text?
- What does it mean that regexes are greedy? How can I get around it?
- How do I process each word on each line?
- How can I print out a word-frequency or line-frequency summary?
- How can I do approximate matching?

- How do I efficiently match many regular expressions at once?
- Why don't word-boundary searches with `\b` work for me?
- Why does using $&, $‘, or $’ slow my program down?
- What good is `\G` in a regular expression?
- Are Perl regexes DFAs or NFAs? Are they POSIX compliant?
- What's wrong with using grep or map in a void context?
- How can I match strings with multibyte characters?
- How do I match a pattern that is supplied by the user?

16.5.7 *perlfaq7*: General Perl Language Issues

General Perl language issues that don't clearly fit into any of the other sections.

- Can I get a BNF/yacc/RE for the Perl language?
- What are all these $@%&* punctuation signs, and how do I know when to use them?
- Do I always/never have to quote my strings or use semicolons and commas?
- How do I skip some return values?
- How do I temporarily block warnings?
- What's an extension?
- Why do Perl operators have different precedence than C operators?
- How do I declare/create a structure?
- How do I create a module?
- How do I create a class?
- How can I tell if a variable is tainted?
- What's a closure?
- What is variable suicide and how can I prevent it?
- How can I pass/return a {Function, FileHandle, Array, Hash, Method, Regex}?
- How do I create a static variable?
- What's the difference between dynamic and lexical (static) scoping? Between local() and my()?
- How can I access a dynamic variable while a similarly named lexical is in scope?
- What's the difference between deep and shallow binding?
- Why doesn't "my($foo) = <FILE>;" work right?
- How do I redefine a builtin function, operator, or method?
- What's the difference between calling a function as &foo and foo()?
- How do I create a switch or case statement?
- How can I catch accesses to undefined variables, functions, or methods?

- Why can't a method included in this same file be found?
- How can I find out my current package?
- How can I comment out a large block of perl code?
- How do I clear a package?
- How can I use a variable as a variable name?
- What does "bad interpreter" mean?

16.5.8 *perlfaq8*: System Interaction

Interprocess communication (IPC), control over the user-interface (keyboard, screen and pointing devices).

- How do I find out which operating system I'm running under?
- How come exec() doesn't return?
- How do I do fancy stuff with the keyboard/screen/mouse?
- How do I print something out in color?
- How do I read just one key without waiting for a return key?
- How do I check whether input is ready on the keyboard?
- How do I clear the screen?
- How do I get the screen size?
- How do I ask the user for a password?
- How do I read and write the serial port?
- How do I decode encrypted password files?
- How do I start a process in the background?
- How do I trap control characters/signals?
- How do I modify the shadow password file on a Unix system?
- How do I set the time and date?
- How can I sleep() or alarm() for under a second?
- How can I measure time under a second?
- How can I do an atexit() or setjmp()/longjmp()? (Exception handling)
- Why doesn't my sockets program work under System V (Solaris)? What does the error message "Protocol not supported" mean?
- How can I call my system's unique C functions from Perl?
- Where do I get the include files to do ioctl() or syscall()?
- Why do setuid perl scripts complain about kernel problems?
- How can I open a pipe both to and from a command?
- Why can't I get the output of a command with system()?
- How can I capture STDERR from an external command?

- Why doesn't open() return an error when a pipe open fails?
- What's wrong with using backticks in a void context?
- How can I call backticks without shell processing?
- Why can't my script read from STDIN after I gave it EOF (ˆD on Unix, ˆZ on MS-DOS)?
- How can I convert my shell script to perl?
- Can I use perl to run a telnet or ftp session?
- How can I write expect in Perl?
- Is there a way to hide perl's command line from programs such as "ps"?
- I {changed directory, modified my environment} in a perl script. How come the change disappeared when I exited the script? How do I get my changes to be visible?
- How do I close a process's filehandle without waiting for it to complete?
- How do I fork a daemon process?
- How do I find out if I'm running interactively or not?
- How do I timeout a slow event?
- How do I set CPU limits?
- How do I avoid zombies on a Unix system?
- How do I use an SQL database?
- How do I make a system() exit on control-C?
- How do I open a file without blocking?
- How do I install a module from CPAN?
- What's the difference between require and use?
- How do I keep my own module/library directory?
- How do I add the directory my program lives in to the module/library search path?
- How do I add a directory to my include path at runtime?
- What is socket.ph and where do I get it?

16.5.9 *perlfaq9*: Networking

Networking, the internet, and a few on the web.

- What is the correct form of response from a CGI script?
- My CGI script runs from the command line but not the browser. (500 Server Error)
- How can I get better error messages from a CGI program?
- How do I remove HTML from a string?
- How do I extract URLs?
- How do I download a file from the user's machine? How do I open a file on another machine?
- How do I make a pop-up menu in HTML?

- How do I fetch an HTML file?
- How do I automate an HTML form submission?
- How do I decode or create those %-encodings on the web?
- How do I redirect to another page?
- How do I put a password on my web pages?
- How do I edit my .htpasswd and .htgroup files with Perl?
- How do I make sure users can’t enter values into a form that cause my CGI script to do bad things?
- How do I parse a mail header?
- How do I decode a CGI form?
- How do I check a valid mail address?
- How do I decode a MIME/BASE64 string?
- How do I return the user’s mail address?
- How do I send mail?
- How do I use MIME to make an attachment to a mail message?
- How do I read mail?
- How do I find out my hostname/domainname/IP address?
- How do I fetch a news article or the active newsgroups?
- How do I fetch/put an FTP file?
- How can I do RPC in Perl?

Chapter 17

perlfaq1

General Questions About Perl ($Revision: 1.14 $, $Date: 2003/11/23 08:02:29 $)

17.1 DESCRIPTION

This section of the FAQ answers very general, high-level questions about Perl.

17.1.1 What is Perl?

Perl is a high-level programming language with an eclectic heritage written by Larry Wall and a cast of thousands. It derives from the ubiquitous C programming language and to a lesser extent from sed, awk, the Unix shell, and at least a dozen other tools and languages. Perl's process, file, and text manipulation facilities make it particularly well-suited for tasks involving quick prototyping, system utilities, software tools, system management tasks, database access, graphical programming, networking, and world wide web programming. These strengths make it especially popular with system administrators and CGI script authors, but mathematicians, geneticists, journalists, and even managers also use Perl. Maybe you should, too.

17.1.2 Who supports Perl? Who develops it? Why is it free?

The original culture of the pre-populist Internet and the deeply-held beliefs of Perl's author, Larry Wall, gave rise to the free and open distribution policy of perl. Perl is supported by its users. The core, the standard Perl library, the optional modules, and the documentation you're reading now were all written by volunteers. See the personal note at the end of the README file in the perl source distribution for more details. See *perlhist* (new as of 5.005) for Perl's milestone releases.

In particular, the core development team (known as the Perl Porters) are a rag-tag band of highly altruistic individuals committed to producing better software for free than you could hope to purchase for money. You may snoop on pending developments via the archives at http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/ and http://archive.develooper.com/perl5-porters@perl.org/ or the news gateway nntp://nntp.perl.org/perl.perl5.porters or its web interface at http://nntp.perl.org/group/perl.perl5.porters , or read the faq at http://simon-cozens.org/writings/p5p-faq , or you can subscribe to the mailing list by sending perl5-porters-request@perl.org a subscription request (an empty message with no subject is fine).

While the GNU project includes Perl in its distributions, there's no such thing as "GNU Perl". Perl is not produced nor maintained by the Free Software Foundation. Perl's licensing terms are also more open than GNU software's tend to be.

You can get commercial support of Perl if you wish, although for most users the informal support will more than suffice. See the answer to "Where can I buy a commercial version of perl?" for more information.

17.1.3 Which version of Perl should I use?

You should definitely use version 5. Version 4 is old, limited, and no longer maintained; its last patch (4.036) was in 1992, long ago and far away. Sure, it's stable, but so is anything that's dead; in fact, perl4 had been called a dead, flea-bitten camel carcass. The most recent production release is 5.8.2 (although 5.005_03 and 5.6.2 are still supported). The most cutting-edge development release is 5.9. Further references to the Perl language in this document refer to the production release unless otherwise specified. There may be one or more official bug fixes by the time you read this, and also perhaps some experimental versions on the way to the next release. All releases prior to 5.004 were subject to buffer overruns, a grave security issue.

17.1.4 What are perl4 and perl5?

Perl4 and perl5 are informal names for different versions of the Perl programming language. It's easier to say "perl5" than it is to say "the 5(.004) release of Perl", but some people have interpreted this to mean there's a language called "perl5", which isn't the case. Perl5 is merely the popular name for the fifth major release (October 1994), while perl4 was the fourth major release (March 1991). There was also a perl1 (in January 1988), a perl2 (June 1988), and a perl3 (October 1989).

The 5.0 release is, essentially, a ground-up rewrite of the original perl source code from releases 1 through 4. It has been modularized, object-oriented, tweaked, trimmed, and optimized until it almost doesn't look like the old code. However, the interface is mostly the same, and compatibility with previous releases is very high. See Perl4 to Perl5 Traps in *perltrap*.

To avoid the "what language is perl5?" confusion, some people prefer to simply use "perl" to refer to the latest version of perl and avoid using "perl5" altogether. It's not really that big a deal, though.

See *perlhist* for a history of Perl revisions.

17.1.5 What is Ponie?

At The O'Reilly Open Source Software Convention in 2003, Artur Bergman, Fotango, and The Perl Foundation announced a project to run perl5 on the Parrot virtual machine named Ponie. Ponie stands for Perl On New Internal Engine. The Perl 5.10 language implementation will be used for Ponie, and there will be no language level differences between perl5 and ponie. Ponie is not a complete rewrite of perl5.

For more details, see http://www.poniecode.org/

17.1.6 What is perl6?

At The Second O'Reilly Open Source Software Convention, Larry Wall announced Perl6 development would begin in earnest. Perl6 was an oft used term for Chip Salzenberg's project to rewrite Perl in C++ named Topaz. However, Topaz provided valuable insights to the next version of Perl and its implementation, but was ultimately abandoned.

If you want to learn more about Perl6, or have a desire to help in the crusade to make Perl a better place then peruse the Perl6 developers page at http://dev.perl.org/perl6/ and get involved.

Perl6 is not scheduled for release yet, and Perl5 will still be supported for quite awhile after its release. Do not wait for Perl6 to do whatever you need to do.

"We're really serious about reinventing everything that needs reinventing." –Larry Wall

17.1.7 How stable is Perl?

Production releases, which incorporate bug fixes and new functionality, are widely tested before release. Since the 5.000 release, we have averaged only about one production release per year.

Larry and the Perl development team occasionally make changes to the internal core of the language, but all possible efforts are made toward backward compatibility. While not quite all perl4 scripts run flawlessly under perl5, an update to perl should nearly never invalidate a program written for an earlier version of perl (barring accidental bug fixes and the rare new keyword).

17.1.8 Is Perl difficult to learn?

No, Perl is easy to start learning–and easy to keep learning. It looks like most programming languages you're likely to have experience with, so if you've ever written a C program, an awk script, a shell script, or even a BASIC program, you're already partway there.

Most tasks only require a small subset of the Perl language. One of the guiding mottos for Perl development is "there's more than one way to do it" (TMTOWTDI, sometimes pronounced "tim toady"). Perl's learning curve is therefore shallow (easy to learn) and long (there's a whole lot you can do if you really want).

Finally, because Perl is frequently (but not always, and certainly not by definition) an interpreted language, you can write your programs and test them without an intermediate compilation step, allowing you to experiment and test/debug quickly and easily. This ease of experimentation flattens the learning curve even more.

Things that make Perl easier to learn: Unix experience, almost any kind of programming experience, an understanding of regular expressions, and the ability to understand other people's code. If there's something you need to do, then it's probably already been done, and a working example is usually available for free. Don't forget the new perl modules, either. They're discussed in Part 3 of this FAQ, along with CPAN, which is discussed in Part 2.

17.1.9 How does Perl compare with other languages like Java, Python, REXX, Scheme, or Tcl?

Favorably in some areas, unfavorably in others. Precisely which areas are good and bad is often a personal choice, so asking this question on Usenet runs a strong risk of starting an unproductive Holy War.

Probably the best thing to do is try to write equivalent code to do a set of tasks. These languages have their own newsgroups in which you can learn about (but hopefully not argue about) them.

Some comparison documents can be found at http://language.perl.com/versus/ if you really can't stop yourself.

17.1.10 Can I do [task] in Perl?

Perl is flexible and extensible enough for you to use on virtually any task, from one-line file-processing tasks to large, elaborate systems. For many people, Perl serves as a great replacement for shell scripting. For others, it serves as a convenient, high-level replacement for most of what they'd program in low-level languages like C or C++. It's ultimately up to you (and possibly your management) which tasks you'll use Perl for and which you won't.

If you have a library that provides an API, you can make any component of it available as just another Perl function or variable using a Perl extension written in C or C++ and dynamically linked into your main perl interpreter. You can also go the other direction, and write your main program in C or C++, and then link in some Perl code on the fly, to create a powerful application. See *perlembed*.

That said, there will always be small, focused, special-purpose languages dedicated to a specific problem domain that are simply more convenient for certain kinds of problems. Perl tries to be all things to all people, but nothing special to anyone. Examples of specialized languages that come to mind include prolog and matlab.

17.1.11 When shouldn't I program in Perl?

When your manager forbids it–but do consider replacing them :-).

Actually, one good reason is when you already have an existing application written in another language that's all done (and done well), or you have an application language specifically designed for a certain task (e.g. prolog, make).

For various reasons, Perl is probably not well-suited for real-time embedded systems, low-level operating systems development work like device drivers or context-switching code, complex multi-threaded shared-memory applications, or extremely large applications. You'll notice that perl is not itself written in Perl.

The new, native-code compiler for Perl may eventually reduce the limitations given in the previous statement to some degree, but understand that Perl remains fundamentally a dynamically typed language, not a statically typed one. You certainly won't be chastised if you don't trust nuclear-plant or brain-surgery monitoring code to it. And Larry will sleep easier, too–Wall Street programs not withstanding. :-)

17.1.12 What's the difference between "perl" and "Perl"?

One bit. Oh, you weren't talking ASCII? :-) Larry now uses "Perl" to signify the language proper and "perl" the implementation of it, i.e. the current interpreter. Hence Tom's quip that "Nothing but perl can parse Perl." You may or may not choose to follow this usage. For example, parallelism means "awk and perl" and "Python and Perl" look OK, while "awk and Perl" and "Python and perl" do not. But never write "PERL", because perl is not an acronym, apocryphal folklore and post-facto expansions notwithstanding.

17.1.13 Is it a Perl program or a Perl script?

Larry doesn't really care. He says (half in jest) that "a script is what you give the actors. A program is what you give the audience."

Originally, a script was a canned sequence of normally interactive commands–that is, a chat script. Something like a UUCP or PPP chat script or an expect script fits the bill nicely, as do configuration scripts run by a program at its start up, such *.cshrc* or *.ircrc*, for example. Chat scripts were just drivers for existing programs, not stand-alone programs in their own right.

A computer scientist will correctly explain that all programs are interpreted and that the only question is at what level. But if you ask this question of someone who isn't a computer scientist, they might tell you that a *program* has been compiled to physical machine code once and can then be run multiple times, whereas a *script* must be translated by a program each time it's used.

Perl programs are (usually) neither strictly compiled nor strictly interpreted. They can be compiled to a byte-code form (something of a Perl virtual machine) or to completely different languages, like C or assembly language. You can't tell just by looking at it whether the source is destined for a pure interpreter, a parse-tree interpreter, a byte-code interpreter, or a native-code compiler, so it's hard to give a definitive answer here.

Now that "script" and "scripting" are terms that have been seized by unscrupulous or unknowing marketeers for their own nefarious purposes, they have begun to take on strange and often pejorative meanings, like "non serious" or "not real programming". Consequently, some Perl programmers prefer to avoid them altogether.

17.1.14 What is a JAPH?

These are the "just another perl hacker" signatures that some people sign their postings with. Randal Schwartz made these famous. About 100 of the earlier ones are available from http://www.cpan.org/misc/japh .

17.1.15 Where can I get a list of Larry Wall witticisms?

Over a hundred quips by Larry, from postings of his or source code, can be found at http://www.cpan.org/misc/lwall-quotes.txt.gz .

17.1.16 How can I convince my sysadmin/supervisor/employees to use version 5/5.6.1/Perl instead of some other language?

If your manager or employees are wary of unsupported software, or software which doesn't officially ship with your operating system, you might try to appeal to their self-interest. If programmers can be more productive using and utilizing Perl constructs, functionality, simplicity, and power, then the typical manager/supervisor/employee may be persuaded. Regarding using Perl in general, it's also sometimes helpful to point out that delivery times may be reduced using Perl compared to other languages.

If you have a project which has a bottleneck, especially in terms of translation or testing, Perl almost certainly will provide a viable, quick solution. In conjunction with any persuasion effort, you should not fail to point out that Perl is used, quite extensively, and with extremely reliable and valuable results, at many large computer software and hardware companies throughout the world. In fact, many Unix vendors now ship Perl by default. Support is usually just a news-posting away, if you can't find the answer in the *comprehensive* documentation, including this FAQ.

See http://www.perl.org/advocacy/ for more information.

If you face reluctance to upgrading from an older version of perl, then point out that version 4 is utterly unmaintained and unsupported by the Perl Development Team. Another big sell for Perl5 is the large number of modules and extensions which greatly reduce development time for any given task. Also mention that the difference between version 4 and version 5 of Perl is like the difference between awk and C++. (Well, OK, maybe it's not quite that distinct, but you get the idea.) If you want support and a reasonable guarantee that what you're developing will continue to work in the future, then you have to run the supported version. As of December 2003 that means running either 5.8.2 (released in November 2003), or one of the older releases like 5.6.2 (also released in November 2003; a maintenance release to let perl 5.6 compile on newer systems as 5.6.1 was released in April 2001) or 5.005_03 (released in March 1999), although 5.004_05 isn't that bad if you **absolutely** need such an old version (released in April 1999) for stability reasons. Anything older than 5.004_05 shouldn't be used.

Of particular note is the massive bug hunt for buffer overflow problems that went into the 5.004 release. All releases prior to that, including perl4, are considered insecure and should be upgraded as soon as possible.

In August 2000 in all Linux distributions a new security problem was found in the optional 'suidperl' (not built or installed by default) in all the Perl branches 5.6, 5.005, and 5.004, see http://www.cpan.org/src/5.0/sperl-2000-08-05/ Perl maintenance releases 5.6.1 and 5.8.0 have this security hole closed. Most, if not all, Linux distribution have patches for this vulnerability available, see http://www.linuxsecurity.com/advisories/ , but the most recommendable way is to upgrade to at least Perl 5.6.1.

17.2 AUTHOR AND COPYRIGHT

Chapter 18

perlfaq2

Obtaining and Learning about Perl ($Revision: 1.25 $, $Date: 2003/10/16 04:57:38 $)

18.1 DESCRIPTION

This section of the FAQ answers questions about where to find source and documentation for Perl, support, and related matters.

18.1.1 What machines support Perl? Where do I get it?

The standard release of Perl (the one maintained by the perl development team) is distributed only in source code form. You can find this at http://www.cpan.org/src/latest.tar.gz , which is in a standard Internet format (a gzipped archive in POSIX tar format).

Perl builds and runs on a bewildering number of platforms. Virtually all known and current Unix derivatives are supported (Perl's native platform), as are other systems like VMS, DOS, OS/2, Windows, QNX, BeOS, OS X, MPE/iX and the Amiga.

Binary distributions for some proprietary platforms, including Apple systems, can be found http://www.cpan.org/ports/ directory. Because these are not part of the standard distribution, they may and in fact do differ from the base Perl port in a variety of ways. You'll have to check their respective release notes to see just what the differences are. These differences can be either positive (e.g. extensions for the features of the particular platform that are not supported in the source release of perl) or negative (e.g. might be based upon a less current source release of perl).

18.1.2 How can I get a binary version of Perl?

If you don't have a C compiler because your vendor for whatever reasons did not include one with your system, the best thing to do is grab a binary version of gcc from the net and use that to compile perl with. CPAN only has binaries for systems that are terribly hard to get free compilers for, not for Unix systems.

Some URLs that might help you are:

```
http://www.cpan.org/ports/
http://www.perl.com/pub/language/info/software.html
```

Someone looking for a Perl for Win16 might look to Laszlo Molnar's djgpp port in http://www.cpan.org/ports/#msdos , which comes with clear installation instructions. A simple installation guide for MS-DOS using Ilya Zakharevich's OS/2 port is available at http://www.cs.ruu.nl/%7Epiet/perl5dos.html and similarly for Windows 3.1 at http://www.cs.ruu.nl/%7Epiet/perlwin3.html .

18.1.3 I don't have a C compiler on my system. How can I compile perl?

Since you don't have a C compiler, you're doomed and your vendor should be sacrificed to the Sun gods. But that doesn't help you.

What you need to do is get a binary version of gcc for your system first. Consult the Usenet FAQs for your operating system for information on where to get such a binary version.

18.1.4 I copied the Perl binary from one machine to another, but scripts don't work.

That's probably because you forgot libraries, or library paths differ. You really should build the whole distribution on the machine it will eventually live on, and then type `make install`. Most other approaches are doomed to failure.

One simple way to check that things are in the right place is to print out the hard-coded @INC that perl looks through for libraries:

```
    % perl -le 'print for @INC'
```

If this command lists any paths that don't exist on your system, then you may need to move the appropriate libraries to these locations, or create symbolic links, aliases, or shortcuts appropriately. @INC is also printed as part of the output of

```
    % perl -V
```

You might also want to check out How do I keep my own module/library directory? in *perlfaq8*.

18.1.5 I grabbed the sources and tried to compile but gdbm/dynamic loading/malloc/linking/... failed. How do I make it work?

Read the *INSTALL* file, which is part of the source distribution. It describes in detail how to cope with most idiosyncrasies that the Configure script can't work around for any given system or architecture.

18.1.6 What modules and extensions are available for Perl? What is CPAN? What does CPAN/src/... mean?

CPAN stands for Comprehensive Perl Archive Network, a ˜1.2Gb archive replicated on nearly 200 machines all over the world. CPAN contains source code, non-native ports, documentation, scripts, and many third-party modules and extensions, designed for everything from commercial database interfaces to keyboard/screen control to web walking and CGI scripts. The master web site for CPAN is http://www.cpan.org/ and there is the CPAN Multiplexer at http://www.cpan.org/CPAN.html which will choose a mirror near you via DNS. See http://www.perl.com/CPAN (without a slash at the end) for how this process works. Also, http://mirror.cpan.org/ has a nice interface to the http://www.cpan.org/MIRRORED.BY mirror directory.

See the CPAN FAQ at http://www.cpan.org/misc/cpan-faq.html for answers to the most frequently asked questions about CPAN including how to become a mirror.

CPAN/path/... is a naming convention for files available on CPAN sites. CPAN indicates the base directory of a CPAN mirror, and the rest of the path is the path from that directory to the file. For instance, if you're using ftp://ftp.funet.fi/pub/languages/perl/CPAN as your CPAN site, the file CPAN/misc/japh is downloadable as ftp://ftp.funet.fi/pub/languages/perl/CPAN/misc/japh .

Considering that there are close to two thousand existing modules in the archive, one probably exists to do nearly anything you can think of. Current categories under CPAN/modules/by-category/ include Perl core modules; development support; operating system interfaces; networking, devices, and interprocess communication; data type utilities; database interfaces; user interfaces; interfaces to other languages; filenames, file systems, and file locking; internationalization and locale; world wide web support; server and daemon utilities; archiving and compression; image manipulation; mail and news; control flow utilities; filehandle and I/O; Microsoft Windows modules; and miscellaneous modules.

See http://www.cpan.org/modules/00modlist.long.html or http://search.cpan.org/ for a more complete list of modules by category.

CPAN is not affiliated with O'Reilly and Associates.

18.1.7 Is there an ISO or ANSI certified version of Perl?

Certainly not. Larry expects that he'll be certified before Perl is.

18.1.8 Where can I get information on Perl?

The complete Perl documentation is available with the Perl distribution. If you have Perl installed locally, you probably have the documentation installed as well: type `man perl` if you're on a system resembling Unix. This will lead you to other important man pages, including how to set your $MANPATH. If you're not on a Unix system, access to the documentation will be different; for example, documentation might only be in HTML format. All proper Perl installations have fully-accessible documentation.

You might also try `perldoc perl` in case your system doesn't have a proper man command, or it's been misinstalled. If that doesn't work, try looking in /usr/local/lib/perl5/pod for documentation.

If all else fails, consult http://perldoc.cpan.org/ or http://www.perldoc.com/ both offer the complete documentation in html format.

Many good books have been written about Perl–see the section below for more details.

Tutorial documents are included in current or upcoming Perl releases include *perltoot* for objects or *perlboot* for a beginner's approach to objects, *perlopentut* for file opening semantics, *perlreftut* for managing references, *perlretut* for regular expressions, *perlthrtut* for threads, *perldebtut* for debugging, and *perlxstut* for linking C and Perl together. There may be more by the time you read this. The following URLs might also be of assistance:

```
    http://perldoc.cpan.org/
    http://www.perldoc.com/
    http://bookmarks.cpan.org/search.cgi?cat=Training%2FTutorials
```

18.1.9 What are the Perl newsgroups on Usenet? Where do I post questions?

Several groups devoted to the Perl language are on Usenet:

```
    comp.lang.perl.announce             Moderated announcement group
    comp.lang.perl.misc                 High traffic general Perl discussion
    comp.lang.perl.moderated        Moderated discussion group
    comp.lang.perl.modules              Use and development of Perl modules
    comp.lang.perl.tk                   Using Tk (and X) from Perl

    comp.infosystems.www.authoring.cgi  Writing CGI scripts for the Web.
```

Some years ago, comp.lang.perl was divided into those groups, and comp.lang.perl itself officially removed. While that group may still be found on some news servers, it is unwise to use it, because postings there will not appear on news servers which honour the official list of group names. Use comp.lang.perl.misc for topics which do not have a more-appropriate specific group.

There is also a Usenet gateway to Perl mailing lists sponsored by perl.org at nntp://nntp.perl.org , a web interface to the same lists at http://nntp.perl.org/group/ and these lists are also available under the `perl.*` hierarchy at http://groups.google.com . Other groups are listed at http://lists.perl.org/ (also known as http://lists.cpan.org/).

A nice place to ask questions is the PerlMonks site, http://www.perlmonks.org/ , or the Perl Beginners mailing list http://lists.perl.org/showlist.cgi?name=beginners .

Note that none of the above are supposed to write your code for you: asking questions about particular problems or general advice is fine, but asking someone to write your code for free is not very cool.

18.1.10 Where should I post source code?

You should post source code to whichever group is most appropriate, but feel free to cross-post to comp.lang.perl.misc. If you want to cross-post to alt.sources, please make sure it follows their posting standards, including setting the Followup-To header line to NOT include alt.sources; see their FAQ (http://www.faqs.org/faqs/alt-sources-intro/) for details.

If you're just looking for software, first use Google (http://www.google.com), Google's usenet search interface (http://groups.google.com), and CPAN Search (http://search.cpan.org). This is faster and more productive than just posting a request.

18.1.11 Perl Books

A number of books on Perl and/or CGI programming are available. A few of these are good, some are OK, but many aren't worth your money. Tom Christiansen maintains a list of these books, some with extensive reviews, at http://www.perl.com/perl/critiques/index.html .

The incontestably definitive reference book on Perl, written by the creator of Perl, is now (July 2000) in its third edition:

```
Programming Perl (the "Camel Book"):
    by Larry Wall, Tom Christiansen, and Jon Orwant
    0-596-00027-8  [3rd edition July 2000]
    http://www.oreilly.com/catalog/pperl3/
(English, translations to several languages are also available)
```

The companion volume to the Camel containing thousands of real-world examples, mini-tutorials, and complete programs is:

```
The Perl Cookbook (the "Ram Book"):
    by Tom Christiansen and Nathan Torkington,
        with Foreword by Larry Wall
    ISBN 1-56592-243-3 [1st Edition August 1998]
    http://perl.oreilly.com/catalog/cookbook/
```

If you're already a seasoned programmer, then the Camel Book might suffice for you to learn Perl from. If you're not, check out the Llama book:

```
Learning Perl (the "Llama Book")
    by Randal L. Schwartz and Tom Phoenix
    ISBN 0-596-00132-0 [3rd edition July 2001]
    http://www.oreilly.com/catalog/lperl3/
```

And for more advanced information on writing larger programs, presented in the same style as the Llama book, continue your education with the Alpaca book:

```
Learning Perl Objects, References, and Modules (the "Alpaca Book")
   by Randal L. Schwartz, with Tom Phoenix (foreword by Damian Conway)
   ISBN 0-596-00478-8 [1st edition June 2003]
   http://www.oreilly.com/catalog/lrnperlorm/
```

If you're not an accidental programmer, but a more serious and possibly even degreed computer scientist who doesn't need as much hand-holding as we try to provide in the Llama, please check out the delightful book

```
Perl: The Programmer's Companion
    by Nigel Chapman
    ISBN 0-471-97563-X [1997, 3rd printing Spring 1998]
    http://www.wiley.com/compbooks/catalog/97563-X.htm
    http://www.wiley.com/compbooks/chapman/perl/perltpc.html (errata etc)
```

If you are more at home in Windows the following is available (though unfortunately rather dated).

```
    Learning Perl on Win32 Systems (the "Gecko Book")
        by Randal L. Schwartz, Erik Olson, and Tom Christiansen,
            with foreword by Larry Wall
        ISBN 1-56592-324-3 [1st edition August 1997]
        http://www.oreilly.com/catalog/lperlwin/
```

Addison-Wesley (http://www.awlonline.com/) and Manning (http://www.manning.com/) are also publishers of some fine Perl books such as *Object Oriented Programming with Perl* by Damian Conway and *Network Programming with Perl* by Lincoln Stein.

An excellent technical book discounter is Bookpool at http://www.bookpool.com/ where a 30% discount or more is not unusual.

What follows is a list of the books that the FAQ authors found personally useful. Your mileage may (but, we hope, probably won't) vary.

Recommended books on (or mostly on) Perl follow.

References

```
        Programming Perl
            by Larry Wall, Tom Christiansen, and Jon Orwant
            ISBN 0-596-00027-8 [3rd edition July 2000]
            http://www.oreilly.com/catalog/pperl3/

        Perl 5 Pocket Reference
        by Johan Vromans
            ISBN 0-596-00032-4 [3rd edition May 2000]
            http://www.oreilly.com/catalog/perlpr3/

        Perl in a Nutshell
        by Ellen Siever, Stephan Spainhour, and Nathan Patwardhan
            ISBN 1-56592-286-7 [1st edition December 1998]
            http://www.oreilly.com/catalog/perlnut/
```

Tutorials

```
        Elements of Programming with Perl
            by Andrew L. Johnson
            ISBN 1-884777-80-5 [1st edition October 1999]
            http://www.manning.com/Johnson/

        Learning Perl
            by Randal L. Schwartz and Tom Phoenix
            ISBN 0-596-00132-0 [3rd edition July 2001]
            http://www.oreilly.com/catalog/lperl3/

        Learning Perl Objects, References, and Modules
           by Randal L. Schwartz, with Tom Phoenix (foreword by Damian Conway)
           ISBN 0-596-00478-8 [1st edition June 2003]
           http://www.oreilly.com/catalog/lrnperlorm/

        Learning Perl on Win32 Systems
            by Randal L. Schwartz, Erik Olson, and Tom Christiansen,
                with foreword by Larry Wall
            ISBN 1-56592-324-3 [1st edition August 1997]
            http://www.oreilly.com/catalog/lperlwin/
```

```
Perl: The Programmer's Companion
    by Nigel Chapman
    ISBN 0-471-97563-X [1997, 3rd printing Spring 1998]
http://www.wiley.com/compbooks/catalog/97563-X.htm
http://www.wiley.com/compbooks/chapman/perl/perltpc.html (errata etc)

Cross-Platform Perl
    by Eric Foster-Johnson
    ISBN 1-55851-483-X [2nd edition September 2000]
    http://www.pconline.com/~erc/perlbook.htm

MacPerl: Power and Ease
    by Vicki Brown and Chris Nandor,
        with foreword by Matthias Neeracher
    ISBN 1-881957-32-2 [1st edition May 1998]
    http://www.macperl.com/ptf_book/
```

Task-Oriented

```
The Perl Cookbook
    by Tom Christiansen and Nathan Torkington
        with foreword by Larry Wall
    ISBN 1-56592-243-3 [1st edition August 1998]
    http://www.oreilly.com/catalog/cookbook/

Effective Perl Programming
    by Joseph Hall
    ISBN 0-201-41975-0 [1st edition 1998]
    http://www.awl.com/
```

Special Topics

```
Mastering Regular Expressions
    by Jeffrey E. F. Friedl
    ISBN 0-596-00289-0 [2nd edition July 2002]
    http://www.oreilly.com/catalog/regex2/

Network Programming with Perl
    by Lincoln Stein
    ISBN 0-201-61571-1 [1st edition 2001]
    http://www.awlonline.com/

Object Oriented Perl
    Damian Conway
        with foreword by Randal L. Schwartz
    ISBN 1-884777-79-1 [1st edition August 1999]
    http://www.manning.com/Conway/

Data Munging with Perl
    Dave Cross
    ISBN 1-930110-00-6 [1st edition 2001]
    http://www.manning.com/cross

Mastering Perl/Tk
    by Steve Lidie and Nancy Walsh
    ISBN 1-56592-716-8 [1st edition January 2002]
    http://www.oreilly.com/catalog/mastperltk/
```

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

18.1.12 Perl in Magazines

The first (and for a long time, only) periodical devoted to All Things Perl, *The Perl Journal* contains tutorials, demonstrations, case studies, announcements, contests, and much more. *TPJ* has columns on web development, databases, Win32 Perl, graphical programming, regular expressions, and networking, and sponsors the Obfuscated Perl Contest and the Perl Poetry Contests. Beginning in November 2002, TPJ moved to a reader-supported monthly e-zine format in which subscribers can download issues as PDF documents. For more details on TPJ, see http://www.tpj.com/

Beyond this, magazines that frequently carry quality articles on Perl are *The Perl Review* (http://www.theperlreview.com), *Unix Review* (http://www.unixreview.com/), *Linux Magazine* (http://www.linuxmagazine.com/), and Usenix's newsletter/magazine to its members, *login:* (http://www.usenix.org/)

The Perl columns of Randal L. Schwartz are available on the web at http://www.stonehenge.com/merlyn/WebTechniques/ , http://www.stonehenge.com/merlyn/UnixReview/ , and http://www.stonehenge.com/merlyn/LinuxMag/ .

18.1.13 Perl on the Net: FTP and WWW Access

To get the best performance, pick a site from the list at http://www.cpan.org/SITES.html . From there you can find the quickest site for you.

You may also use xx.cpan.org where "xx" is the 2-letter country code for your domain; e.g. Australia would use au.cpan.org. [Note: This only applies to countries that host at least one mirror.]

18.1.14 What mailing lists are there for Perl?

Most of the major modules (Tk, CGI, libwww-perl) have their own mailing lists. Consult the documentation that came with the module for subscription information.

A comprehensive list of Perl related mailing lists can be found at:

```
        http://lists.perl.org/
```

18.1.15 Archives of comp.lang.perl.misc

The Google search engine now carries archived and searchable newsgroup content.

http://groups.google.com/groups?group=comp.lang.perl.misc

If you have a question, you can be sure someone has already asked the same question at some point on c.l.p.m. It requires some time and patience to sift through all the content but often you will find the answer you seek.

18.1.16 Where can I buy a commercial version of Perl?

In a real sense, Perl already *is* commercial software: it has a license that you can grab and carefully read to your manager. It is distributed in releases and comes in well-defined packages. There is a very large user community and an extensive literature. The comp.lang.perl.* newsgroups and several of the mailing lists provide free answers to your questions in near real-time. Perl has traditionally been supported by Larry, scores of software designers and developers, and myriad programmers, all working for free to create a useful thing to make life better for everyone.

However, these answers may not suffice for managers who require a purchase order from a company whom they can sue should anything go awry. Or maybe they need very serious hand-holding and contractual obligations. Shrink-wrapped CDs with Perl on them are available from several sources if that will help. For example, many Perl books include a distribution of Perl, as do the O'Reilly Perl Resource Kits (in both the Unix flavor and in the proprietary Microsoft flavor); the free Unix distributions also all come with Perl.

Alternatively, you can purchase commercial incidence based support through the Perl Clinic. The following is a commercial from them:

"The Perl Clinic is a commercial Perl support service operated by ActiveState Tool Corp. and The Ingram Group. The operators have many years of in-depth experience with Perl applications and Perl internals on a wide range of platforms.

"Through our group of highly experienced and well-trained support engineers, we will put our best effort into understanding your problem, providing an explanation of the situation, and a recommendation on how to proceed."

Contact The Perl Clinic at

```
    www.PerlClinic.com

    North America Pacific Standard Time (GMT-8)
    Tel:    1 604 606-4611 hours 8am-6pm
    Fax:    1 604 606-4640

    Europe (GMT)
    Tel:    00 44 1483 862814
    Fax:    00 44 1483 862801
```

See also www.perl.com for updates on tutorials, training, and support.

18.1.17 Where do I send bug reports?

If you are reporting a bug in the perl interpreter or the modules shipped with Perl, use the *perlbug* program in the Perl distribution or mail your report to perlbug@perl.org .

If you are posting a bug with a non-standard port (see the answer to "What platforms is Perl available for?"), a binary distribution, or a non-standard module (such as Tk, CGI, etc), then please see the documentation that came with it to determine the correct place to post bugs.

Read the perlbug(1) man page (perl5.004 or later) for more information.

18.1.18 What is perl.com? Perl Mongers? pm.org? perl.org? cpan.org?

The Perl Home Page at http://www.perl.com/ is currently hosted by The O'Reilly Network, a subsidiary of O'Reilly and Associates.

Perl Mongers is an advocacy organization for the Perl language which maintains the web site http://www.perl.org/ as a general advocacy site for the Perl language.

Perl Mongers uses the pm.org domain for services related to Perl user groups, including the hosting of mailing lists and web sites. See the Perl user group web site at http://www.pm.org/ for more information about joining, starting, or requesting services for a Perl user group.

Perl Mongers also maintain the perl.org domain to provide general support services to the Perl community, including the hosting of mailing lists, web sites, and other services. The web site http://www.perl.org/ is a general advocacy site for the Perl language, and there are many other sub-domains for special topics, such as

```
    http://bugs.perl.org/
    http://history.perl.org/
    http://lists.perl.org/
    http://use.perl.org/
```

http://www.cpan.org/ is the Comprehensive Perl Archive Network, a replicated worlwide repository of Perl software, see the *What is CPAN?* question earlier in this document.

18.2 AUTHOR AND COPYRIGHT

Copyright (c) 1997-2001 Tom Christiansen and Nathan Torkington. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples here are in the public domain. You are permitted and encouraged to use this code and any derivatives thereof in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit to the FAQ would be courteous but is not required.

Chapter 19

perlfaq3

Programming Tools ($Revision: 1.37 $, $Date: 2003/11/24 19:55:50 $)

19.1 DESCRIPTION

This section of the FAQ answers questions related to programmer tools and programming support.

19.1.1 How do I do (anything)?

Have you looked at CPAN (see *perlfaq2*)? The chances are that someone has already written a module that can solve your problem. Have you read the appropriate manpages? Here's a brief index:

```
        Basics          perldata, perlvar, perlsyn, perlop, perlsub
        Execution       perlrun, perldebug
        Functions       perlfunc
        Objects         perlref, perlmod, perlobj, perltie
        Data Structures perlref, perllol, perldsc
        Modules         perlmod, perlmodlib, perlsub
        Regexes         perlre, perlfunc, perlop, perllocale
        Moving to perl5 perltrap, perl
        Linking w/C     perlxstut, perlxs, perlcall, perlguts, perlembed
        Various         http://www.cpan.org/misc/olddoc/FMTEYEWTK.tgz
                        (not a man-page but still useful, a collection
                         of various essays on Perl techniques)
```

A crude table of contents for the Perl manpage set is found in *perltoc*.

19.1.2 How can I use Perl interactively?

The typical approach uses the Perl debugger, described in the perldebug(1) manpage, on an “empty” program, like this:

```
    perl -de 42
```

Now just type in any legal Perl code, and it will be immediately evaluated. You can also examine the symbol table, get stack backtraces, check variable values, set breakpoints, and other operations typically found in symbolic debuggers.

19.1.3 Is there a Perl shell?

The psh (Perl sh) is currently at version 1.8. The Perl Shell is a shell that combines the interactive nature of a Unix shell with the power of Perl. The goal is a full featured shell that behaves as expected for normal shell activity and uses Perl syntax and functionality for control-flow statements and other things. You can get psh at http://www.focusresearch.com/gregor/psh/ .

Zoidberg is a similar project and provides a shell written in perl, configured in perl and operated in perl. It is intended as a login shell and development environment. It can be found at http://zoidberg.sf.net/ or your local CPAN mirror.

The Shell.pm module (distributed with Perl) makes Perl try commands which aren't part of the Perl language as shell commands. perlsh from the source distribution is simplistic and uninteresting, but may still be what you want.

19.1.4 How do I find which modules are installed on my system?

You can use the ExtUtils::Installed module to show all installed distributions, although it can take awhile to do its magic. The standard library which comes with Perl just shows up as "Perl" (although you can get those with Module::CoreList).

```
        use ExtUtils::Installed;

        my $inst    = ExtUtils::Installed->new();
        my @modules = $inst->modules();
```

If you want a list of all of the Perl module filenames, you can use File::Find::Rule.

```
        use File::Find::Rule;

        my @files = File::Find::Rule->file()->name( '*.pm' )->in( @INC );
```

If you do not have that module, you can do the same thing with File::Find which is part of the standard library.

```
    use File::Find;
    my @files;

    find sub { push @files, $File::Find::name if -f _ && /\.pm$/ },
         @INC;

        print join "\n", @files;
```

If you simply need to quickly check to see if a module is available, you can check for its documentation. If you can read the documentation the module is most likely installed. If you cannot read the documentation, the module might not have any (in rare cases).

```
        prompt% perldoc Module::Name
```

You can also try to include the module in a one-liner to see if perl finds it.

```
        perl -MModule::Name -e1
```

19.1.5 How do I debug my Perl programs?

Have you tried `use warnings` or used `-w`? They enable warnings to detect dubious practices.

Have you tried `use strict`? It prevents you from using symbolic references, makes you predeclare any subroutines that you call as bare words, and (probably most importantly) forces you to predeclare your variables with `my`, `our`, or `use vars`.

Did you check the return values of each and every system call? The operating system (and thus Perl) tells you whether they worked, and if not why.

```
open(FH, "> /etc/cantwrite")
  or die "Couldn't write to /etc/cantwrite: $!\n";
```

Did you read *perltrap*? It's full of gotchas for old and new Perl programmers and even has sections for those of you who are upgrading from languages like *awk* and *C*.

Have you tried the Perl debugger, described in *perldebug*? You can step through your program and see what it's doing and thus work out why what it's doing isn't what it should be doing.

19.1.6 How do I profile my Perl programs?

You should get the Devel::DProf module from the standard distribution (or separately on CPAN) and also use Benchmark.pm from the standard distribution. The Benchmark module lets you time specific portions of your code, while Devel::DProf gives detailed breakdowns of where your code spends its time.

Here's a sample use of Benchmark:

```
use Benchmark;

@junk = `cat /etc/motd`;
$count = 10_000;

timethese($count, {
          'map' => sub { my @a = @junk;
                         map { s/a/b/ } @a;
                         return @a },
          'for' => sub { my @a = @junk;
                         for (@a) { s/a/b/ };
                         return @a },
         });
```

This is what it prints (on one machine–your results will be dependent on your hardware, operating system, and the load on your machine):

```
Benchmark: timing 10000 iterations of for, map...
       for:  4 secs ( 3.97 usr  0.01 sys =  3.98 cpu)
       map:  6 secs ( 4.97 usr  0.00 sys =  4.97 cpu)
```

Be aware that a good benchmark is very hard to write. It only tests the data you give it and proves little about the differing complexities of contrasting algorithms.

19.1.7 How do I cross-reference my Perl programs?

The B::Xref module can be used to generate cross-reference reports for Perl programs.

```
  perl -MO=Xref[,OPTIONS] scriptname.plx
```

19.1.8 Is there a pretty-printer (formatter) for Perl?

Perltidy is a Perl script which indents and reformats Perl scripts to make them easier to read by trying to follow the rules of the *perlstyle*. If you write Perl scripts, or spend much time reading them, you will probably find it useful. It is available at http://perltidy.sourceforge.net

Of course, if you simply follow the guidelines in *perlstyle*, you shouldn't need to reformat. The habit of formatting your code as you write it will help prevent bugs. Your editor can and should help you with this. The perl-mode or newer cperl-mode for emacs can provide remarkable amounts of help with most (but not all) code, and even less programmable editors can provide significant assistance. Tom Christiansen and many other VI users swear by the following settings in vi and its clones:

```
    set ai sw=4
    map! ^O {^M}^[O^T
```

Put that in your *.exrc* file (replacing the caret characters with control characters) and away you go. In insert mode, ˆT is for indenting, ˆD is for undenting, and ˆO is for blockdenting– as it were. A more complete example, with comments, can be found at http://www.cpan.org/authors/id/TOMC/scripts/toms.exrc.gz

The a2ps http://www-inf.enst.fr/%7Edemaille/a2ps/black+white.ps.gz does lots of things related to generating nicely printed output of documents, as does enscript at http://people.ssh.fi/mtr/genscript/ .

19.1.9 Is there a ctags for Perl?

Recent versions of ctags do much more than older versions did. EXUBERANT CTAGS is available from http://ctags.sourceforge.net/ and does a good job of making tags files for perl code.

There is also a simple one at http://www.cpan.org/authors/id/TOMC/scripts/ptags.gz which may do the trick. It can be easy to hack this into what you want.

19.1.10 Is there an IDE or Windows Perl Editor?

Perl programs are just plain text, so any editor will do.

If you're on Unix, you already have an IDE–Unix itself. The UNIX philosophy is the philosophy of several small tools that each do one thing and do it well. It's like a carpenter's toolbox.

If you want an IDE, check the following:

Komodo

ActiveState's cross-platform (as of April 2001 Windows and Linux), multi-language IDE has Perl support, including a regular expression debugger and remote debugging (http://www.ActiveState.com/Products/Komodo/index.html). (Visual Perl, a Visual Studio.NET plug-in is currently (early 2001) in beta (http://www.ActiveState.com/Products/VisualPerl/index.html)).

The Object System

(http://www.castlelink.co.uk/object_system/) is a Perl web applications development IDE, apparently for any platform that runs Perl.

Open Perl IDE

(http://open-perl-ide.sourceforge.net/) Open Perl IDE is an integrated development environment for writing and debugging Perl scripts with ActiveState's ActivePerl distribution under Windows 95/98/NT/2000.

PerlBuilder

(http://www.solutionsoft.com/perl.htm) is an integrated development environment for Windows that supports Perl development.

visiPerl+

(http://helpconsulting.net/visiperl/) From Help Consulting, for Windows.

OptiPerl

(http://www.optiperl.com/) is a Windows IDE with simulated CGI environment, including debugger and syntax highlighting editor.

For editors: if you're on Unix you probably have vi or a vi clone already, and possibly an emacs too, so you may not need to download anything. In any emacs the cperl-mode (M-x cperl-mode) gives you perhaps the best available Perl editing mode in any editor.

If you are using Windows, you can use any editor that lets you work with plain text, such as NotePad or WordPad. Word processors, such as Microsoft Word or WordPerfect, typically do not work since they insert all sorts of behind-the-scenes information, although some allow you to save files as "Text Only". You can also download text editors designed specifically for programming, such as Textpad (http://www.textpad.com/) and UltraEdit (http://www.ultraedit.com/), among others.

If you are using MacOS, the same concerns apply. MacPerl (for Classic environments) comes with a simple editor. Popular external editors are BBEdit (http://www.bbedit.com/) or Alpha (http://www.kelehers.org/alpha/). MacOS X users can use Unix editors as well.

GNU Emacs

http://www.gnu.org/software/emacs/windows/ntemacs.html

MicroEMACS

http://www.microemacs.de/

XEmacs

http://www.xemacs.org/Download/index.html

Jed

http://space.mit.edu/˜davis/jed/

or a vi clone such as

Elvis

ftp://ftp.cs.pdx.edu/pub/elvis/ http://www.fh-wedel.de/elvis/

Vile

http://dickey.his.com/vile/vile.html

Vim

http://www.vim.org/

For vi lovers in general, Windows or elsewhere:

```
    http://www.thomer.com/thomer/vi/vi.html
```

nvi (http://www.bostic.com/vi/ , available from CPAN in src/misc/) is yet another vi clone, unfortunately not available for Windows, but in UNIX platforms you might be interested in trying it out, firstly because strictly speaking it is not a vi clone, it is the real vi, or the new incarnation of it, and secondly because you can embed Perl inside it to use Perl as the scripting language. nvi is not alone in this, though: at least also vim and vile offer an embedded Perl.

The following are Win32 multilanguage editor/IDESs that support Perl:

Codewright

http://www.starbase.com/

MultiEdit

http://www.MultiEdit.com/

SlickEdit

http://www.slickedit.com/

There is also a toyedit Text widget based editor written in Perl that is distributed with the Tk module on CPAN. The ptkdb (http://world.std.com/˜aep/ptkdb/) is a Perl/tk based debugger that acts as a development environment of sorts. Perl Composer (http://perlcomposer.sourceforge.net/) is an IDE for Perl/Tk GUI creation.

In addition to an editor/IDE you might be interested in a more powerful shell environment for Win32. Your options include

Bash

from the Cygwin package (http://sources.redhat.com/cygwin/)

Ksh

from the MKS Toolkit (http://www.mks.com/), or the Bourne shell of the U/WIN environment (http://www.research.att.com/sw/tools/uwin/)

Tcsh

ftp://ftp.astron.com/pub/tcsh/ , see also http://www.primate.wisc.edu/software/csh-tcsh-book/

Zsh

ftp://ftp.blarg.net/users/amol/zsh/ , see also http://www.zsh.org/

MKS and U/WIN are commercial (U/WIN is free for educational and research purposes), Cygwin is covered by the GNU Public License (but that shouldn't matter for Perl use). The Cygwin, MKS, and U/WIN all contain (in addition to the shells) a comprehensive set of standard UNIX toolkit utilities.

If you're transferring text files between Unix and Windows using FTP be sure to transfer them in ASCII mode so the ends of lines are appropriately converted.

On Mac OS the MacPerl Application comes with a simple 32k text editor that behaves like a rudimentary IDE. In contrast to the MacPerl Application the MPW Perl tool can make use of the MPW Shell itself as an editor (with no 32k limit).

BBEdit and BBEdit Lite

are text editors for Mac OS that have a Perl sensitivity mode (http://web.barebones.com/).

Alpha

is an editor, written and extensible in Tcl, that nonetheless has built in support for several popular markup and programming languages including Perl and HTML (http://alpha.olm.net/).

Pepper and Pe are programming language sensitive text editors for Mac OS X and BeOS respectively (http://www.hekkelman.com/).

19.1.11 Where can I get Perl macros for vi?

For a complete version of Tom Christiansen's vi configuration file, see http://www.cpan.org/authors/Tom_Christiansen/scripts/toms.exrc.gz , the standard benchmark file for vi emulators. The file runs best with nvi, the current version of vi out of Berkeley, which incidentally can be built with an embedded Perl interpreter–see http://www.cpan.org/src/misc/ .

19.1.12 Where can I get perl-mode for emacs?

Since Emacs version 19 patchlevel 22 or so, there have been both a perl-mode.el and support for the Perl debugger built in. These should come with the standard Emacs 19 distribution.

In the Perl source directory, you'll find a directory called "emacs", which contains a cperl-mode that color-codes keywords, provides context-sensitive help, and other nifty things.

Note that the perl-mode of emacs will have fits with `"main'foo"` (single quote), and mess up the indentation and highlighting. You are probably using `"main::foo"` in new Perl code anyway, so this shouldn't be an issue.

19.1.13 How can I use curses with Perl?

The Curses module from CPAN provides a dynamically loadable object module interface to a curses library. A small demo can be found at the directory http://www.cpan.org/authors/Tom_Christiansen/scripts/rep.gz ; this program repeats a command and updates the screen as needed, rendering **rep ps axu** similar to **top**.

19.1.14 How can I use X or Tk with Perl?

Tk is a completely Perl-based, object-oriented interface to the Tk toolkit that doesn't force you to use Tcl just to get at Tk. Sx is an interface to the Athena Widget set. Both are available from CPAN. See the directory http://www.cpan.org/modules/by-category/08_User_Interfaces/

Invaluable for Perl/Tk programming are the Perl/Tk FAQ at http://w4.lns.cornell.edu/%7Epvhp/ptk/ptkTOC.html , the Perl/Tk Reference Guide available at http://www.cpan.org/authors/Stephen_O_Lidie/ , and the online manpages at http://www-users.cs.umn.edu/%7Eamundson/perl/perltk/toc.html .

19.1.15 How can I generate simple menus without using CGI or Tk?

The http://www.cpan.org/authors/id/SKUNZ/perlmenu.v4.0.tar.gz module, which is curses-based, can help with this.

19.1.16 How can I make my Perl program run faster?

The best way to do this is to come up with a better algorithm. This can often make a dramatic difference. Jon Bentley's book *Programming Pearls* (that's not a misspelling!) has some good tips on optimization, too. Advice on benchmarking boils down to: benchmark and profile to make sure you're optimizing the right part, look for better algorithms instead of microtuning your code, and when all else fails consider just buying faster hardware. You will probably want to read the answer to the earlier question "How do I profile my Perl programs?" if you haven't done so already.

A different approach is to autoload seldom-used Perl code. See the AutoSplit and AutoLoader modules in the standard distribution for that. Or you could locate the bottleneck and think about writing just that part in C, the way we used to take bottlenecks in C code and write them in assembler. Similar to rewriting in C, modules that have critical sections can be written in C (for instance, the PDL module from CPAN).

If you're currently linking your perl executable to a shared *libc.so*, you can often gain a 10-25% performance benefit by rebuilding it to link with a static libc.a instead. This will make a bigger perl executable, but your Perl programs (and programmers) may thank you for it. See the *INSTALL* file in the source distribution for more information.

The undump program was an ancient attempt to speed up Perl program by storing the already-compiled form to disk. This is no longer a viable option, as it only worked on a few architectures, and wasn't a good solution anyway.

19.1.17 How can I make my Perl program take less memory?

When it comes to time-space tradeoffs, Perl nearly always prefers to throw memory at a problem. Scalars in Perl use more memory than strings in C, arrays take more than that, and hashes use even more. While there's still a lot to be done, recent releases have been addressing these issues. For example, as of 5.004, duplicate hash keys are shared amongst all hashes using them, so require no reallocation.

In some cases, using substr() or vec() to simulate arrays can be highly beneficial. For example, an array of a thousand booleans will take at least 20,000 bytes of space, but it can be turned into one 125-byte bit vector–a considerable memory savings. The standard Tie::SubstrHash module can also help for certain types of data structure. If you're working with specialist data structures (matrices, for instance) modules that implement these in C may use less memory than equivalent Perl modules.

Another thing to try is learning whether your Perl was compiled with the system malloc or with Perl's builtin malloc. Whichever one it is, try using the other one and see whether this makes a difference. Information about malloc is in the *INSTALL* file in the source distribution. You can find out whether you are using perl's malloc by typing `perl -V:usemymalloc`.

Of course, the best way to save memory is to not do anything to waste it in the first place. Good programming practices can go a long way toward this:

- Don't slurp!

  Don't read an entire file into memory if you can process it line by line. Or more concretely, use a loop like this:

  ```
  #
  # Good Idea
  #
  while (<FILE>) {
     # ...
  }
  ```

  instead of this:

  ```
  #
  # Bad Idea
  #
  @data = <FILE>;
  foreach (@data) {
      # ...
  }
  ```

  When the files you're processing are small, it doesn't much matter which way you do it, but it makes a huge difference when they start getting larger.

- Use map and grep selectively

  Remember that both map and grep expect a LIST argument, so doing this:

  ```
  @wanted = grep {/pattern/} <FILE>;
  ```

  will cause the entire file to be slurped. For large files, it's better to loop:

  ```
  while (<FILE>) {
          push(@wanted, $_) if /pattern/;
  }
  ```

- Avoid unnecessary quotes and stringification

  Don't quote large strings unless absolutely necessary:

  ```
  my $copy = "$large_string";
  ```

  makes 2 copies of $large_string (one for $copy and another for the quotes), whereas

  ```
  my $copy = $large_string;
  ```

  only makes one copy.

  Ditto for stringifying large arrays:

  ```
  {
          local $, = "\n";
          print @big_array;
  }
  ```

  is much more memory-efficient than either

  ```
  print join "\n", @big_array;
  ```

or

```
        {
                local $" = "\n";
                print "@big_array";
        }
```

- Pass by reference

  Pass arrays and hashes by reference, not by value. For one thing, it's the only way to pass multiple lists or hashes (or both) in a single call/return. It also avoids creating a copy of all the contents. This requires some judgment, however, because any changes will be propagated back to the original data. If you really want to mangle (er, modify) a copy, you'll have to sacrifice the memory needed to make one.

- Tie large variables to disk.

  For "big" data stores (i.e. ones that exceed available memory) consider using one of the DB modules to store it on disk instead of in RAM. This will incur a penalty in access time, but that's probably better than causing your hard disk to thrash due to massive swapping.

19.1.18 Is it safe to return a reference to local or lexical data?

Yes. Perl's garbage collection system takes care of this so everything works out right.

```
    sub makeone {
        my @a = ( 1 .. 10 );
        return \@a;
    }

    for ( 1 .. 10 ) {
        push @many, makeone();
    }

    print $many[4][5], "\n";

    print "@many\n";
```

19.1.19 How can I free an array or hash so my program shrinks?

You usually can't. On most operating systems, memory allocated to a program can never be returned to the system. That's why long-running programs sometimes re-exec themselves. Some operating systems (notably, systems that use mmap(2) for allocating large chunks of memory) can reclaim memory that is no longer used, but on such systems, perl must be configured and compiled to use the OS's malloc, not perl's.

However, judicious use of my() on your variables will help make sure that they go out of scope so that Perl can free up that space for use in other parts of your program. A global variable, of course, never goes out of scope, so you can't get its space automatically reclaimed, although undef()ing and/or delete()ing it will achieve the same effect. In general, memory allocation and de-allocation isn't something you can or should be worrying about much in Perl, but even this capability (preallocation of data types) is in the works.

19.1.20 How can I make my CGI script more efficient?

Beyond the normal measures described to make general Perl programs faster or smaller, a CGI program has additional issues. It may be run several times per second. Given that each time it runs it will need to be re-compiled and will often allocate a megabyte or more of system memory, this can be a killer. Compiling into C **isn't going to help you** because the process start-up overhead is where the bottleneck is.

There are two popular ways to avoid this overhead. One solution involves running the Apache HTTP server (available from http://www.apache.org/) with either of the mod_perl or mod_fastcgi plugin modules.

With mod_perl and the Apache::Registry module (distributed with mod_perl), httpd will run with an embedded Perl interpreter which pre-compiles your script and then executes it within the same address space without forking. The Apache extension also gives Perl access to the internal server API, so modules written in Perl can do just about anything a module written in C can. For more on mod_perl, see http://perl.apache.org/

With the FCGI module (from CPAN) and the mod_fastcgi module (available from http://www.fastcgi.com/) each of your Perl programs becomes a permanent CGI daemon process.

Both of these solutions can have far-reaching effects on your system and on the way you write your CGI programs, so investigate them with care.

See http://www.cpan.org/modules/by-category/15_World_Wide_Web_HTML_HTTP_CGI/ .

A non-free, commercial product, "The Velocity Engine for Perl", (http://www.binevolve.com/ or http://www.binevolve.com/velocigen/) might also be worth looking at. It will allow you to increase the performance of your Perl programs, running programs up to 25 times faster than normal CGI Perl when running in persistent Perl mode or 4 to 5 times faster without any modification to your existing CGI programs. Fully functional evaluation copies are available from the web site.

19.1.21 How can I hide the source for my Perl program?

Delete it. :-) Seriously, there are a number of (mostly unsatisfactory) solutions 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–only by people with access to the filesystem.) So you have to leave the permissions at the socially friendly 0755 level.

Some people 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 (Starting from Perl 5.8 the Filter::Simple and Filter::Util::Call modules are included in the standard distribution), but any decent programmer will be able to decrypt it. You can try using the byte code compiler and interpreter described below, but the curious might still 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 (true of every language, not just Perl).

It is very easy to recover the source of Perl programs. You simply feed the program to the perl interpreter and use the modules in the B:: hierarchy. The B::Deparse module should be able to defeat most attempts to hide source. Again, this is not unique to 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." We are not lawyers, of course, so you should see a lawyer if you want to be sure your license's wording will stand up in court.

19.1.22 How can I compile my Perl program into byte code or C?

Malcolm Beattie has written a multifunction backend compiler, available from CPAN, that can do both these things. It is included in the perl5.005 release, but is still considered experimental. This means it's fun to play with if you're a programmer but not really for people looking for turn-key solutions.

Merely compiling into C does not in and of itself guarantee that your code will run very much faster. That's because except for lucky cases where a lot of native type inferencing is possible, the normal Perl run-time system is still present and so your program will take just as long to run and be just as big. Most programs save little more than compilation time, leaving execution no more than 10-30% faster. A few rare programs actually benefit significantly (even running several times faster), but this takes some tweaking of your code.

You'll probably be astonished to learn that the current version of the compiler generates a compiled form of your script whose executable is just as big as the original perl executable, and then some. That's because as currently written, all programs are prepared for a full eval() statement. You can tremendously reduce this cost by building a shared *libperl.so* library and linking against that. See the *INSTALL* podfile in the Perl source distribution for details. If you link your main perl binary with this, it will make it minuscule. For example, on one author's system, */usr/bin/perl* is only 11k in size!

In general, the compiler will do nothing to make a Perl program smaller, faster, more portable, or more secure. In fact, it can make your situation worse. The executable will be bigger, your VM system may take longer to load the whole thing, the binary is fragile and hard to fix, and compilation never stopped software piracy in the form of crackers, viruses, or bootleggers. The real advantage of the compiler is merely packaging, and once you see the size of what it makes (well, unless you use a shared *libperl.so*), you'll probably want a complete Perl install anyway.

19.1.23 How can I compile Perl into Java?

You can also integrate Java and Perl with the Perl Resource Kit from O'Reilly and Associates. See http://www.oreilly.com/catalog/prkunix/ .

Perl 5.6 comes with Java Perl Lingo, or JPL. JPL, still in development, allows Perl code to be called from Java. See jpl/README in the Perl source tree.

19.1.24 How can I get `#!perl` to work on [MS-DOS,NT,...]?

For OS/2 just use

```
    extproc perl -S -your_switches
```

as the first line in `*.cmd` file (`-S` due to a bug in cmd.exe's 'extproc' handling). For DOS one should first invent a corresponding batch file and codify it in `ALTERNATE_SHEBANG` (see the *dosish.h* file in the source distribution for more information).

The Win95/NT installation, when using the ActiveState port of Perl, will modify the Registry to associate the `.pl` extension with the perl interpreter. If you install another port, perhaps even building your own Win95/NT Perl from the standard sources by using a Windows port of gcc (e.g., with cygwin or mingw32), then you'll have to modify the Registry yourself. In addition to associating `.pl` with the interpreter, NT people can use: `SET PATHEXT=%PATHEXT%;.PL` to let them run the program `install-linux.pl` merely by typing `install-linux`.

Macintosh Perl programs will have the appropriate Creator and Type, so that double-clicking them will invoke the Perl application.

IMPORTANT!: Whatever you do, PLEASE don't get frustrated, and just throw the perl interpreter into your cgi-bin directory, in order to get your programs working for a web server. This is an EXTREMELY big security risk. Take the time to figure out how to do it correctly.

19.1.25 Can I write useful Perl programs on the command line?

Yes. Read *perlrun* for more information. Some examples follow. (These assume standard Unix shell quoting rules.)

```
    # sum first and last fields
    perl -lane 'print $F[0] + $F[-1]' *

    # identify text files
    perl -le 'for(@ARGV) {print if -f && -T _}' *
    # remove (most) comments from C program
    perl -0777 -pe 's{/\*.*?\*/}{}gs' foo.c

    # make file a month younger than today, defeating reaper daemons
    perl -e '$X=24*60*60; utime(time(),time() + 30 * $X,@ARGV)' *

    # find first unused uid
    perl -le '$i++ while getpwuid($i); print $i'

    # display reasonable manpath
    echo $PATH | perl -nl -072 -e '
        s![^/+]*$!man!&&-d&&!$s{$_}++&&push@m,$_;END{print"@m"}'
```

OK, the last one was actually an Obfuscated Perl Contest entry. :-)

19.1.26 Why don't Perl one-liners work on my DOS/Mac/VMS system?

The problem is usually that the command interpreters on those systems have rather different ideas about quoting than the Unix shells under which the one-liners were created. On some systems, you may have to change single-quotes to double ones, which you must *NOT* do on Unix or Plan9 systems. You might also have to change a single % to a %%.

For example:

```
    # Unix
    perl -e 'print "Hello world\n"'

    # DOS, etc.
    perl -e "print \"Hello world\n\""

    # Mac
    print "Hello world\n"
     (then Run "Myscript" or Shift-Command-R)

    # MPW
    perl -e 'print "Hello world\n"'

    # VMS
    perl -e "print ""Hello world\n"""
```

The problem is that none of these examples are reliable: they depend on the command interpreter. Under Unix, the first two often work. Under DOS, it's entirely possible that neither works. If 4DOS was the command shell, you'd probably have better luck like this:

```
  perl -e "print <Ctrl-x>"Hello world\n<Ctrl-x>""
```

Under the Mac, it depends which environment you are using. The MacPerl shell, or MPW, is much like Unix shells in its support for several quoting variants, except that it makes free use of the Mac's non-ASCII characters as control characters.

Using qq(), q(), and qx(), instead of "double quotes", 'single quotes', and `backticks`, may make one-liners easier to write.

There is no general solution to all of this. It is a mess.

[Some of this answer was contributed by Kenneth Albanowski.]

19.1.27 Where can I learn about CGI or Web programming in Perl?

For modules, get the CGI or LWP modules from CPAN. For textbooks, see the two especially dedicated to web stuff in the question on books. For problems and questions related to the web, like “Why do I get 500 Errors” or “Why doesn’t it run from the browser right when it runs fine on the command line”, see the troubleshooting guides and references in *perlfaq9* or in the CGI MetaFAQ:

```
        http://www.perl.org/CGI_MetaFAQ.html
```

19.1.28 Where can I learn about object-oriented Perl programming?

A good place to start is *perltoot*, and you can use *perlobj*, *perlboot*, *perltoot*, *perltooc*, and *perlbot* for reference. (If you are using really old Perl, you may not have all of these, try http://www.perldoc.com/ , but consider upgrading your perl.)

A good book on OO on Perl is the "Object-Oriented Perl" by Damian Conway from Manning Publications, http://www.manning.com/Conway/index.html

19.1.29 Where can I learn about linking C with Perl? [h2xs, xsubpp]

If you want to call C from Perl, start with *perlxstut*, moving on to *perlxs*, *xsubpp*, and *perlguts*. If you want to call Perl from C, then read *perlembed*, *perlcall*, and *perlguts*. Don’t forget that you can learn a lot from looking at how the authors of existing extension modules wrote their code and solved their problems.

19.1.30 I’ve read perlembed, perlguts, etc., but I can’t embed perl in my C program; what am I doing wrong?

Download the ExtUtils::Embed kit from CPAN and run ‘make test’. If the tests pass, read the pods again and again and again. If they fail, see *perlbug* and send a bug report with the output of `make test TEST_VERBOSE=1` along with `perl -V`.

19.1.31 When I tried to run my script, I got this message. What does it mean?

A complete list of Perl’s error messages and warnings with explanatory text can be found in *perldiag*. You can also use the splain program (distributed with Perl) to explain the error messages:

```
    perl program 2>diag.out
    splain [-v] [-p] diag.out
```

or change your program to explain the messages for you:

```
    use diagnostics;
```

or

```
    use diagnostics -verbose;
```

19.1.32 What’s MakeMaker?

This module (part of the standard Perl distribution) is designed to write a Makefile for an extension module from a Makefile.PL. For more information, see *ExtUtils::MakeMaker*.

19.2 AUTHOR AND COPYRIGHT

Copyright (c) 1997-2002 Tom Christiansen and Nathan Torkington. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples here are in the public domain. You are permitted and encouraged to use this code and any derivatives thereof in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit to the FAQ would be courteous but is not required.

Chapter 20

perlfaq4

Data Manipulation ($Revision: 1.54 $, $Date: 2003/11/30 00:50:08 $)

20.1 DESCRIPTION

This section of the FAQ answers questions related to manipulating numbers, dates, strings, arrays, hashes, and miscellaneous data issues.

20.2 Data: Numbers

20.2.1 Why am I getting long decimals (eg, 19.9499999999999) instead of the numbers I should be getting (eg, 19.95)?

Internally, your computer represents floating-point numbers in binary. Digital (as in powers of two) computers cannot store all numbers exactly. Some real numbers lose precision in the process. This is a problem with how computers store numbers and affects all computer languages, not just Perl.

perlnumber show the gory details of number representations and conversions.

To limit the number of decimal places in your numbers, you can use the printf or sprintf function. See the "Floating Point Arithmetic" for more details.

```
        printf "%.2f", 10/3;

        my $number = sprintf "%.2f", 10/3;
```

20.2.2 Why is int() broken?

Your int() is most probably working just fine. It's the numbers that aren't quite what you think.

First, see the above item "Why am I getting long decimals (eg, 19.9499999999999) instead of the numbers I should be getting (eg, 19.95)?".

For example, this

```
    print int(0.6/0.2-2), "\n";
```

will in most computers print 0, not 1, because even such simple numbers as 0.6 and 0.2 cannot be presented exactly by floating-point numbers. What you think in the above as 'three' is really more like 2.9999999999999995559.

20.2.3 Why isn't my octal data interpreted correctly?

Perl only understands octal and hex numbers as such when they occur as literals in your program. Octal literals in perl must start with a leading "0" and hexadecimal literals must start with a leading "0x". If they are read in from somewhere and assigned, no automatic conversion takes place. You must explicitly use oct() or hex() if you want the values converted to decimal. oct() interprets hex ("0x350"), octal ("0350" or even without the leading "0", like "377") and binary ("0b1010") numbers, while hex() only converts hexadecimal ones, with or without a leading "0x", like "0x255", "3A", "ff", or "deadbeef". The inverse mapping from decimal to octal can be done with either the "%o" or "%O" sprintf() formats.

This problem shows up most often when people try using chmod(), mkdir(), umask(), or sysopen(), which by widespread tradition typically take permissions in octal.

```
chmod(644,  $file); # WRONG
chmod(0644, $file); # right
```

Note the mistake in the first line was specifying the decimal literal 644, rather than the intended octal literal 0644. The problem can be seen with:

```
printf("%#o",644); # prints 01204
```

Surely you had not intended `chmod(01204, $file);` - did you? If you want to use numeric literals as arguments to chmod() et al. then please try to express them as octal constants, that is with a leading zero and with the following digits restricted to the set 0..7.

20.2.4 Does Perl have a round() function? What about ceil() and floor()? Trig functions?

Remember that int() merely truncates toward 0. For rounding to a certain number of digits, sprintf() or printf() is usually the easiest route.

```
printf("%.3f", 3.1415926535);       # prints 3.142
```

The POSIX module (part of the standard Perl distribution) implements ceil(), floor(), and a number of other mathematical and trigonometric functions.

```
use POSIX;
$ceil   = ceil(3.5);                        # 4
$floor  = floor(3.5);                       # 3
```

In 5.000 to 5.003 perls, trigonometry was done in the Math::Complex module. With 5.004, the Math::Trig module (part of the standard Perl distribution) implements the trigonometric functions. Internally it uses the Math::Complex module and some functions can break out from the real axis into the complex plane, for example the inverse sine of 2.

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.

To see why, notice how you'll still have an issue on half-way-point alternation:

```
for ($i = 0; $i < 1.01; $i += 0.05) { printf "%.1f ",$i}

0.0 0.1 0.1 0.2 0.2 0.2 0.3 0.3 0.4 0.4 0.5 0.5 0.6 0.7 0.7
0.8 0.8 0.9 0.9 1.0 1.0
```

Don't blame Perl. It's the same as in C. IEEE says we have to do this. Perl numbers whose absolute values are integers under 2**31 (on 32 bit machines) will work pretty much like mathematical integers. Other numbers are not guaranteed.

20.2.5 How do I convert between numeric representations/bases/radixes?

As always with Perl there is more than one way to do it. Below are a few examples of approaches to making common conversions between number representations. This is intended to be representational rather than exhaustive.

Some of the examples below use the Bit::Vector module from CPAN. The reason you might choose Bit::Vector over the perl built in functions is that it works with numbers of ANY size, that it is optimized for speed on some operations, and for at least some programmers the notation might be familiar.

How do I convert hexadecimal into decimal

Using perl's built in conversion of 0x notation:

```
$dec = 0xDEADBEEF;
```

Using the hex function:

```
$dec = hex("DEADBEEF");
```

Using pack:

```
$dec = unpack("N", pack("H8", substr("0" x 8 . "DEADBEEF", -8)));
```

Using the CPAN module Bit::Vector:

```
use Bit::Vector;
$vec = Bit::Vector->new_Hex(32, "DEADBEEF");
$dec = $vec->to_Dec();
```

How do I convert from decimal to hexadecimal

Using sprintf:

```
$hex = sprintf("%X", 3735928559); # upper case A-F
$hex = sprintf("%x", 3735928559); # lower case a-f
```

Using unpack:

```
$hex = unpack("H*", pack("N", 3735928559));
```

Using Bit::Vector:

```
use Bit::Vector;
$vec = Bit::Vector->new_Dec(32, -559038737);
$hex = $vec->to_Hex();
```

And Bit::Vector supports odd bit counts:

```
use Bit::Vector;
$vec = Bit::Vector->new_Dec(33, 3735928559);
$vec->Resize(32); # suppress leading 0 if unwanted
$hex = $vec->to_Hex();
```

How do I convert from octal to decimal

Using Perl's built in conversion of numbers with leading zeros:

```
    $dec = 033653337357; # note the leading 0!
```

Using the oct function:

```
    $dec = oct("33653337357");
```

Using Bit::Vector:

```
    use Bit::Vector;
    $vec = Bit::Vector->new(32);
    $vec->Chunk_List_Store(3, split(//, reverse "33653337357"));
    $dec = $vec->to_Dec();
```

How do I convert from decimal to octal

Using sprintf:

```
    $oct = sprintf("%o", 3735928559);
```

Using Bit::Vector:

```
    use Bit::Vector;
    $vec = Bit::Vector->new_Dec(32, -559038737);
    $oct = reverse join('', $vec->Chunk_List_Read(3));
```

How do I convert from binary to decimal

Perl 5.6 lets you write binary numbers directly with the 0b notation:

```
    $number = 0b10110110;
```

Using oct:

```
    my $input = "10110110";
    $decimal = oct( "0b$input" );
```

Using pack and ord:

```
    $decimal = ord(pack('B8', '10110110'));
```

Using pack and unpack for larger strings:

```
    $int = unpack("N", pack("B32",
        substr("0" x 32 . "11110101011011011111011101111", -32)));
    $dec = sprintf("%d", $int);

    # substr() is used to left pad a 32 character string with zeros.
```

Using Bit::Vector:

```
    $vec = Bit::Vector->new_Bin(32, "11011110101011011011111011101111");
    $dec = $vec->to_Dec();
```

How do I convert from decimal to binary

Using sprintf (perl 5.6+):

```
    $bin = sprintf("%b", 3735928559);
```

Using unpack:

```
    $bin = unpack("B*", pack("N", 3735928559));
```

Using Bit::Vector:

```
    use Bit::Vector;
    $vec = Bit::Vector->new_Dec(32, -559038737);
    $bin = $vec->to_Bin();
```

The remaining transformations (e.g. hex -> oct, bin -> hex, etc.) are left as an exercise to the inclined reader.

20.2.6 Why doesn't & work the way I want it to?

The behavior of binary arithmetic operators depends on whether they're used on numbers or strings. The operators treat a string as a series of bits and work with that (the string `"3"` is the bit pattern `00110011`). The operators work with the binary form of a number (the number 3 is treated as the bit pattern `00000011`).

So, saying `11 & 3` performs the "and" operation on numbers (yielding 3). Saying `"11" & "3"` performs the "and" operation on strings (yielding `"1"`).

Most problems with & and | arise because the programmer thinks they have a number but really it's a string. The rest arise because the programmer says:

```
if ("\020\020" & "\101\101") {
    # ...
}
```

but a string consisting of two null bytes (the result of `"\020\020" & "\101\101"`) is not a false value in Perl. You need:

```
if ( ("\020\020" & "\101\101") !~ /[^\000]/) {
    # ...
}
```

20.2.7 How do I multiply matrices?

Use the Math::Matrix or Math::MatrixReal modules (available from CPAN) or the PDL extension (also available from CPAN).

20.2.8 How do I perform an operation on a series of integers?

To call a function on each element in an array, and collect the results, use:

```
@results = map { my_func($_) } @array;
```

For example:

```
@triple = map { 3 * $_ } @single;
```

To call a function on each element of an array, but ignore the results:

```
foreach $iterator (@array) {
    some_func($iterator);
}
```

To call a function on each integer in a (small) range, you **can** use:

```
    @results = map { some_func($_) } (5 .. 25);
```

but you should be aware that the `..` operator creates an array of all integers in the range. This can take a lot of memory for large ranges. Instead use:

```
    @results = ();
    for ($i=5; $i < 500_005; $i++) {
        push(@results, some_func($i));
    }
```

This situation has been fixed in Perl5.005. Use of `..` in a `for` loop will iterate over the range, without creating the entire range.

```
    for my $i (5 .. 500_005) {
        push(@results, some_func($i));
    }
```

will not create a list of 500,000 integers.

20.2.9 How can I output Roman numerals?

Get the http://www.cpan.org/modules/by-module/Roman module.

20.2.10 Why aren't my random numbers random?

If you're using a version of Perl before 5.004, you must call `srand` once at the start of your program to seed the random number generator.

```
         BEGIN { srand() if $] < 5.004 }
```

5.004 and later automatically call `srand` at the beginning. Don't call `srand` more than once—you make your numbers less random, rather than more.

Computers are good at being predictable and bad at being random (despite appearances caused by bugs in your programs :-). see the *random* article in the "Far More Than You Ever Wanted To Know" collection in http://www.cpan.org/misc/olddoc/FMTEYEWTK.tgz , courtesy of Tom Phoenix, talks more about this. John von Neumann said, "Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin."

If you want numbers that are more random than `rand` with `srand` provides, you should also check out the Math::TrulyRandom module from CPAN. It uses the imperfections in your system's timer to generate random numbers, but this takes quite a while. If you want a better pseudorandom generator than comes with your operating system, look at "Numerical Recipes in C" at http://www.nr.com/ .

20.2.11 How do I get a random number between X and Y?

`rand($x)` returns a number such that `0 <= rand($x) < $x`. Thus what you want to have perl figure out is a random number in the range from 0 to the difference between your *X* and *Y*.

That is, to get a number between 10 and 15, inclusive, you want a random number between 0 and 5 that you can then add to 10.

```
    my $number = 10 + int rand( 15-10+1 );
```

Hence you derive the following simple function to abstract that. It selects a random integer between the two given integers (inclusive), For example: `random_int_in(50,120)`.

```
sub random_int_in ($$) {
  my($min, $max) = @_;
   # Assumes that the two arguments are integers themselves!
  return $min if $min == $max;
  ($min, $max) = ($max, $min)  if  $min > $max;
  return $min + int rand(1 + $max - $min);
}
```

20.3 Data: Dates

20.3.1 How do I find the day or week of the year?

The localtime function returns the day of the week. Without an argument localtime uses the current time.

```
$day_of_year = (localtime)[7];
```

The POSIX module can also format a date as the day of the year or week of the year.

```
use POSIX qw/strftime/;
my $day_of_year  = strftime "%j", localtime;
my $week_of_year = strftime "%W", localtime;
```

To get the day of year for any date, use the Time::Local module to get a time in epoch seconds for the argument to localtime.

```
use POSIX qw/strftime/;
use Time::Local;
my $week_of_year = strftime "%W",
        localtime( timelocal( 0, 0, 0, 18, 11, 1987 ) );
```

The Date::Calc module provides two functions for to calculate these.

```
use Date::Calc;
my $day_of_year  = Day_of_Year(  1987, 12, 18 );
my $week_of_year = Week_of_Year( 1987, 12, 18 );
```

20.3.2 How do I find the current century or millennium?

Use the following simple functions:

```
sub get_century    {
    return int((((localtime(shift || time))[5] + 1999))/100);
}
sub get_millennium {
    return 1+int((((localtime(shift || time))[5] + 1899))/1000);
}
```

On some systems, the POSIX module's strftime() function has been extended in a non-standard way to use a `%C` format, which they sometimes claim is the "century". It isn't, because on most such systems, this is only the first two digits of the four-digit year, and thus cannot be used to reliably determine the current century or millennium.

20.3.3 How can I compare two dates and find the difference?

If you're storing your dates as epoch seconds then simply subtract one from the other. If you've got a structured date (distinct year, day, month, hour, minute, seconds values), then for reasons of accessibility, simplicity, and efficiency, merely use either timelocal or timegm (from the Time::Local module in the standard distribution) to reduce structured dates to epoch seconds. However, if you don't know the precise format of your dates, then you should probably use either of the Date::Manip and Date::Calc modules from CPAN before you go hacking up your own parsing routine to handle arbitrary date formats.

20.3.4 How can I take a string and turn it into epoch seconds?

If it's a regular enough string that it always has the same format, you can split it up and pass the parts to `timelocal` in the standard Time::Local module. Otherwise, you should look into the Date::Calc and Date::Manip modules from CPAN.

20.3.5 How can I find the Julian Day?

Use the Time::JulianDay module (part of the Time-modules bundle available from CPAN.)

Before you immerse yourself too deeply in this, be sure to verify that it is the *Julian* Day you really want. Are you interested in a way of getting serial days so that you just can tell how many days they are apart or so that you can do also other date arithmetic? If you are interested in performing date arithmetic, this can be done using modules Date::Manip or Date::Calc.

There is too many details and much confusion on this issue to cover in this FAQ, but the term is applied (correctly) to a calendar now supplanted by the Gregorian Calendar, with the Julian Calendar failing to adjust properly for leap years on centennial years (among other annoyances). The term is also used (incorrectly) to mean: [1] days in the Gregorian Calendar; and [2] days since a particular starting time or 'epoch', usually 1970 in the Unix world and 1980 in the MS-DOS/Windows world. If you find that it is not the first meaning that you really want, then check out the Date::Manip and Date::Calc modules. (Thanks to David Cassell for most of this text.)

20.3.6 How do I find yesterday's date?

If you only need to find the date (and not the same time), you can use the Date::Calc module.

```
        use Date::Calc qw(Today Add_Delta_Days);

        my @date = Add_Delta_Days( Today(), -1 );

        print "@date\n";
```

Most people try to use the time rather than the calendar to figure out dates, but that assumes that your days are twenty-four hours each. For most people, there are two days a year when they aren't: the switch to and from summer time throws this off. Russ Allbery offers this solution.

```
    sub yesterday {
                my $now  = defined $_[0] ? $_[0] : time;
                my $then = $now - 60 * 60 * 24;
                my $ndst = (localtime $now)[8] > 0;
                my $tdst = (localtime $then)[8] > 0;
                $then - ($tdst - $ndst) * 60 * 60;
                }
```

Should give you "this time yesterday" in seconds since epoch relative to the first argument or the current time if no argument is given and suitable for passing to localtime or whatever else you need to do with it. $ndst is whether we're currently in daylight savings time; $tdst is whether the point 24 hours ago was in daylight savings time. If $tdst and $ndst are the same, a boundary wasn't crossed, and the correction will subtract 0. If $tdst is 1 and $ndst is 0, subtract an hour more from yesterday's time since we gained an extra hour while going off daylight savings time. If $tdst is 0 and $ndst is 1, subtract a negative hour (add an hour) to yesterday's time since we lost an hour.

All of this is because during those days when one switches off or onto DST, a "day" isn't 24 hours long; it's either 23 or 25.

The explicit settings of $ndst and $tdst are necessary because localtime only says it returns the system tm struct, and the system tm struct at least on Solaris doesn't guarantee any particular positive value (like, say, 1) for isdst, just a positive value. And that value can potentially be negative, if DST information isn't available (this sub just treats those cases like no DST).

Note that between 2am and 3am on the day after the time zone switches off daylight savings time, the exact hour of "yesterday" corresponding to the current hour is not clearly defined. Note also that if used between 2am and 3am the day after the change to daylight savings time, the result will be between 3am and 4am of the previous day; it's arguable whether this is correct.

This sub does not attempt to deal with leap seconds (most things don't).

20.3.7 Does Perl have a Year 2000 problem? Is Perl Y2K compliant?

Short answer: No, Perl does not have a Year 2000 problem. Yes, Perl is Y2K compliant (whatever that means). The programmers you've hired to use it, however, probably are not.

Long answer: The question belies a true understanding of the issue. Perl is just as Y2K compliant as your pencil–no more, and no less. Can you use your pencil to write a non-Y2K-compliant memo? Of course you can. Is that the pencil's fault? Of course it isn't.

The date and time functions supplied with Perl (gmtime and localtime) supply adequate information to determine the year well beyond 2000 (2038 is when trouble strikes for 32-bit machines). The year returned by these functions when used in a list context is the year minus 1900. For years between 1910 and 1999 this *happens* to be a 2-digit decimal number. To avoid the year 2000 problem simply do not treat the year as a 2-digit number. It isn't.

When gmtime() and localtime() are used in scalar context they return a timestamp string that contains a fully-expanded year. For example, `$timestamp = gmtime(1005613200)` sets $timestamp to "Tue Nov 13 01:00:00 2001". There's no year 2000 problem here.

That doesn't mean that Perl can't be used to create non-Y2K compliant programs. It can. But so can your pencil. It's the fault of the user, not the language. At the risk of inflaming the NRA: "Perl doesn't break Y2K, people do." See http://language.perl.com/news/y2k.html for a longer exposition.

20.4 Data: Strings

20.4.1 How do I validate input?

The answer to this question is usually a regular expression, perhaps with auxiliary logic. See the more specific questions (numbers, mail addresses, etc.) for details.

20.4.2 How do I unescape a string?

It depends just what you mean by "escape". URL escapes are dealt with in *perlfaq9*. Shell escapes with the backslash (\) character are removed with

```
    s/\\(.)/$1/g;
```

This won't expand `"\n"` or `"\t"` or any other special escapes.

20.4.3 How do I remove consecutive pairs of characters?

To turn "abbcccd" into "abccd":

```
s/(.)\1/$1/g;       # add /s to include newlines
```

Here's a solution that turns "abbcccd" to "abcd":

```
y///cs;     # y == tr, but shorter :-)
```

20.4.4 How do I expand function calls in a string?

This is documented in *perlref*. In general, this is fraught with quoting and readability problems, but it is possible. To interpolate a subroutine call (in list context) into a string:

```
print "My sub returned @{[mysub(1,2,3)]} that time.\n";
```

See also "How can I expand variables in text strings?" in this section of the FAQ.

20.4.5 How do I find matching/nesting anything?

This isn't something that can be done in one regular expression, no matter how complicated. To find something between two single characters, a pattern like `/x([^x]*)x/` will get the intervening bits in $1. For multiple ones, then something more like `/alpha(.*?)omega/` would be needed. But none of these deals with nested patterns. For balanced expressions using `(`, `{`, `[` or `<` as delimiters, use the CPAN module Regexp::Common, or see (??{ code }) in *perlre*. For other cases, you'll have to write a parser.

If you are serious about writing a parser, there are a number of modules or oddities that will make your life a lot easier. There are the CPAN modules Parse::RecDescent, Parse::Yapp, and Text::Balanced; and the byacc program. Starting from perl 5.8 the Text::Balanced is part of the standard distribution.

One simple destructive, inside-out approach that you might try is to pull out the smallest nesting parts one at a time:

```
while (s/BEGIN((?:(?!BEGIN)(?!END).)*)END//gs) {
    # do something with $1
}
```

A more complicated and sneaky approach is to make Perl's regular expression engine do it for you. This is courtesy Dean Inada, and rather has the nature of an Obfuscated Perl Contest entry, but it really does work:

```
# $_ contains the string to parse
# BEGIN and END are the opening and closing markers for the
# nested text.

@( = ('(','');
@) = (')','');
($re=$_)=~s/((BEGIN)|(END)|.)/$)[!$3]\Q$1\E$([!$2]/gs;
@$ = (eval{/$re/},$@!~/unmatched/i);
print join("\n",@$[0..$#$]) if( $$[-1] );
```

20.4.6 How do I reverse a string?

Use reverse() in scalar context, as documented in reverse in *perlfunc*.

```
$reversed = reverse $string;
```

20.4.7 How do I expand tabs in a string?

You can do it yourself:

```
1 while $string =~ s/\t+/' ' x (length($&) * 8 - length($`) % 8)/e;
```

Or you can just use the Text::Tabs module (part of the standard Perl distribution).

```
use Text::Tabs;
@expanded_lines = expand(@lines_with_tabs);
```

20.4.8 How do I reformat a paragraph?

Use Text::Wrap (part of the standard Perl distribution):

```
use Text::Wrap;
print wrap("\t", '  ', @paragraphs);
```

The paragraphs you give to Text::Wrap should not contain embedded newlines. Text::Wrap doesn't justify the lines (flush-right).

Or use the CPAN module Text::Autoformat. Formatting files can be easily done by making a shell alias, like so:

```
alias fmt="perl -i -MText::Autoformat -n0777 \
    -e 'print autoformat $_, {all=>1}' $*"
```

See the documentation for Text::Autoformat to appreciate its many capabilities.

20.4.9 How can I access or change N characters of a string?

You can access the first characters of a string with substr(). To get the first character, for example, start at position 0 and grab the string of length 1.

```
    $string = "Just another Perl Hacker";
$first_char = substr( $string, 0, 1 );  #  'J'
```

To change part of a string, you can use the optional fourth argument which is the replacement string.

```
substr( $string, 13, 4, "Perl 5.8.0" );
```

You can also use substr() as an lvalue.

```
substr( $string, 13, 4 ) =  "Perl 5.8.0";
```

20.4.10 How do I change the Nth occurrence of something?

You have to keep track of N yourself. For example, let's say you want to change the fifth occurrence of "whoever" or "whomever" into "whosoever" or "whomsoever", case insensitively. These all assume that $_ contains the string to be altered.

```
$count = 0;
s{((whom?)ever)}{
    ++$count == 5           # is it the 5th?
        ? "${2}soever"      # yes, swap
        : $1                # renege and leave it there
}ige;
```

In the more general case, you can use the /g modifier in a while loop, keeping count of matches.

```
$WANT = 3;
$count = 0;
$_ = "One fish two fish red fish blue fish";
while (/(\w+)\s+fish\b/gi) {
    if (++$count == $WANT) {
        print "The third fish is a $1 one.\n";
    }
}
```

That prints out: "The third fish is a red one." You can also use a repetition count and repeated pattern like this:

```
/(?:\w+\s+fish\s+){2}(\w+)\s+fish/i;
```

20.4.11 How can I count the number of occurrences of a substring within a string?

There are a number of ways, with varying efficiency. If you want a count of a certain single character (X) within a string, you can use the tr/// function like so:

```
$string = "ThisXlineXhasXsomeXx'sXinXit";
$count = ($string =~ tr/X//);
print "There are $count X characters in the string";
```

This is fine if you are just looking for a single character. However, if you are trying to count multiple character substrings within a larger string, tr/// won't work. What you can do is wrap a while() loop around a global pattern match. For example, let's count negative integers:

```
$string = "-9 55 48 -2 23 -76 4 14 -44";
while ($string =~ /-\d+/g) { $count++ }
print "There are $count negative numbers in the string";
```

Another version uses a global match in list context, then assigns the result to a scalar, producing a count of the number of matches.

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

20.4.12 How do I capitalize all the words on one line?

To make the first letter of each word upper case:

```
        $line =~ s/\b(\w)/\U$1/g;
```

This has the strange effect of turning "`don't do it`" into "`Don'T Do It`". Sometimes you might want this. Other times you might need a more thorough solution (Suggested by brian d foy):

```
    $string =~ s/ (
                 (^\w)    #at the beginning of the line
                   |      # or
                 (\s\w)   #preceded by whitespace
                   )
                /\U$1/xg;
    $string =~ /([\w']+)/\u\L$1/g;
```

To make the whole line upper case:

```
        $line = uc($line);
```

To force each word to be lower case, with the first letter upper case:

```
        $line =~ s/(\w+)/\u\L$1/g;
```

You can (and probably should) enable locale awareness of those characters by placing a `use locale` pragma in your program. See *perllocale* for endless details on locales.

This is sometimes referred to as putting something into "title case", but that's not quite accurate. Consider the proper capitalization of the movie *Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb*, for example.

Damian Conway's *Text::Autoformat* module provides some smart case transformations:

```
    use Text::Autoformat;
    my $x = "Dr. Strangelove or: How I Learned to Stop ".
      "Worrying and Love the Bomb";

    print $x, "\n";
    for my $style (qw( sentence title highlight ))
    {
        print autoformat($x, { case => $style }), "\n";
    }
```

20.4.13 How can I split a [character] delimited string except when inside [character]?

Several modules can handle this sort of pasing—Text::Balanced, Text::CVS, Text::CVS_XS, and Text::ParseWords, among others.

Take the example case of trying to split a string that is comma-separated into its different fields. You can't use `split(/,/)` because you shouldn't split if the comma is inside quotes. For example, take a data line like this:

```
    SAR001,"","Cimetrix, Inc","Bob Smith","CAM",N,8,1,0,7,"Error, Core Dumped"
```

Due to the restriction of the quotes, this is a fairly complex problem. Thankfully, we have Jeffrey Friedl, author of *Mastering Regular Expressions*, to handle these for us. He suggests (assuming your string is contained in $text):

```
 @new = ();
 push(@new, $+) while $text =~ m{
     "([^\"\\]*(?:\\.[^\"\\]*)*)",?  # groups the phrase inside the quotes
   | ([^,]+),?
   | ,
 }gx;
 push(@new, undef) if substr($text,-1,1) eq ',';
```

If you want to represent quotation marks inside a quotation-mark-delimited field, escape them with backslashes (eg, `"like \"this\""`.

Alternatively, the Text::ParseWords module (part of the standard Perl distribution) lets you say:

```
use Text::ParseWords;
@new = quotewords(",", 0, $text);
```

There's also a Text::CSV (Comma-Separated Values) module on CPAN.

20.4.14 How do I strip blank space from the beginning/end of a string?

Although the simplest approach would seem to be

```
$string =~ s/^\s*(.*?)\s*$/$1/;
```

not only is this unnecessarily slow and destructive, it also fails with embedded newlines. It is much faster to do this operation in two steps:

```
$string =~ s/^\s+//;
$string =~ s/\s+$//;
```

Or more nicely written as:

```
for ($string) {
    s/^\s+//;
    s/\s+$//;
}
```

This idiom takes advantage of the `foreach` loop's aliasing behavior to factor out common code. You can do this on several strings at once, or arrays, or even the values of a hash if you use a slice:

```
# trim whitespace in the scalar, the array,
# and all the values in the hash
foreach ($scalar, @array, @hash{keys %hash}) {
    s/^\s+//;
    s/\s+$//;
}
```

20.4.15 How do I pad a string with blanks or pad a number with zeroes?

In the following examples, `$pad_len` is the length to which you wish to pad the string, `$text` or `$num` contains the string to be padded, and `$pad_char` contains the padding character. You can use a single character string constant instead of the `$pad_char` variable if you know what it is in advance. And in the same way you can use an integer in place of `$pad_len` if you know the pad length in advance.

The simplest method uses the `sprintf` function. It can pad on the left or right with blanks and on the left with zeroes and it will not truncate the result. The `pack` function can only pad strings on the right with blanks and it will truncate the result to a maximum length of `$pad_len`.

```
# Left padding a string with blanks (no truncation):
    $padded = sprintf("%${pad_len}s", $text);
    $padded = sprintf("%*s", $pad_len, $text);  # same thing

# Right padding a string with blanks (no truncation):
    $padded = sprintf("%-${pad_len}s", $text);
    $padded = sprintf("%-*s", $pad_len, $text); # same thing

# Left padding a number with 0 (no truncation):
    $padded = sprintf("%0${pad_len}d", $num);
    $padded = sprintf("%0*d", $pad_len, $num); # same thing

# Right padding a string with blanks using pack (will truncate):
$padded = pack("A$pad_len",$text);
```

If you need to pad with a character other than blank or zero you can use one of the following methods. They all generate a pad string with the `x` operator and combine that with `$text`. These methods do not truncate `$text`.

Left and right padding with any character, creating a new string:

```
$padded = $pad_char x ( $pad_len - length( $text ) ) . $text;
$padded = $text . $pad_char x ( $pad_len - length( $text ) );
```

Left and right padding with any character, modifying `$text` directly:

```
substr( $text, 0, 0 ) = $pad_char x ( $pad_len - length( $text ) );
$text .= $pad_char x ( $pad_len - length( $text ) );
```

20.4.16 How do I extract selected columns from a string?

Use substr() or unpack(), both documented in *perlfunc*. If you prefer thinking in terms of columns instead of widths, you can use this kind of thing:

```
# determine the unpack format needed to split Linux ps output
# arguments are cut columns
my $fmt = cut2fmt(8, 14, 20, 26, 30, 34, 41, 47, 59, 63, 67, 72);

sub cut2fmt {
    my(@positions) = @_;
    my $template  = '';
    my $lastpos   = 1;
    for my $place (@positions) {
        $template .= "A" . ($place - $lastpos) . " ";
        $lastpos   = $place;
    }
    $template .= "A*";
    return $template;
}
```

20.4.17 How do I find the soundex value of a string?

Use the standard Text::Soundex module distributed with Perl. Before you do so, you may want to determine whether 'soundex' is in fact what you think it is. Knuth's soundex algorithm compresses words into a small space, and so it does not necessarily distinguish between two words which you might want to appear separately. For example, the last names 'Knuth' and 'Kant' are both mapped to the soundex code K530. If Text::Soundex does not do what you are looking for, you might want to consider the String::Approx module available at CPAN.

20.4.18 How can I expand variables in text strings?

Let’s assume that you have a string like:

```
    $text = 'this has a $foo in it and a $bar';
```

If those were both global variables, then this would suffice:

```
    $text =~ s/\$(\w+)/${$1}/g;  # no /e needed
```

But since they are probably lexicals, or at least, they could be, you’d have to do this:

```
    $text =~ s/(\$\w+)/$1/eeg;
    die if $@;                  # needed /ee, not /e
```

It’s probably better in the general case to treat those variables as entries in some special hash. For example:

```
    %user_defs = (
        foo  => 23,
        bar  => 19,
    );
    $text =~ s/\$(\w+)/$user_defs{$1}/g;
```

See also “How do I expand function calls in a string?” in this section of the FAQ.

20.4.19 What’s wrong with always quoting "$ vars"?

The problem is that those double-quotes force stringification– coercing numbers and references into strings–even when you don’t want them to be strings. Think of it this way: double-quote expansion is used to produce new strings. If you already have a string, why do you need more?

If you get used to writing odd things like these:

```
    print "$var";       # BAD
    $new = "$old";      # BAD
    somefunc("$var");   # BAD
```

You’ll be in trouble. Those should (in 99.8% of the cases) be the simpler and more direct:

```
    print $var;
    $new = $old;
    somefunc($var);
```

Otherwise, besides slowing you down, you’re going to break code when the thing in the scalar is actually neither a string nor a number, but a reference:

```
    func(\@array);
    sub func {
        my $aref = shift;
        my $oref = "$aref";  # WRONG
    }
```

You can also get into subtle problems on those few operations in Perl that actually do care about the difference between a string and a number, such as the magical ++ autoincrement operator or the syscall() function.

Stringification also destroys arrays.

```
    @lines = `command`;
    print "@lines";             # WRONG - extra blanks
    print @lines;               # right
```

20.4.20 Why don't my <<HERE documents work?

Check for these three things:

There must be no space after the << part.

There (probably) should be a semicolon at the end.

You can't (easily) have any space in front of the tag.

If you want to indent the text in the here document, you can do this:

```
    # all in one
    ($VAR = <<HERE_TARGET) =~ s/^\s+//gm;
        your text
        goes here
    HERE_TARGET
```

But the HERE_TARGET must still be flush against the margin. If you want that indented also, you'll have to quote in the indentation.

```
    ($quote = <<'    FINIS') =~ s/^\s+//gm;
            ...we will have peace, when you and all your works have
            perished--and the works of your dark master to whom you
            would deliver us. You are a liar, Saruman, and a corrupter
            of men's hearts.  --Theoden in /usr/src/perl/taint.c
        FINIS
    $quote =~ s/\s+--/\n--/;
```

A nice general-purpose fixer-upper function for indented here documents follows. It expects to be called with a here document as its argument. It looks to see whether each line begins with a common substring, and if so, strips that substring off. Otherwise, it takes the amount of leading whitespace found on the first line and removes that much off each subsequent line.

```
    sub fix {
        local $_ = shift;
        my ($white, $leader);  # common whitespace and common leading string
        if (/^\s*(?:([^\w\s]+)(\s*).*\n)(?:\s*\1\2?.*\n)+$/) {
            ($white, $leader) = ($2, quotemeta($1));
        } else {
            ($white, $leader) = (/^(\s+)/, '');
        }
        s/^\s*?$leader(?:$white)?//gm;
        return $_;
    }
```

This works with leading special strings, dynamically determined:

```
    $remember_the_main = fix<<'    MAIN_INTERPRETER_LOOP';
        @@@ int
        @@@ runops() {
        @@@     SAVEI32(runlevel);
        @@@     runlevel++;
        @@@     while ( op = (*op->op_ppaddr)() );
        @@@     TAINT_NOT;
        @@@     return 0;
        @@@ }
    MAIN_INTERPRETER_LOOP
```

Or with a fixed amount of leading whitespace, with remaining indentation correctly preserved:

```
$poem = fix<<EVER_ON_AND_ON;
   Now far ahead the Road has gone,
      And I must follow, if I can,
   Pursuing it with eager feet,
      Until it joins some larger way
   Where many paths and errands meet.
      And whither then? I cannot say.
            --Bilbo in /usr/src/perl/pp_ctl.c
EVER_ON_AND_ON
```

20.5 Data: Arrays

20.5.1 What is the difference between a list and an array?

An array has a changeable length. A list does not. An array is something you can push or pop, while a list is a set of values. Some people make the distinction that a list is a value while an array is a variable. Subroutines are passed and return lists, you put things into list context, you initialize arrays with lists, and you foreach() across a list. `@` variables are arrays, anonymous arrays are arrays, arrays in scalar context behave like the number of elements in them, subroutines access their arguments through the array `@_`, and push/pop/shift only work on arrays.

As a side note, there's no such thing as a list in scalar context. When you say

```
$scalar = (2, 5, 7, 9);
```

you're using the comma operator in scalar context, so it uses the scalar comma operator. There never was a list there at all! This causes the last value to be returned: 9.

20.5.2 What is the difference between $ array[1] and @array[1]?

The former is a scalar value; the latter an array slice, making it a list with one (scalar) value. You should use $ when you want a scalar value (most of the time) and @ when you want a list with one scalar value in it (very, very rarely; nearly never, in fact).

Sometimes it doesn't make a difference, but sometimes it does. For example, compare:

```
$good[0] = `some program that outputs several lines`;
```

with

```
@bad[0]  = `same program that outputs several lines`;
```

The `use warnings` pragma and the **-w** flag will warn you about these matters.

20.5.3 How can I remove duplicate elements from a list or array?

There are several possible ways, depending on whether the array is ordered and whether you wish to preserve the ordering.

a)

If @in is sorted, and you want @out to be sorted: (this assumes all true values in the array)

```
$prev = "not equal to $in[0]";
@out = grep($_ ne $prev && ($prev = $_, 1), @in);
```

This is nice in that it doesn't use much extra memory, simulating uniq(1)'s behavior of removing only adjacent duplicates. The ", 1" guarantees that the expression is true (so that grep picks it up) even if the $_ is 0, "", or undef.

b)

If you don't know whether @in is sorted:

```
undef %saw;
@out = grep(!$saw{$_}++, @in);
```

c)

Like (b), but @in contains only small integers:

```
@out = grep(!$saw[$_]++, @in);
```

d)

A way to do (b) without any loops or greps:

```
undef %saw;
@saw{@in} = ();
@out = sort keys %saw;  # remove sort if undesired
```

e)

Like (d), but @in contains only small positive integers:

```
undef @ary;
@ary[@in] = @in;
@out = grep {defined} @ary;
```

But perhaps you should have been using a hash all along, eh?

20.5.4 How can I tell whether a certain element is contained in a list or array?

Hearing the word "in" is an *in*dication that you probably should have used a hash, not a list or array, to store your data. Hashes are designed to answer this question quickly and efficiently. Arrays aren't.

That being said, there are several ways to approach this. If you are going to make this query many times over arbitrary string values, the fastest way is probably to invert the original array and maintain a hash whose keys are the first array's values.

```
@blues = qw/azure cerulean teal turquoise lapis-lazuli/;
%is_blue = ();
for (@blues) { $is_blue{$_} = 1 }
```

Now you can check whether $is_blue{$some_color}. It might have been a good idea to keep the blues all in a hash in the first place.

If the values are all small integers, you could use a simple indexed array. This kind of an array will take up less space:

```
@primes = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31);
@is_tiny_prime = ();
for (@primes) { $is_tiny_prime[$_] = 1 }
# or simply  @istiny_prime[@primes] = (1) x @primes;
```

Now you check whether $is_tiny_prime[$some_number].

If the values in question are integers instead of strings, you can save quite a lot of space by using bit strings instead:

```
@articles = ( 1..10, 150..2000, 2017 );
undef $read;
for (@articles) { vec($read,$_,1) = 1 }
```

Now check whether `vec($read,$n,1)` is true for some `$n`.

Please do not use

```
($is_there) = grep $_ eq $whatever, @array;
```

or worse yet

```
($is_there) = grep /$whatever/, @array;
```

These are slow (checks every element even if the first matches), inefficient (same reason), and potentially buggy (what if there are regex characters in $whatever?). If you're only testing once, then use:

```
$is_there = 0;
foreach $elt (@array) {
    if ($elt eq $elt_to_find) {
        $is_there = 1;
        last;
    }
}
if ($is_there) { ... }
```

20.5.5 How do I compute the difference of two arrays? How do I compute the intersection of two arrays?

Use a hash. Here's code to do both and more. It assumes that each element is unique in a given array:

```
@union = @intersection = @difference = ();
%count = ();
foreach $element (@array1, @array2) { $count{$element}++ }
foreach $element (keys %count) {
    push @union, $element;
    push @{ $count{$element} > 1 ? \@intersection : \@difference }, $element;
}
```

Note that this is the *symmetric difference*, that is, all elements in either A or in B but not in both. Think of it as an xor operation.

20.5.6 How do I test whether two arrays or hashes are equal?

The following code works for single-level arrays. It uses a stringwise comparison, and does not distinguish defined versus undefined empty strings. Modify if you have other needs.

```
$are_equal = compare_arrays(\@frogs, \@toads);

sub compare_arrays {
    my ($first, $second) = @_;
    no warnings;  # silence spurious -w undef complaints
    return 0 unless @$first == @$second;
    for (my $i = 0; $i < @$first; $i++) {
        return 0 if $first->[$i] ne $second->[$i];
    }
    return 1;
}
```

For multilevel structures, you may wish to use an approach more like this one. It uses the CPAN module FreezeThaw:

```
use FreezeThaw qw(cmpStr);
@a = @b = ( "this", "that", [ "more", "stuff" ] );

printf "a and b contain %s arrays\n",
    cmpStr(\@a, \@b) == 0
        ? "the same"
        : "different";
```

This approach also works for comparing hashes. Here we'll demonstrate two different answers:

```
use FreezeThaw qw(cmpStr cmpStrHard);

%a = %b = ( "this" => "that", "extra" => [ "more", "stuff" ] );
$a{EXTRA} = \%b;
$b{EXTRA} = \%a;

printf "a and b contain %s hashes\n",
    cmpStr(\%a, \%b) == 0 ? "the same" : "different";

printf "a and b contain %s hashes\n",
    cmpStrHard(\%a, \%b) == 0 ? "the same" : "different";
```

The first reports that both those the hashes contain the same data, while the second reports that they do not. Which you prefer is left as an exercise to the reader.

20.5.7 How do I find the first array element for which a condition is true?

To find the first array element which satisfies a condition, you can use the first() function in the List::Util module, which comes with Perl 5.8. This example finds the first element that contains "Perl".

```
        use List::Util qw(first);

        my $element = first { /Perl/ } @array;
```

If you cannot use List::Util, you can make your own loop to do the same thing. Once you find the element, you stop the loop with last.

```
        my $found;
        foreach my $element ( @array )
                {
                if( /Perl/ ) { $found = $element; last }
                }
```

If you want the array index, you can iterate through the indices and check the array element at each index until you find one that satisfies the condition.

```
        my( $found, $index ) = ( undef, -1 );
    for( $i = 0; $i < @array; $i++ )
        {
        if( $array[$i] =~ /Perl/ )
                {
                $found = $array[$i];
                $index = $i;
                last;
                }
        }
```

20.5.8 How do I handle linked lists?

In general, you usually don't need a linked list in Perl, since with regular arrays, you can push and pop or shift and unshift at either end, or you can use splice to add and/or remove arbitrary number of elements at arbitrary points. Both pop and shift are both O(1) operations on Perl's dynamic arrays. In the absence of shifts and pops, push in general needs to reallocate on the order every log(N) times, and unshift will need to copy pointers each time.

If you really, really wanted, you could use structures as described in *perldsc* or *perltoot* and do just what the algorithm book tells you to do. For example, imagine a list node like this:

```
$node = {
    VALUE => 42,
    LINK  => undef,
};
```

You could walk the list this way:

```
print "List: ";
for ($node = $head;  $node; $node = $node->{LINK}) {
    print $node->{VALUE}, " ";
}
print "\n";
```

You could add to the list this way:

```
my ($head, $tail);
$tail = append($head, 1);       # grow a new head
for $value ( 2 .. 10 ) {
    $tail = append($tail, $value);
}

sub append {
    my($list, $value) = @_;
    my $node = { VALUE => $value };
    if ($list) {
        $node->{LINK} = $list->{LINK};
        $list->{LINK} = $node;
    } else {
        $_[0] = $node;      # replace caller's version
    }
    return $node;
}
```

But again, Perl's built-in are virtually always good enough.

20.5.9 How do I handle circular lists?

Circular lists could be handled in the traditional fashion with linked lists, or you could just do something like this with an array:

```
unshift(@array, pop(@array));  # the last shall be first
push(@array, shift(@array));   # and vice versa
```

20.5.10 How do I shuffle an array randomly?

If you either have Perl 5.8.0 or later installed, or if you have Scalar-List-Utils 1.03 or later installed, you can say:

```
use List::Util 'shuffle';

    @shuffled = shuffle(@list);
```

If not, you can use a Fisher-Yates shuffle.

```
sub fisher_yates_shuffle {
    my $deck = shift;  # $deck is a reference to an array
    my $i = @$deck;
    while ($i--) {
        my $j = int rand ($i+1);
        @$deck[$i,$j] = @$deck[$j,$i];
    }
}

# shuffle my mpeg collection
#
my @mpeg = <audio/*/*.mp3>;
fisher_yates_shuffle( \@mpeg );    # randomize @mpeg in place
print @mpeg;
```

Note that the above implementation shuffles an array in place, unlike the List::Util::shuffle() which takes a list and returns a new shuffled list.

You've probably seen shuffling algorithms that work using splice, randomly picking another element to swap the current element with

```
srand;
@new = ();
@old = 1 .. 10;  # just a demo
while (@old) {
    push(@new, splice(@old, rand @old, 1));
}
```

This is bad because splice is already O(N), and since you do it N times, you just invented a quadratic algorithm; that is, O(N**2). This does not scale, although Perl is so efficient that you probably won't notice this until you have rather largish arrays.

20.5.11 How do I process/modify each element of an array?

Use for/foreach:

```
for (@lines) {
    s/foo/bar/;     # change that word
    y/XZ/ZX/;       # swap those letters
}
```

Here's another; let's compute spherical volumes:

```
for (@volumes = @radii) {   # @volumes has changed parts
    $_ **= 3;
    $_ *= (4/3) * 3.14159;  # this will be constant folded
}
```

which can also be done with map() which is made to transform one list into another:

```
        @volumes = map {$_ ** 3 * (4/3) * 3.14159} @radii;
```

If you want to do the same thing to modify the values of the hash, you can use the `values` function. As of Perl 5.6 the values are not copied, so if you modify $orbit (in this case), you modify the value.

```
    for $orbit ( values %orbits ) {
        ($orbit **= 3) *= (4/3) * 3.14159;
    }
```

Prior to perl 5.6 `values` returned copies of the values, so older perl code often contains constructions such as `@orbits{keys %orbits}` instead of `values %orbits` where the hash is to be modified.

20.5.12 How do I select a random element from an array?

Use the rand() function (see rand in *perlfunc*):

```
    $index   = rand @array;
    $element = $array[$index];
```

Or, simply: my $element = $array[rand @array];

20.5.13 How do I permute N elements of a list?

Use the List::Permutor module on CPAN. If the list is actually an array, try the Algorithm::Permute module (also on CPAN). It's written in XS code and is very efficient.

```
        use Algorithm::Permute;
        my @array = 'a'..'d';
        my $p_iterator = Algorithm::Permute->new ( \@array );
        while (my @perm = $p_iterator->next) {
           print "next permutation: (@perm)\n";
        }
```

For even faster execution, you could do:

```
   use Algorithm::Permute;
   my @array = 'a'..'d';
   Algorithm::Permute::permute {
      print "next permutation: (@array)\n";
   } @array;
```

Here's a little program that generates all permutations of all the words on each line of input. The algorithm embodied in the permute() function is discussed in Volume 4 (still unpublished) of Knuth's *The Art of Computer Programming* and will work on any list:

```
        #!/usr/bin/perl -n
        # Fischer-Kause ordered permutation generator
        sub permute (&@) {
                my $code = shift;
                my @idx = 0..$#_;
                while ( $code->(@_[@idx]) ) {
                        my $p = $#idx;
                        --$p while $idx[$p-1] > $idx[$p];
                        my $q = $p or return;
                        push @idx, reverse splice @idx, $p;
                        ++$q while $idx[$p-1] > $idx[$q];
                        @idx[$p-1,$q]=@idx[$q,$p-1];
                }
        }

        permute {print"@_\n"} split;
```

20.5.14 How do I sort an array by (anything)?

Supply a comparison function to sort() (described in sort in *perlfunc*):

```
    @list = sort { $a <=> $b } @list;
```

The default sort function is cmp, string comparison, which would sort `(1, 2, 10)` into `(1, 10, 2)`. <=>, used above, is the numerical comparison operator.

If you have a complicated function needed to pull out the part you want to sort on, then don't do it inside the sort function. Pull it out first, because the sort BLOCK can be called many times for the same element. Here's an example of how to pull out the first word after the first number on each item, and then sort those words case-insensitively.

```
    @idx = ();
    for (@data) {
        ($item) = /\d+\s*(\S+)/;
        push @idx, uc($item);
    }
    @sorted = @data[ sort { $idx[$a] cmp $idx[$b] } 0 .. $#idx ];
```

which could also be written this way, using a trick that's come to be known as the Schwartzian Transform:

```
    @sorted = map  { $_->[0] }
              sort { $a->[1] cmp $b->[1] }
              map  { [ $_, uc( (/\d+\s*(\S+)/)[0]) ] } @data;
```

If you need to sort on several fields, the following paradigm is useful.

```
    @sorted = sort { field1($a) <=> field1($b) ||
                     field2($a) cmp field2($b) ||
                     field3($a) cmp field3($b)
                   }     @data;
```

This can be conveniently combined with precalculation of keys as given above.

See the *sort* article in the "Far More Than You Ever Wanted To Know" collection in http://www.cpan.org/misc/olddoc/FMTEYEWTK.tgz for more about this approach.

See also the question below on sorting hashes.

20.5.15 How do I manipulate arrays of bits?

Use pack() and unpack(), or else vec() and the bitwise operations.

For example, this sets $vec to have bit N set if $ints[N] was set:

```
    $vec = '';
    foreach(@ints) { vec($vec,$_,1) = 1 }
```

Here's how, given a vector in $vec, you can get those bits into your @ints array:

```
    sub bitvec_to_list {
        my $vec = shift;
        my @ints;
        # Find null-byte density then select best algorithm
        if ($vec =~ tr/\0// / length $vec > 0.95) {
            use integer;
            my $i;
            # This method is faster with mostly null-bytes
            while($vec =~ /[^\0]/g ) {
                $i = -9 + 8 * pos $vec;
                push @ints, $i if vec($vec, ++$i, 1);
                push @ints, $i if vec($vec, ++$i, 1);
                push @ints, $i if vec($vec, ++$i, 1);
                push @ints, $i if vec($vec, ++$i, 1);
                push @ints, $i if vec($vec, ++$i, 1);
                push @ints, $i if vec($vec, ++$i, 1);
                push @ints, $i if vec($vec, ++$i, 1);
                push @ints, $i if vec($vec, ++$i, 1);
            }
        } else {
            # This method is a fast general algorithm
            use integer;
            my $bits = unpack "b*", $vec;
            push @ints, 0 if $bits =~ s/^(\d)// && $1;
            push @ints, pos $bits while($bits =~ /1/g);
        }
        return \@ints;
    }
```

This method gets faster the more sparse the bit vector is. (Courtesy of Tim Bunce and Winfried Koenig.)

You can make the while loop a lot shorter with this suggestion from Benjamin Goldberg:

```
        while($vec =~ /[^\0]+/g ) {
           push @ints, grep vec($vec, $_, 1), $-[0] * 8 .. $+[0] * 8;
        }
```

Or use the CPAN module Bit::Vector:

```
    $vector = Bit::Vector->new($num_of_bits);
    $vector->Index_List_Store(@ints);
    @ints = $vector->Index_List_Read();
```

Bit::Vector provides efficient methods for bit vector, sets of small integers and "big int" math.

Here's a more extensive illustration using vec():

```
# vec demo
$vector = "\xff\x0f\xef\xfe";
print "Ilya's string \\xff\\x0f\\xef\\xfe represents the number ",
    unpack("N", $vector), "\n";
$is_set = vec($vector, 23, 1);
print "Its 23rd bit is ", $is_set ? "set" : "clear", ".\n";
pvec($vector);

set_vec(1,1,1);
set_vec(3,1,1);
set_vec(23,1,1);

set_vec(3,1,3);
set_vec(3,2,3);
set_vec(3,4,3);
set_vec(3,4,7);
set_vec(3,8,3);
set_vec(3,8,7);

set_vec(0,32,17);
set_vec(1,32,17);

sub set_vec {
    my ($offset, $width, $value) = @_;
    my $vector = '';
    vec($vector, $offset, $width) = $value;
    print "offset=$offset width=$width value=$value\n";
    pvec($vector);
}

sub pvec {
    my $vector = shift;
    my $bits = unpack("b*", $vector);
    my $i = 0;
    my $BASE = 8;

    print "vector length in bytes: ", length($vector), "\n";
    @bytes = unpack("A8" x length($vector), $bits);
    print "bits are: @bytes\n\n";
}
```

20.5.16 Why does defined() return true on empty arrays and hashes?

The short story is that you should probably only use defined on scalars or functions, not on aggregates (arrays and hashes). See defined in *perlfunc* in the 5.004 release or later of Perl for more detail.

20.6 Data: Hashes (Associative Arrays)

20.6.1 How do I process an entire hash?

Use the each() function (see each in *perlfunc*) if you don't care whether it's sorted:

```
while ( ($key, $value) = each %hash) {
    print "$key = $value\n";
}
```

If you want it sorted, you'll have to use foreach() on the result of sorting the keys as shown in an earlier question.

20.6.2 What happens if I add or remove keys from a hash while iterating over it?

Don't do that. :-)

[lwall] In Perl 4, you were not allowed to modify a hash at all while iterating over it. In Perl 5 you can delete from it, but you still can't add to it, because that might cause a doubling of the hash table, in which half the entries get copied up to the new top half of the table, at which point you've totally bamboozled the iterator code. Even if the table doesn't double, there's no telling whether your new entry will be inserted before or after the current iterator position.

Either treasure up your changes and make them after the iterator finishes or use keys to fetch all the old keys at once, and iterate over the list of keys.

20.6.3 How do I look up a hash element by value?

Create a reverse hash:

```
    %by_value = reverse %by_key;
    $key = $by_value{$value};
```

That's not particularly efficient. It would be more space-efficient to use:

```
    while (($key, $value) = each %by_key) {
        $by_value{$value} = $key;
    }
```

If your hash could have repeated values, the methods above will only find one of the associated keys. This may or may not worry you. If it does worry you, you can always reverse the hash into a hash of arrays instead:

```
     while (($key, $value) = each %by_key) {
         push @{$key_list_by_value{$value}}, $key;
     }
```

20.6.4 How can I know how many entries are in a hash?

If you mean how many keys, then all you have to do is use the keys() function in a scalar context:

```
    $num_keys = keys %hash;
```

The keys() function also resets the iterator, which means that you may see strange results if you use this between uses of other hash operators such as each().

20.6.5 How do I sort a hash (optionally by value instead of key)?

Internally, hashes are stored in a way that prevents you from imposing an order on key-value pairs. Instead, you have to sort a list of the keys or values:

```
    @keys = sort keys %hash;    # sorted by key
    @keys = sort {
                    $hash{$a} cmp $hash{$b}
            } keys %hash;       # and by value
```

Here we'll do a reverse numeric sort by value, and if two keys are identical, sort by length of key, or if that fails, by straight ASCII comparison of the keys (well, possibly modified by your locale–see *perllocale*).

```
    @keys = sort {
                $hash{$b} <=> $hash{$a}
                          ||
                length($b) <=> length($a)
                          ||
                      $a cmp $b
    } keys %hash;
```

20.6.6 How can I always keep my hash sorted?

You can look into using the DB_File module and tie() using the $DB_BTREE hash bindings as documented in DB_File/"In Memory Databases". The Tie::IxHash module from CPAN might also be instructive.

20.6.7 What's the difference between "delete" and "undef" with hashes?

Hashes contain pairs of scalars: the first is the key, the second is the value. The key will be coerced to a string, although the value can be any kind of scalar: string, number, or reference. If a key $key is present in %hash, `exists($hash{$key})` will return true. The value for a given key can be `undef`, in which case `$hash{$key}` will be `undef` while `exists $hash{$key}` will return true. This corresponds to (`$key`, `undef`) being in the hash.

Pictures help... here's the %hash table:

```
  keys  values
+------+------+
|  a   |  3   |
|  x   |  7   |
|  d   |  0   |
|  e   |  2   |
+------+------+
```

And these conditions hold

```
$hash{'a'}                       is true
$hash{'d'}                       is false
defined $hash{'d'}               is true
defined $hash{'a'}               is true
exists $hash{'a'}                is true (Perl5 only)
grep ($_ eq 'a', keys %hash)     is true
```

If you now say

```
undef $hash{'a'}
```

your table now reads:

```
  keys  values
+------+------+
|  a   | undef|
|  x   |  7   |
|  d   |  0   |
|  e   |  2   |
+------+------+
```

and these conditions now hold; changes in caps:

```
$hash{'a'}                       is FALSE
$hash{'d'}                       is false
defined $hash{'d'}               is true
defined $hash{'a'}               is FALSE
exists $hash{'a'}                is true (Perl5 only)
grep ($_ eq 'a', keys %hash)     is true
```

Notice the last two: you have an undef value, but a defined key!

Now, consider this:

```
        delete $hash{'a'}
```

your table now reads:

```
          keys  values
        +------+------+
        |  x   |  7   |
        |  d   |  0   |
        |  e   |  2   |
        +------+------+
```

and these conditions now hold; changes in caps:

```
        $hash{'a'}                       is false
        $hash{'d'}                       is false
        defined $hash{'d'}               is true
        defined $hash{'a'}               is false
        exists $hash{'a'}                is FALSE (Perl5 only)
        grep ($_ eq 'a', keys %hash)     is FALSE
```

See, the whole entry is gone!

20.6.8 Why don't my tied hashes make the defined/exists distinction?

This depends on the tied hash's implementation of EXISTS(). For example, there isn't the concept of undef with hashes that are tied to DBM* files. It also means that exists() and defined() do the same thing with a DBM* file, and what they end up doing is not what they do with ordinary hashes.

20.6.9 How do I reset an each() operation part-way through?

Using `keys %hash` in scalar context returns the number of keys in the hash *and* resets the iterator associated with the hash. You may need to do this if you use `last` to exit a loop early so that when you re-enter it, the hash iterator has been reset.

20.6.10 How can I get the unique keys from two hashes?

First you extract the keys from the hashes into lists, then solve the "removing duplicates" problem described above. For example:

```
    %seen = ();
    for $element (keys(%foo), keys(%bar)) {
        $seen{$element}++;
    }
    @uniq = keys %seen;
```

Or more succinctly:

```
    @uniq = keys %{{%foo,%bar}};
```

Or if you really want to save space:

```
    %seen = ();
    while (defined ($key = each %foo)) {
        $seen{$key}++;
    }
    while (defined ($key = each %bar)) {
        $seen{$key}++;
    }
    @uniq = keys %seen;
```

20.6.11 How can I store a multidimensional array in a DBM file?

Either stringify the structure yourself (no fun), or else get the MLDBM (which uses Data::Dumper) module from CPAN and layer it on top of either DB_File or GDBM_File.

20.6.12 How can I make my hash remember the order I put elements into it?

Use the Tie::IxHash from CPAN.

```
use Tie::IxHash;
tie my %myhash, 'Tie::IxHash';
for (my $i=0; $i<20; $i++) {
    $myhash{$i} = 2*$i;
}
my @keys = keys %myhash;
# @keys = (0,1,2,3,...)
```

20.6.13 Why does passing a subroutine an undefined element in a hash create it?

If you say something like:

```
somefunc($hash{"nonesuch key here"});
```

Then that element "autovivifies"; that is, it springs into existence whether you store something there or not. That's because functions get scalars passed in by reference. If somefunc() modifies `$_[0]`, it has to be ready to write it back into the caller's version.

This has been fixed as of Perl5.004.

Normally, merely accessing a key's value for a nonexistent key does *not* cause that key to be forever there. This is different than awk's behavior.

20.6.14 How can I make the Perl equivalent of a C structure/C++ class/hash or array of hashes or arrays?

Usually a hash ref, perhaps like this:

```
$record = {
    NAME   => "Jason",
    EMPNO  => 132,
    TITLE  => "deputy peon",
    AGE    => 23,
    SALARY => 37_000,
    PALS   => [ "Norbert", "Rhys", "Phineas"],
};
```

References are documented in *perlref* and the upcoming *perlreftut*. Examples of complex data structures are given in *perldsc* and *perllol*. Examples of structures and object-oriented classes are in *perltoot*.

20.6.15 How can I use a reference as a hash key?

You can't do this directly, but you could use the standard Tie::RefHash module distributed with Perl.

20.7 Data: Misc

20.7.1 How do I handle binary data correctly?

Perl is binary clean, so this shouldn't be a problem. For example, this works fine (assuming the files are found):

```
if (‘cat /vmunix‘ =~ /gzip/) {
    print "Your kernel is GNU-zip enabled!\n";
}
```

On less elegant (read: Byzantine) systems, however, you have to play tedious games with "text" versus "binary" files. See binmode in *perlfunc* or *perlopentut*.

If you're concerned about 8-bit ASCII data, then see *perllocale*.

If you want to deal with multibyte characters, however, there are some gotchas. See the section on Regular Expressions.

20.7.2 How do I determine whether a scalar is a number/whole/integer/float?

Assuming that you don't care about IEEE notations like "NaN" or "Infinity", you probably just want to use a regular expression.

```
if (/\D/)            { print "has nondigits\n" }
if (/^\d+$/)         { print "is a whole number\n" }
if (/^-?\d+$/)       { print "is an integer\n" }
if (/^[+-]?\d+$/)    { print "is a +/- integer\n" }
if (/^-?\d+\.?\d*$/) { print "is a real number\n" }
if (/^-?(?:\d+(?:\.\d*)?|\.\d+)$/) { print "is a decimal number\n" }
if (/^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/)
                     { print "a C float\n" }
```

There are also some commonly used modules for the task. *Scalar::Util* (distributed with 5.8) provides access to perl's internal function `looks_like_number` for determining whether a variable looks like a number. *Data::Types* exports functions that validate data types using both the above and other regular expressions. Thirdly, there is `Regexp::Common` which has regular expressions to match various types of numbers. Those three modules are available from the CPAN.

If you're on a POSIX system, Perl supports the `POSIX::strtod` function. Its semantics are somewhat cumbersome, so here's a `getnum` wrapper function for more convenient access. This function takes a string and returns the number it found, or `undef` for input that isn't a C float. The `is_numeric` function is a front end to `getnum` if you just want to say, "Is this a float?"

```
sub getnum {
    use POSIX qw(strtod);
    my $str = shift;
    $str =~ s/^\s+//;
    $str =~ s/\s+$//;
    $! = 0;
    my($num, $unparsed) = strtod($str);
    if (($str eq ’’) || ($unparsed != 0) || $!) {
        return undef;
    } else {
        return $num;
    }
}

sub is_numeric { defined getnum($_[0]) }
```

Or you could check out the *String::Scanf* module on the CPAN instead. The POSIX module (part of the standard Perl distribution) provides the `strtod` and `strtol` for converting strings to double and longs, respectively.

20.7.3 How do I keep persistent data across program calls?

For some specific applications, you can use one of the DBM modules. See AnyDBM_File. More generically, you should consult the FreezeThaw or Storable modules from CPAN. Starting from Perl 5.8 Storable is part of the standard distribution. Here's one example using Storable's `store` and `retrieve` functions:

```
    use Storable;
    store(\%hash, "filename");

    # later on...
    $href = retrieve("filename");        # by ref
    %hash = %{ retrieve("filename") };   # direct to hash
```

20.7.4 How do I print out or copy a recursive data structure?

The Data::Dumper module on CPAN (or the 5.005 release of Perl) is great for printing out data structures. The Storable module on CPAN (or the 5.8 release of Perl), provides a function called `dclone` that recursively copies its argument.

```
    use Storable qw(dclone);
    $r2 = dclone($r1);
```

Where $r1 can be a reference to any kind of data structure you'd like. It will be deeply copied. Because `dclone` takes and returns references, you'd have to add extra punctuation if you had a hash of arrays that you wanted to copy.

```
    %newhash = %{ dclone(\%oldhash) };
```

20.7.5 How do I define methods for every class/object?

Use the UNIVERSAL class (see *UNIVERSAL*).

20.7.6 How do I verify a credit card checksum?

Get the Business::CreditCard module from CPAN.

20.7.7 How do I pack arrays of doubles or floats for XS code?

The kgbpack.c code in the PGPLOT module on CPAN does just this. If you're doing a lot of float or double processing, consider using the PDL module from CPAN instead–it makes number-crunching easy.

20.8 AUTHOR AND COPYRIGHT

Copyright (c) 1997-2002 Tom Christiansen and Nathan Torkington. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

Chapter 21

perlfaq5

Files and Formats ($Revision: 1.30 $, $Date: 2003/11/23 08:07:46 $)

21.1 DESCRIPTION

This section deals with I/O and the "f" issues: filehandles, flushing, formats, and footers.

21.1.1 How do I flush/unbuffer an output filehandle? Why must I do this?

Perl does not support truly unbuffered output (except insofar as you can `syswrite(OUT, $char, 1)`), although it does support is "command buffering", in which a physical write is performed after every output command.

The C standard I/O library (stdio) normally buffers characters sent to devices so that there isn't a system call for each byte. In most stdio implementations, the type of output buffering and the size of the buffer varies according to the type of device. Perl's print() and write() functions normally buffer output, while syswrite() bypasses buffering all together.

If you want your output to be sent immediately when you execute print() or write() (for instance, for some network protocols), you must set the handle's autoflush flag. This flag is the Perl variable $| and when it is set to a true value, Perl will flush the handle's buffer after each print() or write(). Setting $| affects buffering only for the currently selected default file handle. You choose this handle with the one argument select() call (see $| in *perlvar* and select in *perlfunc*).

Use select() to choose the desired handle, then set its per-filehandle variables.

```
    $old_fh = select(OUTPUT_HANDLE);
    $| = 1;
    select($old_fh);
```

Some idioms can handle this in a single statement:

```
    select((select(OUTPUT_HANDLE), $| = 1)[0]);

    $| = 1, select $_ for select OUTPUT_HANDLE;
```

Some modules offer object-oriented access to handles and their variables, although they may be overkill if this is the only thing you do with them. You can use IO::Handle:

```
    use IO::Handle;
    open(DEV, ">/dev/printer");   # but is this?
    DEV->autoflush(1);
```

or IO::Socket:

```
    use IO::Socket;               # this one is kinda a pipe?
        my $sock = IO::Socket::INET->new( 'www.example.com:80' ) ;

    $sock->autoflush();
```

21.1.2 How do I change one line in a file/delete a line in a file/insert a line in the middle of a file/append to the beginning of a file?

Use the Tie::File module, which is included in the standard distribution since Perl 5.8.0.

21.1.3 How do I count the number of lines in a file?

One fairly efficient way is to count newlines in the file. The following program uses a feature of tr///, as documented in *perlop*. If your text file doesn't end with a newline, then it's not really a proper text file, so this may report one fewer line than you expect.

```
    $lines = 0;
    open(FILE, $filename) or die "Can't open `$filename': $!";
    while (sysread FILE, $buffer, 4096) {
        $lines += ($buffer =~ tr/\n//);
    }
    close FILE;
```

This assumes no funny games with newline translations.

21.1.4 How can I use Perl's `-i` option from within a program?

`-i` sets the value of Perl's `$^I` variable, which in turn affects the behavior of <>; see *perlrun* for more details. By modifying the appropriate variables directly, you can get the same behavior within a larger program. For example:

```
     # ...
     {
        local($^I, @ARGV) = ('.orig', glob("*.c"));
        while (<>) {
           if ($. == 1) {
               print "This line should appear at the top of each file\n";
           }
           s/\b(p)earl\b/${1}erl/i;        # Correct typos, preserving case
           print;
           close ARGV if eof;              # Reset $.
        }
     }
     # $^I and @ARGV return to their old values here
```

This block modifies all the `.c` files in the current directory, leaving a backup of the original data from each file in a new `.c.orig` file.

21.1.5 How do I make a temporary file name?

Use the File::Temp module, see *File::Temp* for more information.

```
  use File::Temp qw/ tempfile tempdir /;

  $dir = tempdir( CLEANUP => 1 );
  ($fh, $filename) = tempfile( DIR => $dir );

  # or if you don't need to know the filename

  $fh = tempfile( DIR => $dir );
```

The File::Temp has been a standard module since Perl 5.6.1. If you don't have a modern enough Perl installed, use the `new_tmpfile` class method from the IO::File module to get a filehandle opened for reading and writing. Use it if you don't need to know the file's name:

```
    use IO::File;
    $fh = IO::File->new_tmpfile()
        or die "Unable to make new temporary file: $!";
```

If you're committed to creating a temporary file by hand, use the process ID and/or the current time-value. If you need to have many temporary files in one process, use a counter:

```
    BEGIN {
        use Fcntl;
        my $temp_dir = -d '/tmp' ? '/tmp' : $ENV{TMPDIR} || $ENV{TEMP};
        my $base_name = sprintf("%s/%d-%d-0000", $temp_dir, $$, time());
        sub temp_file {
            local *FH;
            my $count = 0;
            until (defined(fileno(FH)) || $count++ > 100) {
                $base_name =~ s/-(\d+)$/"-" . (1 + $1)/e;
                # O_EXCL is required for security reasons.
                sysopen(FH, $base_name, O_WRONLY|O_EXCL|O_CREAT);
            }
            if (defined(fileno(FH))
                return (*FH, $base_name);
            } else {
                return ();
            }
        }
    }
```

21.1.6 How can I manipulate fixed-record-length files?

The most efficient way is using pack() and unpack(). This is faster than using substr() when taking many, many strings. It is slower for just a few.

Here is a sample chunk of code to break up and put back together again some fixed-format input lines, in this case from the output of a normal, Berkeley-style ps:

```
    # sample input line:
    #   15158 p5  T      0:00 perl /home/tchrist/scripts/now-what
    my $PS_T = 'A6 A4 A7 A5 A*';
    open my $ps, '-|', 'ps';
    print scalar <$ps>;
    my @fields = qw( pid tt stat time command );
    while (<$ps>) {
        my %process;
        @process{@fields} = unpack($PS_T, $_);
        for my $field ( @fields ) {
            print "$field: <$process{$field}>\n";
        }
        print 'line=', pack($PS_T, @process{@fields} ), "\n";
    }
```

We've used a hash slice in order to easily handle the fields of each row. Storing the keys in an array means it's easy to operate on them as a group or loop over them with for. It also avoids polluting the program with global variables and using symbolic references.

21.1.7 How can I make a filehandle local to a subroutine? How do I pass filehandles between subroutines? How do I make an array of filehandles?

As of perl5.6, open() autovivifies file and directory handles as references if you pass it an uninitialized scalar variable. You can then pass these references just like any other scalar, and use them in the place of named handles.

```
        open my    $fh, $file_name;

        open local $fh, $file_name;

        print $fh "Hello World!\n";

        process_file( $fh );
```

Before perl5.6, you had to deal with various typeglob idioms which you may see in older code.

```
        open FILE, "> $filename";
        process_typeglob(   *FILE );
        process_reference( \*FILE );

        sub process_typeglob  { local *FH = shift; print FH  "Typeglob!" }
        sub process_reference { local $fh = shift; print $fh "Reference!" }
```

If you want to create many anonymous handles, you should check out the Symbol or IO::Handle modules.

21.1.8 How can I use a filehandle indirectly?

An indirect filehandle is using something other than a symbol in a place that a filehandle is expected. Here are ways to get indirect filehandles:

```
    $fh =   SOME_FH;       # bareword is strict-subs hostile
    $fh =  "SOME_FH";      # strict-refs hostile; same package only
    $fh =  *SOME_FH;       # typeglob
    $fh = \*SOME_FH;       # ref to typeglob (bless-able)
    $fh =  *SOME_FH{IO};   # blessed IO::Handle from *SOME_FH typeglob
```

Or, you can use the `new` method from one of the IO::* modules to create an anonymous filehandle, store that in a scalar variable, and use it as though it were a normal filehandle.

```
    use IO::Handle;                     # 5.004 or higher
    $fh = IO::Handle->new();
```

Then use any of those as you would a normal filehandle. Anywhere that Perl is expecting a filehandle, an indirect filehandle may be used instead. An indirect filehandle is just a scalar variable that contains a filehandle. Functions like `print`, `open`, `seek`, or the <FH> diamond operator will accept either a named filehandle or a scalar variable containing one:

```
    ($ifh, $ofh, $efh) = (*STDIN, *STDOUT, *STDERR);
    print $ofh "Type it: ";
    $got = <$ifh>
    print $efh "What was that: $got";
```

If you're passing a filehandle to a function, you can write the function in two ways:

```
    sub accept_fh {
        my $fh = shift;
        print $fh "Sending to indirect filehandle\n";
    }
```

Or it can localize a typeglob and use the filehandle directly:

```
    sub accept_fh {
        local *FH = shift;
        print  FH "Sending to localized filehandle\n";
    }
```

Both styles work with either objects or typeglobs of real filehandles. (They might also work with strings under some circumstances, but this is risky.)

```
    accept_fh(*STDOUT);
    accept_fh($handle);
```

In the examples above, we assigned the filehandle to a scalar variable before using it. That is because only simple scalar variables, not expressions or subscripts of hashes or arrays, can be used with built-ins like `print`, `printf`, or the diamond operator. Using something other than a simple scalar variable as a filehandle is illegal and won't even compile:

```
    @fd = (*STDIN, *STDOUT, *STDERR);
    print $fd[1] "Type it: ";                          # WRONG
    $got = <$fd[0]>                                    # WRONG
    print $fd[2] "What was that: $got";                # WRONG
```

With `print` and `printf`, you get around this by using a block and an expression where you would place the filehandle:

```
    print  { $fd[1] } "funny stuff\n";
    printf { $fd[1] } "Pity the poor %x.\n", 3_735_928_559;
    # Pity the poor deadbeef.
```

That block is a proper block like any other, so you can put more complicated code there. This sends the message out to one of two places:

```
    $ok = -x "/bin/cat";
    print { $ok ? $fd[1] : $fd[2] } "cat stat $ok\n";
    print { $fd[ 1+ ($ok || 0) ]  } "cat stat $ok\n";
```

This approach of treating `print` and `printf` like object methods calls doesn't work for the diamond operator. That's because it's a real operator, not just a function with a comma-less argument. Assuming you've been storing typeglobs in your structure as we did above, you can use the built-in function named `readline` to read a record just as <> does. Given the initialization shown above for @fd, this would work, but only because readline() requires a typeglob. It doesn't work with objects or strings, which might be a bug we haven't fixed yet.

```
    $got = readline($fd[0]);
```

Let it be noted that the flakiness of indirect filehandles is not related to whether they're strings, typeglobs, objects, or anything else. It's the syntax of the fundamental operators. Playing the object game doesn't help you at all here.

21.1.9 How can I set up a footer format to be used with write()?

There's no builtin way to do this, but *perlform* has a couple of techniques to make it possible for the intrepid hacker.

21.1.10 How can I write() into a string?

See Accessing Formatting Internals in *perlform* for an swrite() function.

21.1.11 How can I output my numbers with commas added?

This subroutine will add commas to your number:

```
        sub commify {
           local $_  = shift;
           1 while s/^([-+]?\d+)(\d{3})/$1,$2/;
           return $_;
           }
```

This regex from Benjamin Goldberg will add commas to numbers:

```
   s/(^[-+]?\d+?(?=(?>(?:\d{3})+)(?!\d))|\G\d{3}(?=\d))/$1,/g;
```

It is easier to see with comments:

```
   s/(
       ^[-+]?            # beginning of number.
       \d{1,3}?          # first digits before first comma
       (?=               # followed by, (but not included in the match) :
          (?>(?:\d{3})+) # some positive multiple of three digits.
          (?!\d)         # an *exact* multiple, not x * 3 + 1 or whatever.
       )
      |                  # or:
       \G\d{3}           # after the last group, get three digits
       (?=\d)            # but they have to have more digits after them.
   )/$1,/xg;
```

21.1.12 How can I translate tildes (˜) in a filename?

Use the <> (glob()) operator, documented in *perlfunc*. Older versions of Perl require that you have a shell installed that groks tildes. Recent perl versions have this feature built in. The File::KGlob module (available from CPAN) gives more portable glob functionality.

Within Perl, you may use this directly:

```
        $filename =~ s{
          ^ ~             # find a leading tilde
          (               # save this in $1
              [^/]        # a non-slash character
                    *     # repeated 0 or more times (0 means me)
          )
        }{
          $1
              ? (getpwnam($1))[7]
              : ( $ENV{HOME} || $ENV{LOGDIR} )
        }ex;
```

21.1.13 How come when I open a file read-write it wipes it out?

Because you're using something like this, which truncates the file and *then* gives you read-write access:

```
open(FH, "+> /path/name");          # WRONG (almost always)
```

Whoops. You should instead use this, which will fail if the file doesn't exist.

```
open(FH, "+< /path/name");          # open for update
```

Using ">" always clobbers or creates. Using "<" never does either. The "+" doesn't change this.
Here are examples of many kinds of file opens. Those using sysopen() all assume

```
use Fcntl;
```

To open file for reading:

```
open(FH, "< $path")                                 || die $!;
sysopen(FH, $path, O_RDONLY)                        || die $!;
```

To open file for writing, create new file if needed or else truncate old file:

```
open(FH, "> $path") || die $!;
sysopen(FH, $path, O_WRONLY|O_TRUNC|O_CREAT)        || die $!;
sysopen(FH, $path, O_WRONLY|O_TRUNC|O_CREAT, 0666)  || die $!;
```

To open file for writing, create new file, file must not exist:

```
sysopen(FH, $path, O_WRONLY|O_EXCL|O_CREAT)         || die $!;
sysopen(FH, $path, O_WRONLY|O_EXCL|O_CREAT, 0666)   || die $!;
```

To open file for appending, create if necessary:

```
open(FH, ">> $path") || die $!;
sysopen(FH, $path, O_WRONLY|O_APPEND|O_CREAT)       || die $!;
sysopen(FH, $path, O_WRONLY|O_APPEND|O_CREAT, 0666) || die $!;
```

To open file for appending, file must exist:

```
sysopen(FH, $path, O_WRONLY|O_APPEND)               || die $!;
```

To open file for update, file must exist:

```
open(FH, "+< $path")                                || die $!;
sysopen(FH, $path, O_RDWR)                          || die $!;
```

To open file for update, create file if necessary:

```
sysopen(FH, $path, O_RDWR|O_CREAT)                  || die $!;
sysopen(FH, $path, O_RDWR|O_CREAT, 0666)            || die $!;
```

To open file for update, file must not exist:

```
sysopen(FH, $path, O_RDWR|O_EXCL|O_CREAT)           || die $!;
sysopen(FH, $path, O_RDWR|O_EXCL|O_CREAT, 0666)     || die $!;
```

To open a file without blocking, creating if necessary:

```
sysopen(FH, "/foo/somefile", O_WRONLY|O_NDELAY|O_CREAT)
        or die "can't open /foo/somefile: $!":
```

Be warned that neither creation nor deletion of files is guaranteed to be an atomic operation over NFS. That is, two processes might both successfully create or unlink the same file! Therefore O_EXCL isn't as exclusive as you might wish.
See also the new *perlopentut* if you have it (new for 5.6).

21.1.14 Why do I sometimes get an "Argument list too long" when I use <*>?

The <> operator performs a globbing operation (see above). In Perl versions earlier than v5.6.0, the internal glob() operator forks csh(1) to do the actual glob expansion, but csh can't handle more than 127 items and so gives the error message `Argument list too long`. People who installed tcsh as csh won't have this problem, but their users may be surprised by it.

To get around this, either upgrade to Perl v5.6.0 or later, do the glob yourself with readdir() and patterns, or use a module like File::KGlob, one that doesn't use the shell to do globbing.

21.1.15 Is there a leak/bug in glob()?

Due to the current implementation on some operating systems, when you use the glob() function or its angle-bracket alias in a scalar context, you may cause a memory leak and/or unpredictable behavior. It's best therefore to use glob() only in list context.

21.1.16 How can I open a file with a leading ">" or trailing blanks?

Normally perl ignores trailing blanks in filenames, and interprets certain leading characters (or a trailing "|") to mean something special.

The three argument form of open() lets you specify the mode separately from the filename. The open() function treats special mode characters and whitespace in the filename as literals

```
        open FILE, "<", "  file  ";  # filename is "   file   "
        open FILE, ">", ">file";     # filename is ">file"
```

It may be a lot clearer to use sysopen(), though:

```
    use Fcntl;
    $badpath = "<<<something really wicked   ";
    sysopen (FH, $badpath, O_WRONLY | O_CREAT | O_TRUNC)
        or die "can't open $badpath: $!";
```

21.1.17 How can I reliably rename a file?

If your operating system supports a proper mv(1) utility or its functional equivalent, this works:

```
    rename($old, $new) or system("mv", $old, $new);
```

It may be more portable to use the File::Copy module instead. You just copy to the new file to the new name (checking return values), then delete the old one. This isn't really the same semantically as a rename(), which preserves meta-information like permissions, timestamps, inode info, etc.

Newer versions of File::Copy export a move() function.

21.1.18 How can I lock a file?

Perl's builtin flock() function (see *perlfunc* for details) will call flock(2) if that exists, fcntl(2) if it doesn't (on perl version 5.004 and later), and lockf(3) if neither of the two previous system calls exists. On some systems, it may even use a different form of native locking. Here are some gotchas with Perl's flock():

1. Produces a fatal error if none of the three system calls (or their close equivalent) exists.
2. lockf(3) does not provide shared locking, and requires that the filehandle be open for writing (or appending, or read/writing).

3. Some versions of flock() can't lock files over a network (e.g. on NFS file systems), so you'd need to force the use of fcntl(2) when you build Perl. But even this is dubious at best. See the flock entry of *perlfunc* and the *INSTALL* file in the source distribution for information on building Perl to do this.

   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 files locked with flock() may be modified by programs that do not also use flock(). Cars that stop for red lights get on well with each other, but not with cars that don't stop for red lights. See the perlport manpage, your port's specific documentation, or your system-specific local manpages for details. It's best to assume traditional behavior if you're writing portable programs. (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.)

   For more information on file locking, see also File Locking in *perlopentut* if you have it (new for 5.6).

21.1.19 Why can't I just open(FH, ">file.lock")?

A common bit of code **NOT TO USE** is this:

```
    sleep(3) while -e "file.lock";      # PLEASE DO NOT USE
    open(LCK, "> file.lock");           # THIS BROKEN CODE
```

This is a classic race condition: you take two steps to do something which must be done in one. That's why computer hardware provides an atomic test-and-set instruction. In theory, this "ought" to work:

```
    sysopen(FH, "file.lock", O_WRONLY|O_EXCL|O_CREAT)
                or die "can't open  file.lock: $!";
```

except that lamentably, file creation (and deletion) is not atomic over NFS, so this won't work (at least, not every time) over the net. Various schemes involving link() have been suggested, but these tend to involve busy-wait, which is also subdesirable.

21.1.20 I still don't get locking. I just want to increment the number in the file. How can I do this?

Didn't anyone ever tell you web-page hit counters were useless? They don't count number of hits, they're a waste of time, and they serve only to stroke the writer's vanity. It's better to pick a random number; they're more realistic.

Anyway, this is what you can do if you can't help yourself.

```
    use Fcntl qw(:DEFAULT :flock);
    sysopen(FH, "numfile", O_RDWR|O_CREAT)       or die "can't open numfile: $!";
    flock(FH, LOCK_EX)                           or die "can't flock numfile: $!";
    $num = <FH> || 0;
    seek(FH, 0, 0)                               or die "can't rewind numfile: $!";
    truncate(FH, 0)                              or die "can't truncate numfile: $!";
    (print FH $num+1, "\n")                      or die "can't write numfile: $!";
    close FH                                     or die "can't close numfile: $!";
```

Here's a much better web-page hit counter:

```
    $hits = int( (time() - 850_000_000) / rand(1_000) );
```

If the count doesn't impress your friends, then the code might. :-)

21.1.21 All I want to do is append a small amount of text to the end of a file. Do I still have to use locking?

If you are on a system that correctly implements flock() and you use the example appending code from "perldoc -f flock" everything will be OK even if the OS you are on doesn't implement append mode correctly (if such a system exists.) So if you are happy to restrict yourself to OSs that implement flock() (and that's not really much of a restriction) then that is what you should do.

If you know you are only going to use a system that does correctly implement appending (i.e. not Win32) then you can omit the seek() from the above code.

If you know you are only writing code to run on an OS and filesystem that does implement append mode correctly (a local filesystem on a modern Unix for example), and you keep the file in block-buffered mode and you write less than one buffer-full of output between each manual flushing of the buffer then each bufferload is almost guaranteed to be written to the end of the file in one chunk without getting intermingled with anyone else's output. You can also use the syswrite() function which is simply a wrapper around your systems write(2) system call.

There is still a small theoretical chance that a signal will interrupt the system level write() operation before completion. There is also a possibility that some STDIO implementations may call multiple system level write()s even if the buffer was empty to start. There may be some systems where this probability is reduced to zero.

21.1.22 How do I randomly update a binary file?

If you're just trying to patch a binary, in many cases something as simple as this works:

```
    perl -i -pe 's{window manager}{window mangler}g' /usr/bin/emacs
```

However, if you have fixed sized records, then you might do something more like this:

```
    $RECSIZE = 220; # size of record, in bytes
    $recno   = 37;  # which record to update
    open(FH, "+<somewhere") || die "can't update somewhere: $!";
    seek(FH, $recno * $RECSIZE, 0);
    read(FH, $record, $RECSIZE) == $RECSIZE || die "can't read record $recno: $!";
    # munge the record
    seek(FH, -$RECSIZE, 1);
    print FH $record;
    close FH;
```

Locking and error checking are left as an exercise for the reader. Don't forget them or you'll be quite sorry.

21.1.23 How do I get a file's timestamp in perl?

If you want to retrieve the time at which the file was last read, written, or had its meta-data (owner, etc) changed, you use the **-M**, **-A**, or **-C** file test operations as documented in *perlfunc*. These retrieve the age of the file (measured against the start-time of your program) in days as a floating point number. Some platforms may not have all of these times. See *perlport* for details. To retrieve the "raw" time in seconds since the epoch, you would call the stat function, then use localtime(), gmtime(), or POSIX::strftime() to convert this into human-readable form.

Here's an example:

```
    $write_secs = (stat($file))[9];
    printf "file %s updated at %s\n", $file,
        scalar localtime($write_secs);
```

If you prefer something more legible, use the File::stat module (part of the standard distribution in version 5.004 and later):

```
    # error checking left as an exercise for reader.
    use File::stat;
    use Time::localtime;
    $date_string = ctime(stat($file)->mtime);
    print "file $file updated at $date_string\n";
```

The POSIX::strftime() approach has the benefit of being, in theory, independent of the current locale. See *perllocale* for details.

21.1.24 How do I set a file's timestamp in perl?

You use the utime() function documented in utime in *perlfunc*. By way of example, here's a little program that copies the read and write times from its first argument to all the rest of them.

```
    if (@ARGV < 2) {
        die "usage: cptimes timestamp_file other_files ...\n";
    }
    $timestamp = shift;
    ($atime, $mtime) = (stat($timestamp))[8,9];
    utime $atime, $mtime, @ARGV;
```

Error checking is, as usual, left as an exercise for the reader.

Note that utime() currently doesn't work correctly with Win95/NT ports. A bug has been reported. Check it carefully before using utime() on those platforms.

21.1.25 How do I print to more than one file at once?

To connect one filehandle to several output filehandles, you can use the IO::Tee or Tie::FileHandle::Multiplex modules.

If you only have to do this once, you can print individually to each filehandle.

```
    for $fh (FH1, FH2, FH3) { print $fh "whatever\n" }
```

21.1.26 How can I read in an entire file all at once?

You can use the File::Slurp module to do it in one step.

```
        use File::Slurp;

        $all_of_it = read_file($filename); # entire file in scalar
    @all_lines = read_file($filename); # one line perl element
```

The customary Perl approach for processing all the lines in a file is to do so one line at a time:

```
    open (INPUT, $file)         || die "can't open $file: $!";
    while (<INPUT>) {
        chomp;
        # do something with $_
    }
    close(INPUT)                || die "can't close $file: $!";
```

This is tremendously more efficient than reading the entire file into memory as an array of lines and then processing it one element at a time, which is often–if not almost always–the wrong approach. Whenever you see someone do this:

```
    @lines = <INPUT>;
```

you should think long and hard about why you need everything loaded at once. It's just not a scalable solution. You might also find it more fun to use the standard Tie::File module, or the DB_File module's $DB_RECNO bindings, which allow you to tie an array to a file so that accessing an element the array actually accesses the corresponding line in the file.

You can read the entire filehandle contents into a scalar.

```
    {
        local(*INPUT, $/);
        open (INPUT, $file)     || die "can't open $file: $!";
        $var = <INPUT>;
    }
```

That temporarily undefs your record separator, and will automatically close the file at block exit. If the file is already open, just use this:

```
    $var = do { local $/; <INPUT> };
```

For ordinary files you can also use the read function.

```
        read( INPUT, $var, -s INPUT );
```

The third argument tests the byte size of the data on the INPUT filehandle and reads that many bytes into the buffer $var.

21.1.27 How can I read in a file by paragraphs?

Use the `$/` variable (see *perlvar* for details). You can either set it to `""` to eliminate empty paragraphs (`"abc\n\n\n\ndef"`, for instance, gets treated as two paragraphs and not three), or `"\n\n"` to accept empty paragraphs.

Note that a blank line must have no blanks in it. Thus `"fred\n \nstuff\n\n"` is one paragraph, but `"fred\n\nstuff\n\n"` is two.

21.1.28 How can I read a single character from a file? From the keyboard?

You can use the builtin `getc()` function for most filehandles, but it won't (easily) work on a terminal device. For STDIN, either use the Term::ReadKey module from CPAN or use the sample code in getc in *perlfunc*.

If your system supports the portable operating system programming interface (POSIX), you can use the following code, which you'll note turns off echo processing as well.

```
    #!/usr/bin/perl -w
    use strict;
    $| = 1;
    for (1..4) {
        my $got;
        print "gimme: ";
        $got = getone();
        print "--> $got\n";
    }
    exit;

    BEGIN {
        use POSIX qw(:termios_h);

        my ($term, $oterm, $echo, $noecho, $fd_stdin);
        $fd_stdin = fileno(STDIN);

        $term     = POSIX::Termios->new();
        $term->getattr($fd_stdin);
        $oterm     = $term->getlflag();

        $echo     = ECHO | ECHOK | ICANON;
        $noecho   = $oterm & ~$echo;

        sub cbreak {
            $term->setlflag($noecho);
            $term->setcc(VTIME, 1);
            $term->setattr($fd_stdin, TCSANOW);
        }

        sub cooked {
            $term->setlflag($oterm);
            $term->setcc(VTIME, 0);
            $term->setattr($fd_stdin, TCSANOW);
        }

        sub getone {
            my $key = '';
            cbreak();
            sysread(STDIN, $key, 1);
            cooked();
            return $key;
        }

    }

    END { cooked() }
```

The Term::ReadKey module from CPAN may be easier to use. Recent versions include also support for non-portable systems as well.

```
    use Term::ReadKey;
    open(TTY, "</dev/tty");
    print "Gimme a char: ";
    ReadMode "raw";
    $key = ReadKey 0, *TTY;
    ReadMode "normal";
    printf "\nYou said %s, char number %03d\n",
        $key, ord $key;
```

21.1.29 How can I tell whether there's a character waiting on a filehandle?

The very first thing you should do is look into getting the Term::ReadKey extension from CPAN. As we mentioned earlier, it now even has limited support for non-portable (read: not open systems, closed, proprietary, not POSIX, not Unix, etc) systems.

You should also check out the Frequently Asked Questions list in comp.unix.* for things like this: the answer is essentially the same. It's very system dependent. Here's one solution that works on BSD systems:

```
    sub key_ready {
        my($rin, $nfd);
        vec($rin, fileno(STDIN), 1) = 1;
        return $nfd = select($rin,undef,undef,0);
    }
```

If you want to find out how many characters are waiting, there's also the FIONREAD ioctl call to be looked at. The *h2ph* tool that comes with Perl tries to convert C include files to Perl code, which can be `require`d. FIONREAD ends up defined as a function in the *sys/ioctl.ph* file:

```
    require 'sys/ioctl.ph';

    $size = pack("L", 0);
    ioctl(FH, FIONREAD(), $size)    or die "Couldn't call ioctl: $!\n";
    $size = unpack("L", $size);
```

If *h2ph* wasn't installed or doesn't work for you, you can *grep* the include files by hand:

```
    % grep FIONREAD /usr/include/*/*
    /usr/include/asm/ioctls.h:#define FIONREAD      0x541B
```

Or write a small C program using the editor of champions:

```
    % cat > fionread.c
    #include <sys/ioctl.h>
    main() {
        printf("%#08x\n", FIONREAD);
    }
    ^D
    % cc -o fionread fionread.c
    % ./fionread
    0x4004667f
```

And then hard code it, leaving porting as an exercise to your successor.

```
    $FIONREAD = 0x4004667f;         # XXX: opsys dependent

    $size = pack("L", 0);
    ioctl(FH, $FIONREAD, $size)     or die "Couldn't call ioctl: $!\n";
    $size = unpack("L", $size);
```

FIONREAD requires a filehandle connected to a stream, meaning that sockets, pipes, and tty devices work, but *not* files.

21.1.30 How do I do a `tail -f` in perl?

First try

```
    seek(GWFILE, 0, 1);
```

The statement `seek(GWFILE, 0, 1)` doesn't change the current position, but it does clear the end-of-file condition on the handle, so that the next <GWFILE> makes Perl try again to read something.

If that doesn't work (it relies on features of your stdio implementation), then you need something more like this:

```
        for (;;) {
          for ($curpos = tell(GWFILE); <GWFILE>; $curpos = tell(GWFILE)) {
            # search for some stuff and put it into files
          }
          # sleep for a while
          seek(GWFILE, $curpos, 0);  # seek to where we had been
        }
```

If this still doesn't work, look into the POSIX module. POSIX defines the clearerr() method, which can remove the end of file condition on a filehandle. The method: read until end of file, clearerr(), read some more. Lather, rinse, repeat.

There's also a File::Tail module from CPAN.

21.1.31 How do I dup() a filehandle in Perl?

If you check open in *perlfunc*, you'll see that several of the ways to call open() should do the trick. For example:

```
    open(LOG, ">>/foo/logfile");
    open(STDERR, ">&LOG");
```

Or even with a literal numeric descriptor:

```
   $fd = $ENV{MHCONTEXTFD};
   open(MHCONTEXT, "<&=$fd");   # like fdopen(3S)
```

Note that "<&STDIN" makes a copy, but "<&=STDIN" make an alias. That means if you close an aliased handle, all aliases become inaccessible. This is not true with a copied one.

Error checking, as always, has been left as an exercise for the reader.

21.1.32 How do I close a file descriptor by number?

This should rarely be necessary, as the Perl close() function is to be used for things that Perl opened itself, even if it was a dup of a numeric descriptor as with MHCONTEXT above. But if you really have to, you may be able to do this:

```
    require 'sys/syscall.ph';
    $rc = syscall(&SYS_close, $fd + 0);  # must force numeric
    die "can't sysclose $fd: $!" unless $rc == -1;
```

Or, just use the fdopen(3S) feature of open():

```
    {
        local *F;
        open F, "<&=$fd" or die "Cannot reopen fd=$fd: $!";
        close F;
    }
```

21.1.33 Why can't I use "C:\temp\foo" in DOS paths? Why doesn't ‘C:\temp\foo.exe‘ work?

Whoops! You just put a tab and a formfeed into that filename! Remember that within double quoted strings ("like\this"), the backslash is an escape character. The full list of these is in Quote and Quote-like Operators in *perlop*. Unsurprisingly, you don't have a file called "c:(tab)emp(formfeed)oo" or "c:(tab)emp(formfeed)oo.exe" on your legacy DOS filesystem.

Either single-quote your strings, or (preferably) use forward slashes. Since all DOS and Windows versions since something like MS-DOS 2.0 or so have treated / and \ the same in a path, you might as well use the one that doesn't clash with Perl–or the POSIX shell, ANSI C and C++, awk, Tcl, Java, or Python, just to mention a few. POSIX paths are more portable, too.

21.1.34 Why doesn't glob("*.*") get all the files?

Because even on non-Unix ports, Perl's glob function follows standard Unix globbing semantics. You'll need `glob("*")` to get all (non-hidden) files. This makes glob() portable even to legacy systems. Your port may include proprietary globbing functions as well. Check its documentation for details.

21.1.35 Why does Perl let me delete read-only files? Why does `-i` clobber protected files? Isn't this a bug in Perl?

This is elaborately and painstakingly described in the *file-dir-perms* article in the "Far More Than You Ever Wanted To Know" collection in http://www.cpan.org/misc/olddoc/FMTEYEWTK.tgz .

The executive summary: learn how your filesystem works. The permissions on a file say what can happen to the data in that file. The permissions on a directory say what can happen to the list of files in that directory. If you delete a file, you're removing its name from the directory (so the operation depends on the permissions of the directory, not of the file). If you try to write to the file, the permissions of the file govern whether you're allowed to.

21.1.36 How do I select a random line from a file?

Here's an algorithm from the Camel Book:

```
    srand;
    rand($.) < 1 && ($line = $_) while <>;
```

This has a significant advantage in space over reading the whole file in. You can find a proof of this method in *The Art of Computer Programming*, Volume 2, Section 3.4.2, by Donald E. Knuth.

You can use the File::Random module which provides a function for that algorithm:

```
        use File::Random qw/random_line/;
        my $line = random_line($filename);
```

Another way is to use the Tie::File module, which treats the entire file as an array. Simply access a random array element.

21.1.37 Why do I get weird spaces when I print an array of lines?

Saying

```
    print "@lines\n";
```

joins together the elements of `@lines` with a space between them. If `@lines` were `("little", "fluffy", "clouds")` then the above statement would print

```
    little fluffy clouds
```

but if each element of `@lines` was a line of text, ending a newline character `("little\n", "fluffy\n", "clouds\n")` then it would print:

```
    little
     fluffy
     clouds
```

If your array contains lines, just print them:

```
    print @lines;
```

21.2 AUTHOR AND COPYRIGHT

Copyright (c) 1997-2002 Tom Christiansen and Nathan Torkington. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples here are in the public domain. You are permitted and encouraged to use this code and any derivatives thereof in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit to the FAQ would be courteous but is not required.

Chapter 22

perlfaq6

Regular Expressions ($Revision: 1.20 $, $Date: 2003/01/03 20:05:28 $)

22.1 DESCRIPTION

This section is surprisingly small because the rest of the FAQ is littered with answers involving regular expressions. For example, decoding a URL and checking whether something is a number are handled with regular expressions, but those answers are found elsewhere in this document (in *perlfaq9*: “How do I decode or create those %-encodings on the web” and *perlfaq4*: “How do I determine whether a scalar is a number/whole/integer/float”, to be precise).

22.1.1 How can I hope to use regular expressions without creating illegible and unmaintainable code?

Three techniques can make regular expressions maintainable and understandable.

Comments Outside the Regex

Describe what you’re doing and how you’re doing it, using normal Perl comments.

```
# turn the line into the first word, a colon, and the
# number of characters on the rest of the line
s/^(\w+)(.*)/ lc($1) . ":" . length($2) /meg;
```

Comments Inside the Regex

The `/x` modifier causes whitespace to be ignored in a regex pattern (except in a character class), and also allows you to use normal comments there, too. As you can imagine, whitespace and comments help a lot.

`/x` lets you turn this:

```
s{<(?:[^>'"]*|".*?"|'.*?')+>}{}gs;
```

into this:

```
s{ <                    # opening angle bracket
    (?:                 # Non-backreffing grouping paren
         [^>'"] *       # 0 or more things that are neither > nor ' nor "
            |           #    or else
         ".*?"          # a section between double quotes (stingy match)
            |           #    or else
         '.*?'          # a section between single quotes (stingy match)
    ) +                 #   all occurring one or more times
   >                    # closing angle bracket
}{}gsx;                 # replace with nothing, i.e. delete
```

It’s still not quite so clear as prose, but it is very useful for describing the meaning of each part of the pattern.

Different Delimiters

While we normally think of patterns as being delimited with `/` characters, they can be delimited by almost any character. *perlre* describes this. For example, the `s///` above uses braces as delimiters. Selecting another delimiter can avoid quoting the delimiter within the pattern:

```
s/\/usr\/local/\/usr\/share/g;      # bad delimiter choice
s#/usr/local#/usr/share#g;          # better
```

22.1.2 I’m having trouble matching over more than one line. What’s wrong?

Either you don’t have more than one line in the string you’re looking at (probably), or else you aren’t using the correct modifier(s) on your pattern (possibly).

There are many ways to get multiline data into a string. If you want it to happen automatically while reading input, you’ll want to set $/ (probably to ” for paragraphs or `undef` for the whole file) to allow you to read more than one line at a time.

Read *perlre* to help you decide which of `/s` and `/m` (or both) you might want to use: `/s` allows dot to include newline, and `/m` allows caret and dollar to match next to a newline, not just at the end of the string. You do need to make sure that you’ve actually got a multiline string in there.

For example, this program detects duplicate words, even when they span line breaks (but not paragraph ones). For this example, we don’t need `/s` because we aren’t using dot in a regular expression that we want to cross line boundaries. Neither do we need `/m` because we aren’t wanting caret or dollar to match at any point inside the record next to newlines. But it’s imperative that $/ be set to something other than the default, or else we won’t actually ever have a multiline record read in.

```
$/ = '';            # read in more whole paragraph, not just one line
while ( <> ) {
    while ( /\b([\w'-]+)(\s+\1)+\b/gi ) {   # word starts alpha
        print "Duplicate $1 at paragraph $.\n";
    }
}
```

Here’s code that finds sentences that begin with "From " (which would be mangled by many mailers):

```
$/ = '';            # read in more whole paragraph, not just one line
while ( <> ) {
    while ( /^From /gm ) { # /m makes ^ match next to \n
        print "leading from in paragraph $.\n";
    }
}
```

Here’s code that finds everything between START and END in a paragraph:

```
undef $/;           # read in whole file, not just one line or paragraph
while ( <> ) {
    while ( /START(.*?)END/sgm ) { # /s makes . cross line boundaries
        print "$1\n";
    }
}
```

22.1.3 How can I pull out lines between two patterns that are themselves on different lines?

You can use Perl's somewhat exotic `..` operator (documented in *perlop*):

```
    perl -ne 'print if /START/ .. /END/' file1 file2 ...
```

If you wanted text and not lines, you would use

```
    perl -0777 -ne 'print "$1\n" while /START(.*?)END/gs' file1 file2 ...
```

But if you want nested occurrences of `START` through `END`, you'll run up against the problem described in the question in this section on matching balanced text.

Here's another example of using `..`:

```
    while (<>) {
        $in_header =   1  .. /^$/;
        $in_body   = /^$/ .. eof();
        # now choose between them
    } continue {
        reset if eof();         # fix $.
    }
```

22.1.4 I put a regular expression into $ / but it didn't work. What's wrong?

Up to Perl 5.8.0, $/ has to be a string. This may change in 5.10, but don't get your hopes up. Until then, you can use these examples if you really need to do this.

Use the four argument form of sysread to continually add to a buffer. After you add to the buffer, you check if you have a complete line (using your regular expression).

```
       local $_ = "";
       while( sysread FH, $_, 8192, length ) {
          while( s/^((?s).*?)your_pattern/ ) {
             my $record = $1;
             # do stuff here.
          }
       }

 You can do the same thing with foreach and a match using the
 c flag and the \G anchor, if you do not mind your entire file
 being in memory at the end.

       local $_ = "";
       while( sysread FH, $_, 8192, length ) {
          foreach my $record ( m/\G((?s).*?)your_pattern/gc ) {
             # do stuff here.
          }
          substr( $_, 0, pos ) = "" if pos;
       }
```

22.1.5 How do I substitute case insensitively on the LHS while preserving case on the RHS?

Here’s a lovely Perlish solution by Larry Rosler. It exploits properties of bitwise xor on ASCII strings.

```
    $_= "this is a TEsT case";

    $old = 'test';
    $new = 'success';

    s{(\Q$old\E)}
     { uc $new | (uc $1 ^ $1) .
        (uc(substr $1, -1) ^ substr $1, -1) x
            (length($new) - length $1)
     }egi;

    print;
```

And here it is as a subroutine, modeled after the above:

```
    sub preserve_case($$) {
        my ($old, $new) = @_;
        my $mask = uc $old ^ $old;

        uc $new | $mask .
            substr($mask, -1) x (length($new) - length($old))
    }

    $a = "this is a TEsT case";
    $a =~ s/(test)/preserve_case($1, "success")/egi;
    print "$a\n";
```

This prints:

```
    this is a SUcCESS case
```

As an alternative, to keep the case of the replacement word if it is longer than the original, you can use this code, by Jeff Pinyan:

```
  sub preserve_case {
    my ($from, $to) = @_;
    my ($lf, $lt) = map length, @_;

    if ($lt < $lf) { $from = substr $from, 0, $lt }
    else { $from .= substr $to, $lf }

    return uc $to | ($from ^ uc $from);
  }
```

This changes the sentence to "this is a SUcCess case."

Just to show that C programmers can write C in any programming language, if you prefer a more C-like solution, the following script makes the substitution have the same case, letter by letter, as the original. (It also happens to run about 240% slower than the Perlish solution runs.) If the substitution has more characters than the string being substituted, the case of the last character is used for the rest of the substitution.

```
    # Original by Nathan Torkington, massaged by Jeffrey Friedl
    #
    sub preserve_case($$)
    {
        my ($old, $new) = @_;
        my ($state) = 0; # 0 = no change; 1 = lc; 2 = uc
        my ($i, $oldlen, $newlen, $c) = (0, length($old), length($new));
        my ($len) = $oldlen < $newlen ? $oldlen : $newlen;

        for ($i = 0; $i < $len; $i++) {
            if ($c = substr($old, $i, 1), $c =~ /[\W\d_]/) {
                $state = 0;
            } elsif (lc $c eq $c) {
                substr($new, $i, 1) = lc(substr($new, $i, 1));
                $state = 1;
            } else {
                substr($new, $i, 1) = uc(substr($new, $i, 1));
                $state = 2;
            }
        }
        # finish up with any remaining new (for when new is longer than old)
        if ($newlen > $oldlen) {
            if ($state == 1) {
                substr($new, $oldlen) = lc(substr($new, $oldlen));
            } elsif ($state == 2) {
                substr($new, $oldlen) = uc(substr($new, $oldlen));
            }
        }
        return $new;
    }
```

22.1.6 How can I make `\w` match national character sets?

Put `use locale;` in your script. The \w character class is taken from the current locale.

See *perllocale* for details.

22.1.7 How can I match a locale-smart version of `/[a-zA-Z]/`?

You can use the POSIX character class syntax `/[[:alpha:]]/` documented in *perlre*.

No matter which locale you are in, the alphabetic characters are the characters in \w without the digits and the underscore. As a regex, that looks like `/[^\W\d_]/`. Its complement, the non-alphabetics, is then everything in \W along with the digits and the underscore, or `/[\W\d_]/`.

22.1.8 How can I quote a variable to use in a regex?

The Perl parser will expand $variable and @variable references in regular expressions unless the delimiter is a single quote. Remember, too, that the right-hand side of a `s///` substitution is considered a double-quoted string (see *perlop* for more details). Remember also that any regex special characters will be acted on unless you precede the substitution with \Q. Here's an example:

```
    $string = "Placido P. Octopus";
    $regex  = "P.";
    $string =~ s/$regex/Polyp/;
    # $string is now "Polypacido P. Octopus"
```

Because `.` is special in regular expressions, and can match any single character, the regex `P.` here has matched the <Pl> in the original string.

To escape the special meaning of `.`, we use `\Q`:

```
    $string = "Placido P. Octopus";
    $regex  = "P.";

    $string =~ s/\Q$regex/Polyp/;
    # $string is now "Placido Polyp Octopus"
```

The use of `\Q` causes the <.> in the regex to be treated as a regular character, so that `P.` matches a `P` followed by a dot.

22.1.9 What is /o really for?

Using a variable in a regular expression match forces a re-evaluation (and perhaps recompilation) each time the regular expression is encountered. The `/o` modifier locks in the regex the first time it's used. This always happens in a constant regular expression, and in fact, the pattern was compiled into the internal format at the same time your entire program was.

Use of `/o` is irrelevant unless variable interpolation is used in the pattern, and if so, the regex engine will neither know nor care whether the variables change after the pattern is evaluated the *very first* time.

`/o` is often used to gain an extra measure of efficiency by not performing subsequent evaluations when you know it won't matter (because you know the variables won't change), or more rarely, when you don't want the regex to notice if they do.

For example, here's a "paragrep" program:

```
    $/ = '';  # paragraph mode
    $pat = shift;
    while (<>) {
        print if /$pat/o;
    }
```

22.1.10 How do I use a regular expression to strip C style comments from a file?

While this actually can be done, it's much harder than you'd think. For example, this one-liner

```
    perl -0777 -pe 's{/\*.*?\*/}{}gs' foo.c
```

will work in many but not all cases. You see, it's too simple-minded for certain kinds of C programs, in particular, those with what appear to be comments in quoted strings. For that, you'd need something like this, created by Jeffrey Friedl and later modified by Fred Curtis.

```
    $/ = undef;
    $_ = <>;
    s#/\*[^*]*\*+([^/*][^*]*\*+)*/|("(\\.|[^"\\])*"|'(\\.|[^'\\])*'|.[^/"'\\]*)#$2#gs
    print;
```

This could, of course, be more legibly written with the `/x` modifier, adding whitespace and comments. Here it is expanded, courtesy of Fred Curtis.

```
    s{
       /\*         ##  Start of /* ... */ comment
       [^*]*\*+    ##  Non-* followed by 1-or-more *'s
       (
         [^/*][^*]*\*+
       )*          ##  0-or-more things which don't start with /
                   ##    but do end with '*'
       /           ##  End of /* ... */ comment

     |         ##     OR  various things which aren't comments:

       (
         "           ##  Start of " ... " string
         (
           \\.           ##  Escaped char
         |               ##    OR
           [^"\\]        ##  Non "\
         )*
         "           ##  End of " ... " string

       |         ##     OR

         '           ##  Start of ' ... ' string
         (
           \\.           ##  Escaped char
         |               ##    OR
           [^'\\]        ##  Non '\
         )*
         '           ##  End of ' ... ' string

       |         ##     OR

         .           ##  Anything other char
         [^/"'\\]*   ##  Chars which doesn't start a comment, string or escape
       )
     }{$2}gxs;
```

A slight modification also removes C++ comments:

```
    s#/\*[^*]*\*+([^/*][^*]*\*+)*/|//[^\n]*|("(\\.|[^"\\])*"|'(\\.|[^'\\])*'|.[^/"'\\]*)#$2#gs;
```

22.1.11 Can I use Perl regular expressions to match balanced text?

Historically, Perl regular expressions were not capable of matching balanced text. As of more recent versions of perl including 5.6.1 experimental features have been added that make it possible to do this. Look at the documentation for the (??{ }) construct in recent perlre manual pages to see an example of matching balanced parentheses. Be sure to take special notice of the warnings present in the manual before making use of this feature.

CPAN contains many modules that can be useful for matching text depending on the context. Damian Conway provides some useful patterns in Regexp::Common. The module Text::Balanced provides a general solution to this problem.

One of the common applications of balanced text matching is working with XML and HTML. There are many modules available that support these needs. Two examples are HTML::Parser and XML::Parser. There are many others.

An elaborate subroutine (for 7-bit ASCII only) to pull out balanced and possibly nested single chars, like ` and ', { and }, or (and) can be found in http://www.cpan.org/authors/id/TOMC/scripts/pull_quotes.gz .

The C::Scan module from CPAN also contains such subs for internal use, but they are undocumented.

22.1.12 What does it mean that regexes are greedy? How can I get around it?

Most people mean that greedy regexes match as much as they can. Technically speaking, it's actually the quantifiers (?, *, +, {}) that are greedy rather than the whole pattern; Perl prefers local greed and immediate gratification to overall greed. To get non-greedy versions of the same quantifiers, use (??, *?, +?, {}?).

An example:

```
        $s1 = $s2 = "I am very very cold";
        $s1 =~ s/ve.*y //;      # I am cold
        $s2 =~ s/ve.*?y //;     # I am very cold
```

Notice how the second substitution stopped matching as soon as it encountered "y ". The *? quantifier effectively tells the regular expression engine to find a match as quickly as possible and pass control on to whatever is next in line, like you would if you were playing hot potato.

22.1.13 How do I process each word on each line?

Use the split function:

```
    while (<>) {
        foreach $word ( split ) {
            # do something with $word here
        }
    }
```

Note that this isn't really a word in the English sense; it's just chunks of consecutive non-whitespace characters.

To work with only alphanumeric sequences (including underscores), you might consider

```
    while (<>) {
        foreach $word (m/(\w+)/g) {
            # do something with $word here
        }
    }
```

22.1.14 How can I print out a word-frequency or line-frequency summary?

To do this, you have to parse out each word in the input stream. We'll pretend that by word you mean chunk of alphabetics, hyphens, or apostrophes, rather than the non-whitespace chunk idea of a word given in the previous question:

```
    while (<>) {
        while ( /(\b[^\W_\d][\w'-]+\b)/g ) {   # misses "`sheep'"
            $seen{$1}++;
        }
    }
    while ( ($word, $count) = each %seen ) {
        print "$count $word\n";
    }
```

If you wanted to do the same thing for lines, you wouldn't need a regular expression:

```
    while (<>) {
        $seen{$_}++;
    }
    while ( ($line, $count) = each %seen ) {
        print "$count $line";
    }
```

If you want these output in a sorted order, see *perlfaq4*: "How do I sort a hash (optionally by value instead of key)?".

22.1.15 How can I do approximate matching?

See the module String::Approx available from CPAN.

22.1.16 How do I efficiently match many regular expressions at once?

The following is extremely inefficient:

```
# slow but obvious way
@popstates = qw(CO ON MI WI MN);
while (defined($line = <>)) {
    for $state (@popstates) {
        if ($line =~ /\b$state\b/i) {
            print $line;
            last;
        }
    }
}
```

That's because Perl has to recompile all those patterns for each of the lines of the file. As of the 5.005 release, there's a much better approach, one which makes use of the new `qr//` operator:

```
# use spiffy new qr// operator, with /i flag even
use 5.005;
@popstates = qw(CO ON MI WI MN);
@poppats   = map { qr/\b$_\b/i } @popstates;
while (defined($line = <>)) {
    for $patobj (@poppats) {
        print $line if $line =~ /$patobj/;
    }
}
```

22.1.17 Why don't word-boundary searches with `\b` work for me?

Two common misconceptions are that `\b` is a synonym for `\s+` and that it's the edge between whitespace characters and non-whitespace characters. Neither is correct. `\b` is the place between a `\w` character and a `\W` character (that is, `\b` is the edge of a "word"). It's a zero-width assertion, just like `^`, `$`, and all the other anchors, so it doesn't consume any characters. *perlre* describes the behavior of all the regex metacharacters.

Here are examples of the incorrect application of `\b`, with fixes:

```
"two words" =~ /(\w+)\b(\w+)/;          # WRONG
"two words" =~ /(\w+)\s+(\w+)/;         # right

" =matchless= text" =~ /\b=(\w+)=\b/;   # WRONG
" =matchless= text" =~ /=(\w+)=/;       # right
```

Although they may not do what you thought they did, `\b` and `\B` can still be quite useful. For an example of the correct use of `\b`, see the example of matching duplicate words over multiple lines.

An example of using `\B` is the pattern `\Bis\B`. This will find occurrences of "is" on the insides of words only, as in "thistle", but not "this" or "island".

22.1.18 Why does using $ &, $ ‘, or $ ’ slow my program down?

Once Perl sees that you need one of these variables anywhere in the program, it provides them on each and every pattern match. The same mechanism that handles these provides for the use of $1, $2, etc., so you pay the same price for each regex that contains capturing parentheses. If you never use $&, etc., in your script, then regexes *without* capturing parentheses won't be penalized. So avoid $&, $', and $‘ if you can, but if you can't, once you've used them at all, use them at will because you've already paid the price. Remember that some algorithms really appreciate them. As of the 5.005 release. the $& variable is no longer "expensive" the way the other two are.

22.1.19 What good is \G in a regular expression?

You use the \G anchor to start the next match on the same string where the last match left off. The regular expression engine cannot skip over any characters to find the next match with this anchor, so \G is similar to the beginning of string anchor, ^. The \G anchor is typically used with the g flag. It uses the value of pos() as the position to start the next match. As the match operator makes successive matches, it updates pos() with the position of the next character past the last match (or the first character of the next match, depending on how you like to look at it). Each string has its own pos() value.

Suppose you want to match all of consective pairs of digits in a string like "1122a44" and stop matching when you encounter non-digits. You want to match 11 and 22 but the letter <a> shows up between 22 and 44 and you want to stop at a. Simply matching pairs of digits skips over the a and still matches 44.

```
        $_ = "1122a44";
        my @pairs = m/(\d\d)/g;   # qw( 11 22 44 )
```

If you use the \G anchor, you force the match after 22 to start with the a. The regular expression cannot match there since it does not find a digit, so the next match fails and the match operator returns the pairs it already found.

```
        $_ = "1122a44";
        my @pairs = m/\G(\d\d)/g; # qw( 11 22 )
```

You can also use the \G anchor in scalar context. You still need the g flag.

```
        $_ = "1122a44";
        while( m/\G(\d\d)/g )
                {
                print "Found $1\n";
                }
```

After the match fails at the letter a, perl resets pos() and the next match on the same string starts at the beginning.

```
        $_ = "1122a44";
        while( m/\G(\d\d)/g )
                {
                print "Found $1\n";
                }

        print "Found $1 after while" if m/(\d\d)/g; # finds "11"
```

You can disable pos() resets on fail with the c flag. Subsequent matches start where the last successful match ended (the value of pos()) even if a match on the same string as failed in the meantime. In this case, the match after the while() loop starts at the a (where the last match stopped), and since it does not use any anchor it can skip over the a to find "44".

```
        $_ = "1122a44";
        while( m/\G(\d\d)/gc )
                {
                print "Found $1\n";
                }
        print "Found $1 after while" if m/(\d\d)/g; # finds "44"
```

Typically you use the \G anchor with the c flag when you want to try a different match if one fails, such as in a tokenizer. Jeffrey Friedl offers this example which works in 5.004 or later.

```
    while (<>) {
      chomp;
      PARSER: {
           m/ \G( \d+\b    )/gcx   && do { print "number: $1\n";  redo; };
           m/ \G( \w+      )/gcx   && do { print "word:   $1\n";  redo; };
           m/ \G( \s+      )/gcx   && do { print "space:  $1\n";  redo; };
           m/ \G( [^\w\d]+ )/gcx   && do { print "other:  $1\n";  redo; };
      }
    }
```

For each line, the PARSER loop first tries to match a series of digits followed by a word boundary. This match has to start at the place the last match left off (or the beginning of the string on the first match). Since `m/ \G( \d+\b )/gcx` uses the c flag, if the string does not match that regular expression, perl does not reset pos() and the next match starts at the same position to try a different pattern.

22.1.20 Are Perl regexes DFAs or NFAs? Are they POSIX compliant?

While it's true that Perl's regular expressions resemble the DFAs (deterministic finite automata) of the egrep(1) program, they are in fact implemented as NFAs (non-deterministic finite automata) to allow backtracking and backreferencing. And they aren't POSIX-style either, because those guarantee worst-case behavior for all cases. (It seems that some people prefer guarantees of consistency, even when what's guaranteed is slowness.) See the book "Mastering Regular Expressions" (from O'Reilly) by Jeffrey Friedl for all the details you could ever hope to know on these matters (a full citation appears in *perlfaq2*).

22.1.21 What's wrong with using grep in a void context?

The problem is that grep builds a return list, regardless of the context. This means you're making Perl go to the trouble of building a list that you then just throw away. If the list is large, you waste both time and space. If your intent is to iterate over the list, then use a for loop for this purpose.

In perls older than 5.8.1, map suffers from this problem as well. But since 5.8.1, this has been fixed, and map is context aware - in void context, no lists are constructed.

22.1.22 How can I match strings with multibyte characters?

Starting from Perl 5.6 Perl has had some level of multibyte character support. Perl 5.8 or later is recommended. Supported multibyte character repertoires include Unicode, and legacy encodings through the Encode module. See *perluniintro*, *perlunicode*, and *Encode*.

If you are stuck with older Perls, you can do Unicode with the `Unicode::String` module, and character conversions using the `Unicode::Map8` and `Unicode::Map` modules. If you are using Japanese encodings, you might try using the jperl 5.005_03.

Finally, the following set of approaches was offered by Jeffrey Friedl, whose article in issue #5 of The Perl Journal talks about this very matter.

Let's suppose you have some weird Martian encoding where pairs of ASCII uppercase letters encode single Martian letters (i.e. the two bytes "CV" make a single Martian letter, as do the two bytes "SG", "VS", "XX", etc.). Other bytes represent single characters, just like ASCII.

So, the string of Martian "I am CVSGXX!" uses 12 bytes to encode the nine characters 'I', ' ', 'a', 'm', ' ', 'CV', 'SG', 'XX', '!'.

Now, say you want to search for the single character `/GX/`. Perl doesn't know about Martian, so it'll find the two bytes "GX" in the "I am CVSGXX!" string, even though that character isn't there: it just looks like it is because "SG" is next to "XX", but there's no real "GX". This is a big problem.

Here are a few ways, all painful, to deal with it:

```
$martian =~ s/([A-Z][A-Z])/ $1 /g; # Make sure adjacent ‘‘martian’’
                                   # bytes are no longer adjacent.
print "found GX!\n" if $martian =~ /GX/;
```

Or like this:

```
@chars = $martian =~ m/([A-Z][A-Z]|[^A-Z])/g;
# above is conceptually similar to:     @chars = $text =~ m/(.)/g;
#
foreach $char (@chars) {
    print "found GX!\n", last if $char eq 'GX';
}
```

Or like this:

```
while ($martian =~ m/\G([A-Z][A-Z]|.)/gs) {  # \G probably unneeded
    print "found GX!\n", last if $1 eq 'GX';
}
```

Here’s another, slightly less painful, way to do it from Benjamin Goldberg:

```
     $martian =~ m/
        (?!<[A-Z])
        (?:[A-Z][A-Z])*?
        GX
     /x;
```

This succeeds if the "martian" character GX is in the string, and fails otherwise. If you don’t like using (?!<), you can replace (?!<[A-Z]) with (?:^|[^A-Z]).

It does have the drawback of putting the wrong thing in $-[0] and $+[0], but this usually can be worked around.

22.1.23 How do I match a pattern that is supplied by the user?

Well, if it’s really a pattern, then just use

```
chomp($pattern = <STDIN>);
if ($line =~ /$pattern/) { }
```

Alternatively, since you have no guarantee that your user entered a valid regular expression, trap the exception this way:

```
if (eval { $line =~ /$pattern/ }) { }
```

If all you really want to search for a string, not a pattern, then you should either use the index() function, which is made for string searching, or if you can’t be disabused of using a pattern match on a non-pattern, then be sure to use `\Q`...`\E`, documented in *perlre*.

```
$pattern = <STDIN>;

open (FILE, $input) or die "Couldn't open input $input: $!; aborting";
while (<FILE>) {
    print if /\Q$pattern\E/;
}
close FILE;
```

22.2 AUTHOR AND COPYRIGHT

Copyright (c) 1997-2002 Tom Christiansen and Nathan Torkington. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

Chapter 23

perlfaq7

General Perl Language Issues ($Revision: 1.15 $, $Date: 2003/07/24 02:17:21 $)

23.1 DESCRIPTION

This section deals with general Perl language issues that don't clearly fit into any of the other sections.

23.1.1 Can I get a BNF/yacc/RE for the Perl language?

There is no BNF, but you can paw your way through the yacc grammar in perly.y in the source distribution if you're particularly brave. The grammar relies on very smart tokenizing code, so be prepared to venture into toke.c as well.

In the words of Chaim Frenkel: "Perl's grammar can not be reduced to BNF. The work of parsing perl is distributed between yacc, the lexer, smoke and mirrors."

23.1.2 What are all these $ @%&* punctuation signs, and how do I know when to use them?

They are type specifiers, as detailed in *perldata*:

```
    $ for scalar values (number, string or reference)
    @ for arrays
    % for hashes (associative arrays)
    & for subroutines (aka functions, procedures, methods)
    * for all types of that symbol name.  In version 4 you used them like
      pointers, but in modern perls you can just use references.
```

There are couple of other symbols that you're likely to encounter that aren't really type specifiers:

```
    <> are used for inputting a record from a filehandle.
    \  takes a reference to something.
```

Note that <FILE> is *neither* the type specifier for files nor the name of the handle. It is the <> operator applied to the handle FILE. It reads one line (well, record–see $/ in *perlvar*) from the handle FILE in scalar context, or *all* lines in list context. When performing open, close, or any other operation besides <> on files, or even when talking about the handle, do *not* use the brackets. These are correct: `eof(FH)`, `seek(FH, 0, 2)` and "copying from STDIN to FILE".

23.1.3 Do I always/never have to quote my strings or use semicolons and commas?

Normally, a bareword doesn't need to be quoted, but in most cases probably should be (and must be under `use strict`). But a hash key consisting of a simple word (that isn't the name of a defined subroutine) and the left-hand operand to the => operator both count as though they were quoted:

```
    This                    is like this
    ------------            ---------------
    $foo{line}              $foo{"line"}
    bar => stuff            "bar" => stuff
```

The final semicolon in a block is optional, as is the final comma in a list. Good style (see *perlstyle*) says to put them in except for one-liners:

```
    if ($whoops) { exit 1 }
    @nums = (1, 2, 3);

    if ($whoops) {
        exit 1;
    }
    @lines = (
        "There Beren came from mountains cold",
        "And lost he wandered under leaves",
    );
```

23.1.4 How do I skip some return values?

One way is to treat the return values as a list and index into it:

```
        $dir = (getpwnam($user))[7];
```

Another way is to use undef as an element on the left-hand-side:

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

You can also use a list slice to select only the elements that you need:

```
        ($dev, $ino, $uid, $gid) = ( stat($file) )[0,1,4,5];
```

23.1.5 How do I temporarily block warnings?

If you are running Perl 5.6.0 or better, the `use warnings` pragma allows fine control of what warning are produced. See *perllexwarn* for more details.

```
    {
        no warnings;           # temporarily turn off warnings
        $a = $b + $c;          # I know these might be undef
    }
```

If you have an older version of Perl, the `$^W` variable (documented in *perlvar*) controls runtime warnings for a block:

```
    {
        local $^W = 0;         # temporarily turn off warnings
        $a = $b + $c;          # I know these might be undef
    }
```

Note that like all the punctuation variables, you cannot currently use my() on `$^W`, only local().

23.1.6 What's an extension?

An extension is a way of calling compiled C code from Perl. Reading *perlxstut* is a good place to learn more about extensions.

23.1.7 Why do Perl operators have different precedence than C operators?

Actually, they don't. All C operators that Perl copies have the same precedence in Perl as they do in C. The problem is with operators that C doesn't have, especially functions that give a list context to everything on their right, eg. print, chmod, exec, and so on. Such functions are called "list operators" and appear as such in the precedence table in *perlop*.

A common mistake is to write:

```
    unlink $file || die "snafu";
```

This gets interpreted as:

```
    unlink ($file || die "snafu");
```

To avoid this problem, either put in extra parentheses or use the super low precedence `or` operator:

```
    (unlink $file) || die "snafu";
    unlink $file or die "snafu";
```

The "English" operators (`and`, `or`, `xor`, and `not`) deliberately have precedence lower than that of list operators for just such situations as the one above.

Another operator with surprising precedence is exponentiation. It binds more tightly even than unary minus, making `-2**2` product a negative not a positive four. It is also right-associating, meaning that `2**3**2` is two raised to the ninth power, not eight squared.

Although it has the same precedence as in C, Perl's `?:` operator produces an lvalue. This assigns $x to either $a or $b, depending on the trueness of $maybe:

```
    ($maybe ? $a : $b) = $x;
```

23.1.8 How do I declare/create a structure?

In general, you don't "declare" a structure. Just use a (probably anonymous) hash reference. See *perlref* and *perldsc* for details. Here's an example:

```
    $person = {};                   # new anonymous hash
    $person->{AGE}  = 24;           # set field AGE to 24
    $person->{NAME} = "Nat";        # set field NAME to "Nat"
```

If you're looking for something a bit more rigorous, try *perltoot*.

23.1.9 How do I create a module?

A module is a package that lives in a file of the same name. For example, the Hello::There module would live in Hello/There.pm. For details, read *perlmod*. You'll also find *Exporter* helpful. If you're writing a C or mixed-language module with both C and Perl, then you should study *perlxstut*.

The `h2xs` program will create stubs for all the important stuff for you:

```
  % h2xs -XA -n My::Module
```

The `-X` switch tells `h2xs` that you are not using `XS` extension code. The `-A` switch tells `h2xs` that you are not using the AutoLoader, and the `-n` switch specifies the name of the module. See *h2xs* for more details.

23.1.10 How do I create a class?

See *perltoot* for an introduction to classes and objects, as well as *perlobj* and *perlbot*.

23.1.11 How can I tell if a variable is tainted?

You can use the tainted() function of the Scalar::Util module, available from CPAN (or included with Perl since release 5.8.0). See also Laundering and Detecting Tainted Data in *perlsec*.

23.1.12 What's a closure?

Closures are documented in *perlref*.

Closure is a computer science term with a precise but hard-to-explain meaning. Closures are implemented in Perl as anonymous subroutines with lasting references to lexical variables outside their own scopes. These lexicals magically refer to the variables that were around when the subroutine was defined (deep binding).

Closures make sense in any programming language where you can have the return value of a function be itself a function, as you can in Perl. Note that some languages provide anonymous functions but are not capable of providing proper closures: the Python language, for example. For more information on closures, check out any textbook on functional programming. Scheme is a language that not only supports but encourages closures.

Here's a classic function-generating function:

```
sub add_function_generator {
  return sub { shift + shift };
}

$add_sub = add_function_generator();
$sum = $add_sub->(4,5);                # $sum is 9 now.
```

The closure works as a *function template* with some customization slots left out to be filled later. The anonymous subroutine returned by add_function_generator() isn't technically a closure because it refers to no lexicals outside its own scope.

Contrast this with the following make_adder() function, in which the returned anonymous function contains a reference to a lexical variable outside the scope of that function itself. Such a reference requires that Perl return a proper closure, thus locking in for all time the value that the lexical had when the function was created.

```
sub make_adder {
    my $addpiece = shift;
    return sub { shift + $addpiece };
}

$f1 = make_adder(20);
$f2 = make_adder(555);
```

Now `&$f1($n)` is always 20 plus whatever $n you pass in, whereas `&$f2($n)` is always 555 plus whatever $n you pass in. The $addpiece in the closure sticks around.

Closures are often used for less esoteric purposes. For example, when you want to pass in a bit of code into a function:

```
my $line;
timeout( 30, sub { $line = <STDIN> } );
```

If the code to execute had been passed in as a string, `'$line = <STDIN>'`, there would have been no way for the hypothetical timeout() function to access the lexical variable $line back in its caller's scope.

23.1.13 What is variable suicide and how can I prevent it?

Variable suicide is when you (temporarily or permanently) lose the value of a variable. It is caused by scoping through my() and local() interacting with either closures or aliased foreach() iterator variables and subroutine arguments. It used to be easy to inadvertently lose a variable's value this way, but now it's much harder. Take this code:

```
my $f = "foo";
sub T {
  while ($i++ < 3) { my $f = $f; $f .= "bar"; print $f, "\n" }
}
T;
print "Finally $f\n";
```

The $f that has "bar" added to it three times should be a new `$f` (`my $f` should create a new local variable each time through the loop). It isn't, however. This was a bug, now fixed in the latest releases (tested against 5.004_05, 5.005_03, and 5.005_56).

23.1.14 How can I pass/return a {Function, FileHandle, Array, Hash, Method, Regex}?

With the exception of regexes, you need to pass references to these objects. See Pass by Reference in *perlsub* for this particular question, and *perlref* for information on references.

See "Passing Regexes", below, for information on passing regular expressions.

Passing Variables and Functions

Regular variables and functions are quite easy to pass: just pass in a reference to an existing or anonymous variable or function:

```
func( \$some_scalar );

func( \@some_array  );
func( [ 1 .. 10 ]   );

func( \%some_hash   );
func( { this => 10, that => 20 }   );

func( \&some_func   );
func( sub { $_[0] ** $_[1] }   );
```

Passing Filehandles

As of Perl 5.6, you can represent filehandles with scalar variables which you treat as any other scalar.

```
open my $fh, $filename or die "Cannot open $filename! $!";
func( $fh );

sub func {
        my $passed_fh = shift;

        my $line = <$fh>;
        }
```

Before Perl 5.6, you had to use the `*FH` or `\*FH` notations. These are "typeglobs"–see Typeglobs and Filehandles in *perldata* and especially Pass by Reference in *perlsub* for more information.

Passing Regexes

To pass regexes around, you'll need to be using a release of Perl sufficiently recent as to support the `qr//` construct, pass around strings and use an exception-trapping eval, or else be very, very clever.

Here's an example of how to pass in a string to be regex compared using `qr//`:

```
sub compare($$) {
    my ($val1, $regex) = @_;
    my $retval = $val1 =~ /$regex/;
    return $retval;
}
$match = compare("old McDonald", qr/d.*D/i);
```

Notice how `qr//` allows flags at the end. That pattern was compiled at compile time, although it was executed later. The nifty `qr//` notation wasn't introduced until the 5.005 release. Before that, you had to approach this problem much less intuitively. For example, here it is again if you don't have `qr//`:

```
sub compare($$) {
    my ($val1, $regex) = @_;
    my $retval = eval { $val1 =~ /$regex/ };
    die if $@;
    return $retval;
}

$match = compare("old McDonald", q/($?i)d.*D/);
```

Make sure you never say something like this:

```
return eval "\$val =~ /$regex/";   # WRONG
```

or someone can sneak shell escapes into the regex due to the double interpolation of the eval and the double-quoted string. For example:

```
$pattern_of_evil = 'danger ${ system("rm -rf * &") } danger';

eval "\$string =~ /$pattern_of_evil/";
```

Those preferring to be very, very clever might see the O'Reilly book, *Mastering Regular Expressions*, by Jeffrey Friedl. Page 273's Build_MatchMany_Function() is particularly interesting. A complete citation of this book is given in *perlfaq2*.

Passing Methods

To pass an object method into a subroutine, you can do this:

```
call_a_lot(10, $some_obj, "methname")
sub call_a_lot {
    my ($count, $widget, $trick) = @_;
    for (my $i = 0; $i < $count; $i++) {
        $widget->$trick();
    }
}
```

Or, you can use a closure to bundle up the object, its method call, and arguments:

```
        my $whatnot =  sub { $some_obj->obfuscate(@args) };
        func($whatnot);
        sub func {
            my $code = shift;
            &$code();
        }
```

You could also investigate the can() method in the UNIVERSAL class (part of the standard perl distribution).

23.1.15 How do I create a static variable?

As with most things in Perl, TMTOWTDI. What is a "static variable" in other languages could be either a function-private variable (visible only within a single function, retaining its value between calls to that function), or a file-private variable (visible only to functions within the file it was declared in) in Perl.

Here's code to implement a function-private variable:

```
    BEGIN {
        my $counter = 42;
        sub prev_counter { return --$counter }
        sub next_counter { return $counter++ }
    }
```

Now prev_counter() and next_counter() share a private variable $counter that was initialized at compile time.

To declare a file-private variable, you'll still use a my(), putting the declaration at the outer scope level at the top of the file. Assume this is in file Pax.pm:

```
    package Pax;
    my $started = scalar(localtime(time()));

    sub begun { return $started }
```

When `use Pax` or `require Pax` loads this module, the variable will be initialized. It won't get garbage-collected the way most variables going out of scope do, because the begun() function cares about it, but no one else can get it. It is not called $Pax::started because its scope is unrelated to the package. It's scoped to the file. You could conceivably have several packages in that same file all accessing the same private variable, but another file with the same package couldn't get to it.

See Persistent Private Variables in *perlsub* for details.

23.1.16 What's the difference between dynamic and lexical (static) scoping? Between local() and my()?

`local($x)` saves away the old value of the global variable `$x` and assigns a new value for the duration of the subroutine *which is visible in other functions called from that subroutine*. This is done at run-time, so is called dynamic scoping. local() always affects global variables, also called package variables or dynamic variables.

`my($x)` creates a new variable that is only visible in the current subroutine. This is done at compile-time, so it is called lexical or static scoping. my() always affects private variables, also called lexical variables or (improperly) static(ly scoped) variables.

For instance:

```
    sub visible {
        print "var has value $var\n";
    }
    sub dynamic {
        local $var = 'local';   # new temporary value for the still-global
        visible();              #   variable called $var
    }

    sub lexical {
        my $var = 'private';    # new private variable, $var
        visible();              # (invisible outside of sub scope)
    }

    $var = 'global';

    visible();                  # prints global
    dynamic();                  # prints local
    lexical();                  # prints global
```

Notice how at no point does the value "private" get printed. That's because $var only has that value within the block of the lexical() function, and it is hidden from called subroutine.

In summary, local() doesn't make what you think of as private, local variables. It gives a global variable a temporary value. my() is what you're looking for if you want private variables.

See Private Variables via my() in *perlsub* and Temporary Values via local() in *perlsub* for excruciating details.

23.1.17 How can I access a dynamic variable while a similarly named lexical is in scope?

If you know your package, you can just mention it explicitly, as in $Some_Pack::var. Note that the notation $::var is **not** the dynamic $var in the current package, but rather the one in the "main" package, as though you had written $main::var.

```
        use vars '$var';
        local $var = "global";
        my    $var = "lexical";

        print "lexical is $var\n";
        print "global  is $main::var\n";
```

Alternatively you can use the compiler directive our() to bring a dynamic variable into the current lexical scope.

```
        require 5.006; # our() did not exist before 5.6
        use vars '$var';

        local $var = "global";
        my $var    = "lexical";

        print "lexical is $var\n";

        {
          our $var;
          print "global  is $var\n";
        }
```

23.1.18 What's the difference between deep and shallow binding?

In deep binding, lexical variables mentioned in anonymous subroutines are the same ones that were in scope when the subroutine was created. In shallow binding, they are whichever variables with the same names happen to be in scope when the subroutine is called. Perl always uses deep binding of lexical variables (i.e., those created with my()). However, dynamic variables (aka global, local, or package variables) are effectively shallowly bound. Consider this just one more reason not to use them. See the answer to §23.1.12.

23.1.19 Why doesn't "my($ foo) = <FILE>;" work right?

`my()` and `local()` give list context to the right hand side of =. The <FH> read operation, like so many of Perl's functions and operators, can tell which context it was called in and behaves appropriately. In general, the scalar() function can help. This function does nothing to the data itself (contrary to popular myth) but rather tells its argument to behave in whatever its scalar fashion is. If that function doesn't have a defined scalar behavior, this of course doesn't help you (such as with sort()).

To enforce scalar context in this particular case, however, you need merely omit the parentheses:

```
    local($foo) = <FILE>;           # WRONG
    local($foo) = scalar(<FILE>);   # ok
    local $foo  = <FILE>;           # right
```

You should probably be using lexical variables anyway, although the issue is the same here:

```
    my($foo) = <FILE>;  # WRONG
    my $foo  = <FILE>;  # right
```

23.1.20 How do I redefine a builtin function, operator, or method?

Why do you want to do that? :-)

If you want to override a predefined function, such as open(), then you'll have to import the new definition from a different module. See Overriding Built-in Functions in *perlsub*. There's also an example in Class::Template in *perltoot*.

If you want to overload a Perl operator, such as + or `**`, then you'll want to use the `use overload` pragma, documented in *overload*.

If you're talking about obscuring method calls in parent classes, see Overridden Methods in *perltoot*.

23.1.21 What's the difference between calling a function as &foo and foo()?

When you call a function as `&foo`, you allow that function access to your current @_ values, and you bypass prototypes. The function doesn't get an empty @_–it gets yours! While not strictly speaking a bug (it's documented that way in *perlsub*), it would be hard to consider this a feature in most cases.

When you call your function as `&foo()`, then you *do* get a new @_, but prototyping is still circumvented.

Normally, you want to call a function using `foo()`. You may only omit the parentheses if the function is already known to the compiler because it already saw the definition (`use` but not `require`), or via a forward reference or `use subs` declaration. Even in this case, you get a clean @_ without any of the old values leaking through where they don't belong.

23.1.22 How do I create a switch or case statement?

This is explained in more depth in the *perlsyn*. Briefly, there's no official case statement, because of the variety of tests possible in Perl (numeric comparison, string comparison, glob comparison, regex matching, overloaded comparisons, ...). Larry couldn't decide how best to do this, so he left it out, even though it's been on the wish list since perl1.

Starting from Perl 5.8 to get switch and case one can use the Switch extension and say:

```
        use Switch;
```

after which one has switch and case. It is not as fast as it could be because it's not really part of the language (it's done using source filters) but it is available, and it's very flexible.

But if one wants to use pure Perl, the general answer is to write a construct like this:

```
    for ($variable_to_test) {
        if    (/pat1/)  { }     # do something
        elsif (/pat2/)  { }     # do something else
        elsif (/pat3/)  { }     # do something else
        else            { }     # default
    }
```

Here's a simple example of a switch based on pattern matching, this time lined up in a way to make it look more like a switch statement. We'll do a multiway conditional based on the type of reference stored in $whatchamacallit:

```
    SWITCH: for (ref $whatchamacallit) {

        /^$/            && die "not a reference";

        /SCALAR/        && do {
                                print_scalar($$ref);
                                last SWITCH;
                        };

        /ARRAY/         && do {
                                print_array(@$ref);
                                last SWITCH;
                        };

        /HASH/          && do {
                                print_hash(%$ref);
                                last SWITCH;
                        };

        /CODE/          && do {
                                warn "can't print function ref";
                                last SWITCH;
                        };

        # DEFAULT

        warn "User defined type skipped";

    }
```

See perlsyn/"Basic BLOCKs and Switch Statements" for many other examples in this style.

Sometimes you should change the positions of the constant and the variable. For example, let's say you wanted to test which of many answers you were given, but in a case-insensitive way that also allows abbreviations. You can use the following technique if the strings all start with different characters or if you want to arrange the matches so that one takes precedence over another, as "SEND" has precedence over "STOP" here:

```
    chomp($answer = <>);
    if    ("SEND"  =~ /^\Q$answer/i) { print "Action is send\n"  }
    elsif ("STOP"  =~ /^\Q$answer/i) { print "Action is stop\n"  }
    elsif ("ABORT" =~ /^\Q$answer/i) { print "Action is abort\n" }
    elsif ("LIST"  =~ /^\Q$answer/i) { print "Action is list\n"  }
    elsif ("EDIT"  =~ /^\Q$answer/i) { print "Action is edit\n"  }
```

A totally different approach is to create a hash of function references.

```
    my %commands = (
        "happy" => \&joy,
        "sad",  => \&sullen,
        "done"  => sub { die "See ya!" },
        "mad"   => \&angry,
    );

    print "How are you? ";
    chomp($string = <STDIN>);
    if ($commands{$string}) {
        $commands{$string}->();
    } else {
        print "No such command: $string\n";
    }
```

23.1.23 How can I catch accesses to undefined variables, functions, or methods?

The AUTOLOAD method, discussed in Autoloading in *perlsub* and AUTOLOAD: Proxy Methods in *perltoot*, lets you capture calls to undefined functions and methods.

When it comes to undefined variables that would trigger a warning under `use warnings`, you can promote the warning to an error.

```
        use warnings FATAL => qw(uninitialized);
```

23.1.24 Why can't a method included in this same file be found?

Some possible reasons: your inheritance is getting confused, you've misspelled the method name, or the object is of the wrong type. Check out *perltoot* for details about any of the above cases. You may also use `print ref($object)` to find out the class `$object` was blessed into.

Another possible reason for problems is because you've used the indirect object syntax (eg, `find Guru "Samy"`) on a class name before Perl has seen that such a package exists. It's wisest to make sure your packages are all defined before you start using them, which will be taken care of if you use the `use` statement instead of `require`. If not, make sure to use arrow notation (eg., `Guru->find("Samy")`) instead. Object notation is explained in *perlobj*.

Make sure to read about creating modules in *perlmod* and the perils of indirect objects in Method Invocation in *perlobj*.

23.1.25 How can I find out my current package?

If you're just a random program, you can do this to find out what the currently compiled package is:

```
    my $packname = __PACKAGE__;
```

But, if you're a method and you want to print an error message that includes the kind of object you were called on (which is not necessarily the same as the one in which you were compiled):

```
    sub amethod {
        my $self  = shift;
        my $class = ref($self) || $self;
        warn "called me from a $class object";
    }
```

23.1.26 How can I comment out a large block of perl code?

You can use embedded POD to discard it. Enclose the blocks you want to comment out in POD markers, for example `=for nobody` and `=cut` (which marks ends of POD blocks).

```
    # program is here

    =for nobody

    all of this stuff

    here will be ignored
    by everyone

    =cut

    # program continues
```

The pod directives cannot go just anywhere. You must put a pod directive where the parser is expecting a new statement, not just in the middle of an expression or some other arbitrary grammar production.

See *perlpod* for more details.

23.1.27 How do I clear a package?

Use this code, provided by Mark-Jason Dominus:

```
    sub scrub_package {
        no strict 'refs';
        my $pack = shift;
        die "Shouldn't delete main package"
            if $pack eq "" || $pack eq "main";
        my $stash = *{$pack . '::'}{HASH};
        my $name;
        foreach $name (keys %$stash) {
            my $fullname = $pack . '::' . $name;
            # Get rid of everything with that name.
            undef $$fullname;
            undef @$fullname;
            undef %$fullname;
            undef &$fullname;
            undef *$fullname;
        }
    }
```

Or, if you're using a recent release of Perl, you can just use the Symbol::delete_package() function instead.

23.1.28 How can I use a variable as a variable name?

Beginners often think they want to have a variable contain the name of a variable.

```
    $fred    = 23;
    $varname = "fred";
    ++$$varname;         # $fred now 24
```

This works *sometimes*, but it is a very bad idea for two reasons.

The first reason is that this technique *only works on global variables*. That means that if $fred is a lexical variable created with my() in the above example, the code wouldn't work at all: you'd accidentally access the global and skip right over the private lexical altogether. Global variables are bad because they can easily collide accidentally and in general make for non-scalable and confusing code.

Symbolic references are forbidden under the `use strict` pragma. They are not true references and consequently are not reference counted or garbage collected.

The other reason why using a variable to hold the name of another variable is a bad idea is that the question often stems from a lack of understanding of Perl data structures, particularly hashes. By using symbolic references, you are just using the package's symbol-table hash (like `%main::`) instead of a user-defined hash. The solution is to use your own hash or a real reference instead.

```
    $USER_VARS{"fred"} = 23;
    $varname = "fred";
    $USER_VARS{$varname}++;  # not $$varname++
```

There we're using the %USER_VARS hash instead of symbolic references. Sometimes this comes up in reading strings from the user with variable references and wanting to expand them to the values of your perl program's variables. This is also a bad idea because it conflates the program-addressable namespace and the user-addressable one. Instead of reading a string and expanding it to the actual contents of your program's own variables:

```
    $str = 'this has a $fred and $barney in it';
    $str =~ s/(\$\w+)/$1/eeg;             # need double eval
```

it would be better to keep a hash around like %USER_VARS and have variable references actually refer to entries in that hash:

```
    $str =~ s/\$(\w+)/$USER_VARS{$1}/g;   # no /e here at all
```

That's faster, cleaner, and safer than the previous approach. Of course, you don't need to use a dollar sign. You could use your own scheme to make it less confusing, like bracketed percent symbols, etc.

```
    $str = 'this has a %fred% and %barney% in it';
    $str =~ s/%(\w+)%/$USER_VARS{$1}/g;   # no /e here at all
```

Another reason that folks sometimes think they want a variable to contain the name of a variable is because they don't know how to build proper data structures using hashes. For example, let's say they wanted two hashes in their program: %fred and %barney, and that they wanted to use another scalar variable to refer to those by name.

```
    $name = "fred";
    $$name{WIFE} = "wilma";     # set %fred

    $name = "barney";
    $$name{WIFE} = "betty";     # set %barney
```

This is still a symbolic reference, and is still saddled with the problems enumerated above. It would be far better to write:

```
    $folks{"fred"}{WIFE}   = "wilma";
    $folks{"barney"}{WIFE} = "betty";
```

And just use a multilevel hash to start with.

The only times that you absolutely *must* use symbolic references are when you really must refer to the symbol table. This may be because it's something that can't take a real reference to, such as a format name. Doing so may also be important for method calls, since these always go through the symbol table for resolution.

In those cases, you would turn off `strict 'refs'` temporarily so you can play around with the symbol table. For example:

```
    @colors = qw(red blue green yellow orange purple violet);
    for my $name (@colors) {
        no strict 'refs';  # renege for the block
        *$name = sub { "<FONT COLOR='$name'>@_</FONT>" };
    }
```

All those functions (red(), blue(), green(), etc.) appear to be separate, but the real code in the closure actually was compiled only once.

So, sometimes you might want to use symbolic references to directly manipulate the symbol table. This doesn't matter for formats, handles, and subroutines, because they are always global–you can't use my() on them. For scalars, arrays, and hashes, though–and usually for subroutines– you probably only want to use hard references.

23.1.29 What does "bad interpreter" mean?

The "bad interpreter" message comes from the shell, not perl. The actual message may vary depending on your platform, shell, and locale settings.

If you see "bad interpreter - no such file or directory", the first line in your perl script (the "shebang" line) does not contain the right path to perl (or any other program capable of running scripts). Sometimes this happens when you move the script from one machine to another and each machine has a different path to perl—/usr/bin/perl versus /usr/local/bin/perl for instance.

If you see "bad interpreter: Permission denied", you need to make your script executable.

In either case, you should still be able to run the scripts with perl explicitly:

```
        % perl script.pl
```

If you get a message like "perl: command not found", perl is not in your PATH, which might also mean that the location of perl is not where you expect it so you need to adjust your shebang line.

23.2 AUTHOR AND COPYRIGHT

Chapter 24

perlfaq8

System Interaction ($Revision: 1.17 $, $Date: 2003/01/26 17:44:04 $)

24.1 DESCRIPTION

This section of the Perl FAQ covers questions involving operating system interaction. Topics include interprocess communication (IPC), control over the user-interface (keyboard, screen and pointing devices), and most anything else not related to data manipulation.

Read the FAQs and documentation specific to the port of perl to your operating system (eg, *perlvms*, *perlplan9*, ...). These should contain more detailed information on the vagaries of your perl.

24.1.1 How do I find out which operating system I'm running under?

The $ˆO variable ($OSNAME if you use English) contains an indication of the name of the operating system (not its release number) that your perl binary was built for.

24.1.2 How come exec() doesn't return?

Because that's what it does: it replaces your currently running program with a different one. If you want to keep going (as is probably the case if you're asking this question) use system() instead.

24.1.3 How do I do fancy stuff with the keyboard/screen/mouse?

How you access/control keyboards, screens, and pointing devices ("mice") is system-dependent. Try the following modules:

Keyboard

```
        Term::Cap               Standard perl distribution
        Term::ReadKey           CPAN
        Term::ReadLine::Gnu     CPAN
        Term::ReadLine::Perl    CPAN
        Term::Screen            CPAN
```

Screen

```
        Term::Cap               Standard perl distribution
        Curses                  CPAN
        Term::ANSIColor         CPAN
```

Mouse

Tk CPAN

Some of these specific cases are shown below.

24.1.4 How do I print something out in color?

In general, you don't, because you don't know whether the recipient has a color-aware display device. If you know that they have an ANSI terminal that understands color, you can use the Term::ANSIColor module from CPAN:

```
use Term::ANSIColor;
print color("red"), "Stop!\n", color("reset");
print color("green"), "Go!\n", color("reset");
```

Or like this:

```
use Term::ANSIColor qw(:constants);
print RED, "Stop!\n", RESET;
print GREEN, "Go!\n", RESET;
```

24.1.5 How do I read just one key without waiting for a return key?

Controlling input buffering is a remarkably system-dependent matter. On many systems, you can just use the **stty** command as shown in getc in *perlfunc*, but as you see, that's already getting you into portability snags.

```
open(TTY, "+</dev/tty") or die "no tty: $!";
system "stty  cbreak </dev/tty >/dev/tty 2>&1";
$key = getc(TTY);           # perhaps this works
# OR ELSE
sysread(TTY, $key, 1);      # probably this does
system "stty -cbreak </dev/tty >/dev/tty 2>&1";
```

The Term::ReadKey module from CPAN offers an easy-to-use interface that should be more efficient than shelling out to **stty** for each key. It even includes limited support for Windows.

```
use Term::ReadKey;
ReadMode('cbreak');
$key = ReadKey(0);
ReadMode('normal');
```

However, using the code requires that you have a working C compiler and can use it to build and install a CPAN module. Here's a solution using the standard POSIX module, which is already on your systems (assuming your system supports POSIX).

```
use HotKey;
$key = readkey();
```

And here's the HotKey module, which hides the somewhat mystifying calls to manipulate the POSIX termios structures.

```
# HotKey.pm
package HotKey;
```

```
@ISA = qw(Exporter);
@EXPORT = qw(cbreak cooked readkey);

use strict;
use POSIX qw(:termios_h);
my ($term, $oterm, $echo, $noecho, $fd_stdin);

$fd_stdin = fileno(STDIN);
$term     = POSIX::Termios->new();
$term->getattr($fd_stdin);
$oterm     = $term->getlflag();

$echo     = ECHO | ECHOK | ICANON;
$noecho   = $oterm & ~$echo;

sub cbreak {
    $term->setlflag($noecho);  # ok, so i don't want echo either
    $term->setcc(VTIME, 1);
    $term->setattr($fd_stdin, TCSANOW);
}

sub cooked {
    $term->setlflag($oterm);
    $term->setcc(VTIME, 0);
    $term->setattr($fd_stdin, TCSANOW);
}

sub readkey {
    my $key = '';
    cbreak();
    sysread(STDIN, $key, 1);
    cooked();
    return $key;
}

END { cooked() }

1;
```

24.1.6 How do I check whether input is ready on the keyboard?

The easiest way to do this is to read a key in nonblocking mode with the Term::ReadKey module from CPAN, passing it an argument of -1 to indicate not to block:

```
use Term::ReadKey;

ReadMode('cbreak');

if (defined ($char = ReadKey(-1)) ) {
    # input was waiting and it was $char
} else {
    # no input was waiting
}

ReadMode('normal');                  # restore normal tty settings
```

24.1.7 How do I clear the screen?

If you only have do so infrequently, use `system`:

```
system("clear");
```

If you have to do this a lot, save the clear string so you can print it 100 times without calling a program 100 times:

```
$clear_string = ‘clear‘;
print $clear_string;
```

If you’re planning on doing other screen manipulations, like cursor positions, etc, you might wish to use Term::Cap module:

```
use Term::Cap;
$terminal = Term::Cap->Tgetent( {OSPEED => 9600} );
$clear_string = $terminal->Tputs(’cl’);
```

24.1.8 How do I get the screen size?

If you have Term::ReadKey module installed from CPAN, you can use it to fetch the width and height in characters and in pixels:

```
use Term::ReadKey;
($wchar, $hchar, $wpixels, $hpixels) = GetTerminalSize();
```

This is more portable than the raw `ioctl`, but not as illustrative:

```
require ’sys/ioctl.ph’;
die "no TIOCGWINSZ " unless defined &TIOCGWINSZ;
open(TTY, "+</dev/tty")                     or die "No tty: $!";
unless (ioctl(TTY, &TIOCGWINSZ, $winsize=’’)) {
    die sprintf "$0: ioctl TIOCGWINSZ (%08x: $!)\n", &TIOCGWINSZ;
}
($row, $col, $xpixel, $ypixel) = unpack(’S4’, $winsize);
print "(row,col) = ($row,$col)";
print "  (xpixel,ypixel) = ($xpixel,$ypixel)" if $xpixel || $ypixel;
print "\n";
```

24.1.9 How do I ask the user for a password?

(This question has nothing to do with the web. See a different FAQ for that.)

There’s an example of this in crypt in *perlfunc*). First, you put the terminal into "no echo" mode, then just read the password normally. You may do this with an old-style ioctl() function, POSIX terminal control (see *POSIX* or its documentation the Camel Book), or a call to the **stty** program, with varying degrees of portability.

You can also do this for most systems using the Term::ReadKey module from CPAN, which is easier to use and in theory more portable.

```
use Term::ReadKey;

ReadMode(’noecho’);
$password = ReadLine(0);
```

24.1.10 How do I read and write the serial port?

This depends on which operating system your program is running on. In the case of Unix, the serial ports will be accessible through files in /dev; on other systems, device names will doubtless differ. Several problem areas common to all device interaction are the following:

lockfiles

Your system may use lockfiles to control multiple access. Make sure you follow the correct protocol. Unpredictable behavior can result from multiple processes reading from one device.

open mode

If you expect to use both read and write operations on the device, you'll have to open it for update (see open in *perlfunc* for details). You may wish to open it without running the risk of blocking by using sysopen() and `O_RDWR|O_NDELAY|O_NOCTTY` from the Fcntl module (part of the standard perl distribution). See sysopen in *perlfunc* for more on this approach.

end of line

Some devices will be expecting a "\r" at the end of each line rather than a "\n". In some ports of perl, "\r" and "\n" are different from their usual (Unix) ASCII values of "\012" and "\015". You may have to give the numeric values you want directly, using octal ("\015"), hex ("0x0D"), or as a control-character specification ("\cM").

```
print DEV "atv1\012";       # wrong, for some devices
print DEV "atv1\015";       # right, for some devices
```

Even though with normal text files a "\n" will do the trick, there is still no unified scheme for terminating a line that is portable between Unix, DOS/Win, and Macintosh, except to terminate *ALL* line ends with "\015\012", and strip what you don't need from the output. This applies especially to socket I/O and autoflushing, discussed next.

flushing output

If you expect characters to get to your device when you print() them, you'll want to autoflush that filehandle. You can use select() and the $| variable to control autoflushing (see $| in *perlvar* and select in *perlfunc*, or *perlfaq5*, "How do I flush/unbuffer an output filehandle? Why must I do this?"):

```
$oldh = select(DEV);
$| = 1;
select($oldh);
```

You'll also see code that does this without a temporary variable, as in

```
select((select(DEV), $| = 1)[0]);
```

Or if you don't mind pulling in a few thousand lines of code just because you're afraid of a little $| variable:

```
use IO::Handle;
DEV->autoflush(1);
```

As mentioned in the previous item, this still doesn't work when using socket I/O between Unix and Macintosh. You'll need to hard code your line terminators, in that case.

non-blocking input

If you are doing a blocking read() or sysread(), you'll have to arrange for an alarm handler to provide a timeout (see alarm in *perlfunc*). If you have a non-blocking open, you'll likely have a non-blocking read, which means you may have to use a 4-arg select() to determine whether I/O is ready on that device (see select in *perlfunc*.

While trying to read from his caller-id box, the notorious Jamie Zawinski <jwz@netscape.com>, after much gnashing of teeth and fighting with sysread, sysopen, POSIX's tcgetattr business, and various other functions that go bump in the night, finally came up with this:

```
sub open_modem {
    use IPC::Open2;
    my $stty = `/bin/stty -g`;
    open2( \*MODEM_IN, \*MODEM_OUT, "cu -l$modem_device -s2400 2>&1");
    # starting cu hoses /dev/tty's stty settings, even when it has
    # been opened on a pipe...
    system("/bin/stty $stty");
    $_ = <MODEM_IN>;
    chomp;
    if ( !m/^Connected/ ) {
        print STDERR "$0: cu printed `$_' instead of `Connected'\n";
    }
}
```

24.1.11 How do I decode encrypted password files?

You spend lots and lots of money on dedicated hardware, but this is bound to get you talked about.

Seriously, you can't if they are Unix password files–the Unix password system employs one-way encryption. It's more like hashing than encryption. The best you can check is whether something else hashes to the same string. You can't turn a hash back into the original string. Programs like Crack can forcibly (and intelligently) try to guess passwords, but don't (can't) guarantee quick success.

If you're worried about users selecting bad passwords, you should proactively check when they try to change their password (by modifying passwd(1), for example).

24.1.12 How do I start a process in the background?

Several modules can start other processes that do not block your Perl program. You can use IPC::Open3, Parallel::Jobs, IPC::Run, and some of the POE modules. See CPAN for more details.

You could also use

```
system("cmd &")
```

or you could use fork as documented in fork in *perlfunc*, with further examples in *perlipc*. Some things to be aware of, if you're on a Unix-like system:

STDIN, STDOUT, and STDERR are shared

Both the main process and the backgrounded one (the "child" process) share the same STDIN, STDOUT and STDERR filehandles. If both try to access them at once, strange things can happen. You may want to close or reopen these for the child. You can get around this with opening a pipe (see open in *perlfunc*) but on some systems this means that the child process cannot outlive the parent.

Signals

You'll have to catch the SIGCHLD signal, and possibly SIGPIPE too. SIGCHLD is sent when the backgrounded process finishes. SIGPIPE is sent when you write to a filehandle whose child process has closed (an untrapped SIGPIPE can cause your program to silently die). This is not an issue with `system("cmd&")`.

Zombies

You have to be prepared to "reap" the child process when it finishes.

```
$SIG{CHLD} = sub { wait };
```

```
    $SIG{CHLD} = 'IGNORE';
```

You can also use a double fork. You immediately wait() for your first child, and the init daemon will wait() for your grandchild once it exits.

```
        unless ($pid = fork) {
                unless (fork) {
            exec "what you really wanna do";
            die "exec failed!";
                }
        exit 0;
        }
    waitpid($pid,0);
```

See Signals in *perlipc* for other examples of code to do this. Zombies are not an issue with `system("prog &")`.

24.1.13 How do I trap control characters/signals?

You don't actually "trap" a control character. Instead, that character generates a signal which is sent to your terminal's currently foregrounded process group, which you then trap in your process. Signals are documented in Signals in *perlipc* and the section on “Signals” in the Camel.

Be warned that very few C libraries are re-entrant. Therefore, if you attempt to print() in a handler that got invoked during another stdio operation your internal structures will likely be in an inconsistent state, and your program will dump core. You can sometimes avoid this by using syswrite() instead of print().

Unless you're exceedingly careful, the only safe things to do inside a signal handler are (1) set a variable and (2) exit. In the first case, you should only set a variable in such a way that malloc() is not called (eg, by setting a variable that already has a value).

For example:

```
    $Interrupted = 0;   # to ensure it has a value
    $SIG{INT} = sub {
        $Interrupted++;
        syswrite(STDERR, "ouch\n", 5);
    }
```

However, because syscalls restart by default, you'll find that if you're in a "slow" call, such as <FH>, read(), connect(), or wait(), that the only way to terminate them is by "longjumping" out; that is, by raising an exception. See the time-out handler for a blocking flock() in Signals in *perlipc* or the section on “Signals” in the Camel book.

24.1.14 How do I modify the shadow password file on a Unix system?

If perl was installed correctly and your shadow library was written properly, the getpw*() functions described in *perlfunc* should in theory provide (read-only) access to entries in the shadow password file. To change the file, make a new shadow password file (the format varies from system to system–see *passwd* for specifics) and use pwd_mkdb(8) to install it (see pwd_mkdb for more details).

24.1.15 How do I set the time and date?

Assuming you're running under sufficient permissions, you should be able to set the system-wide date and time by running the date(1) program. (There is no way to set the time and date on a per-process basis.) This mechanism will work for Unix, MS-DOS, Windows, and NT; the VMS equivalent is `set time`.

However, if all you want to do is change your time zone, you can probably get away with setting an environment variable:

```
    $ENV{TZ} = "MST7MDT";                  # unixish
    $ENV{'SYS$TIMEZONE_DIFFERENTIAL'}="-5" # vms
    system "trn comp.lang.perl.misc";
```

24.1.16 How can I sleep() or alarm() for under a second?

If you want finer granularity than the 1 second that the sleep() function provides, the easiest way is to use the select() function as documented in select in *perlfunc*. Try the Time::HiRes and the BSD::Itimer modules (available from CPAN, and starting from Perl 5.8 Time::HiRes is part of the standard distribution).

24.1.17 How can I measure time under a second?

In general, you may not be able to. The Time::HiRes module (available from CPAN, and starting from Perl 5.8 part of the standard distribution) provides this functionality for some systems.

If your system supports both the syscall() function in Perl as well as a system call like gettimeofday(2), then you may be able to do something like this:

```
require 'sys/syscall.ph';

$TIMEVAL_T = "LL";

$done = $start = pack($TIMEVAL_T, ());

syscall(&SYS_gettimeofday, $start, 0) != -1
           or die "gettimeofday: $!";

   ##########################
   # DO YOUR OPERATION HERE #
   ##########################

syscall( &SYS_gettimeofday, $done, 0) != -1
       or die "gettimeofday: $!";

@start = unpack($TIMEVAL_T, $start);
@done  = unpack($TIMEVAL_T, $done);

# fix microseconds
for ($done[1], $start[1]) { $_ /= 1_000_000 }

$delta_time = sprintf "%.4f", ($done[0]  + $done[1]  )
                                        -
                             ($start[0] + $start[1] );
```

24.1.18 How can I do an atexit() or setjmp()/longjmp()? (Exception handling)

Release 5 of Perl added the END block, which can be used to simulate atexit(). Each package's END block is called when the program or thread ends (see *perlmod* manpage for more details).

For example, you can use this to make sure your filter program managed to finish its output without filling up the disk:

```
END {
    close(STDOUT) || die "stdout close failed: $!";
}
```

The END block isn't called when untrapped signals kill the program, though, so if you use END blocks you should also use

```
    use sigtrap qw(die normal-signals);
```

Perl's exception-handling mechanism is its eval() operator. You can use eval() as setjmp and die() as longjmp. For details of this, see the section on signals, especially the time-out handler for a blocking flock() in Signals in *perlipc* or the section on "Signals" in the Camel Book.

If exception handling is all you're interested in, try the exceptions.pl library (part of the standard perl distribution).

If you want the atexit() syntax (and an rmexit() as well), try the AtExit module available from CPAN.

24.1.19 Why doesn't my sockets program work under System V (Solaris)? What does the error message "Protocol not supported" mean?

Some Sys-V based systems, notably Solaris 2.X, redefined some of the standard socket constants. Since these were constant across all architectures, they were often hardwired into perl code. The proper way to deal with this is to "use Socket" to get the correct values.

Note that even though SunOS and Solaris are binary compatible, these values are different. Go figure.

24.1.20 How can I call my system's unique C functions from Perl?

In most cases, you write an external module to do it–see the answer to "Where can I learn about linking C with Perl? [h2xs, xsubpp]". However, if the function is a system call, and your system supports syscall(), you can use the syscall function (documented in *perlfunc*).

Remember to check the modules that came with your distribution, and CPAN as well—someone may already have written a module to do it. On Windows, try Win32::API. On Macs, try Mac::Carbon. If no module has an interface to the C function, you can inline a bit of C in your Perl source with Inline::C.

24.1.21 Where do I get the include files to do ioctl() or syscall()?

Historically, these would be generated by the h2ph tool, part of the standard perl distribution. This program converts cpp(1) directives in C header files to files containing subroutine definitions, like &SYS_getitimer, which you can use as arguments to your functions. It doesn't work perfectly, but it usually gets most of the job done. Simple files like *errno.h*, *syscall.h*, and *socket.h* were fine, but the hard ones like *ioctl.h* nearly always need to hand-edited. Here's how to install the *.ph files:

```
    1.  become super-user
    2.  cd /usr/include
    3.  h2ph *.h */*.h
```

If your system supports dynamic loading, for reasons of portability and sanity you probably ought to use h2xs (also part of the standard perl distribution). This tool converts C header files to Perl extensions. See *perlxstut* for how to get started with h2xs.

If your system doesn't support dynamic loading, you still probably ought to use h2xs. See *perlxstut* and *ExtUtils::MakeMaker* for more information (in brief, just use **make perl** instead of a plain **make** to rebuild perl with a new static extension).

24.1.22 Why do setuid perl scripts complain about kernel problems?

Some operating systems have bugs in the kernel that make setuid scripts inherently insecure. Perl gives you a number of options (described in *perlsec*) to work around such systems.

24.1.23 How can I open a pipe both to and from a command?

The IPC::Open2 module (part of the standard perl distribution) is an easy-to-use approach that internally uses pipe(), fork(), and exec() to do the job. Make sure you read the deadlock warnings in its documentation, though (see *IPC::Open2*). See Bidirectional Communication with Another Process in *perlipc* and Bidirectional Communication with Yourself in *perlipc*

You may also use the IPC::Open3 module (part of the standard perl distribution), but be warned that it has a different order of arguments from IPC::Open2 (see *IPC::Open3*).

24.1.24 Why can't I get the output of a command with system()?

You're confusing the purpose of system() and backticks (“). system() runs a command and returns exit status information (as a 16 bit value: the low 7 bits are the signal the process died from, if any, and the high 8 bits are the actual exit value). Backticks (“) run a command and return what it sent to STDOUT.

```
$exit_status   = system("mail-users");
$output_string = ‘ls‘;
```

24.1.25 How can I capture STDERR from an external command?

There are three basic ways of running external commands:

```
system $cmd;                # using system()
$output = ‘$cmd‘;           # using backticks (‘‘)
open (PIPE, "cmd |");       # using open()
```

With system(), both STDOUT and STDERR will go the same place as the script's STDOUT and STDERR, unless the system() command redirects them. Backticks and open() read **only** the STDOUT of your command.

You can also use the open3() function from IPC::Open3. Benjamin Goldberg provides some sample code:

To capture a program's STDOUT, but discard its STDERR:

```
use IPC::Open3;
use File::Spec;
use Symbol qw(gensym);
open(NULL, ">", File::Spec->devnull);
my $pid = open3(gensym, \*PH, ">&NULL", "cmd");
while( <PH> ) { }
waitpid($pid, 0);
```

To capture a program's STDERR, but discard its STDOUT:

```
use IPC::Open3;
use File::Spec;
use Symbol qw(gensym);
open(NULL, ">", File::Spec->devnull);
my $pid = open3(gensym, ">&NULL", \*PH, "cmd");
while( <PH> ) { }
waitpid($pid, 0);
```

To capture a program's STDERR, and let its STDOUT go to our own STDERR:

```
use IPC::Open3;
use Symbol qw(gensym);
my $pid = open3(gensym, ">&STDERR", \*PH, "cmd");
while( <PH> ) { }
waitpid($pid, 0);
```

To read both a command's STDOUT and its STDERR separately, you can redirect them to temp files, let the command run, then read the temp files:

```
use IPC::Open3;
use Symbol qw(gensym);
use IO::File;
local *CATCHOUT = IO::File->new_tempfile;
local *CATCHERR = IO::File->new_tempfile;
my $pid = open3(gensym, ">&CATCHOUT", ">&CATCHERR", "cmd");
waitpid($pid, 0);
seek $_, 0, 0 for \*CATCHOUT, \*CATCHERR;
while( <CATCHOUT> ) {}
while( <CATCHERR> ) {}
```

But there's no real need for *both* to be tempfiles... the following should work just as well, without deadlocking:

```
use IPC::Open3;
use Symbol qw(gensym);
use IO::File;
local *CATCHERR = IO::File->new_tempfile;
my $pid = open3(gensym, \*CATCHOUT, ">&CATCHERR", "cmd");
while( <CATCHOUT> ) {}
waitpid($pid, 0);
seek CATCHERR, 0, 0;
while( <CATCHERR> ) {}
```

And it'll be faster, too, since we can begin processing the program's stdout immediately, rather than waiting for the program to finish.

With any of these, you can change file descriptors before the call:

```
open(STDOUT, ">logfile");
system("ls");
```

or you can use Bourne shell file-descriptor redirection:

```
$output = `$cmd 2>some_file`;
open (PIPE, "cmd 2>some_file |");
```

You can also use file-descriptor redirection to make STDERR a duplicate of STDOUT:

```
$output = `$cmd 2>&1`;
open (PIPE, "cmd 2>&1 |");
```

Note that you *cannot* simply open STDERR to be a dup of STDOUT in your Perl program and avoid calling the shell to do the redirection. This doesn't work:

```
open(STDERR, ">&STDOUT");
$alloutput = `cmd args`;  # stderr still escapes
```

This fails because the open() makes STDERR go to where STDOUT was going at the time of the open(). The backticks then make STDOUT go to a string, but don't change STDERR (which still goes to the old STDOUT).

Note that you *must* use Bourne shell (sh(1)) redirection syntax in backticks, not csh(1)! Details on why Perl's system() and backtick and pipe opens all use the Bourne shell are in the *versus/csh.whynot* article in the "Far More Than You Ever Wanted To Know" collection in http://www.cpan.org/misc/olddoc/FMTEYEWTK.tgz . To capture a command's STDERR and STDOUT together:

```
$output = `cmd 2>&1`;                       # either with backticks
$pid = open(PH, "cmd 2>&1 |");              # or with an open pipe
while (<PH>) { }                            #    plus a read
```

To capture a command's STDOUT but discard its STDERR:

```
    $output = `cmd 2>/dev/null`;            # either with backticks
    $pid = open(PH, "cmd 2>/dev/null |");   # or with an open pipe
    while (<PH>) { }                        #    plus a read
```

To capture a command's STDERR but discard its STDOUT:

```
    $output = `cmd 2>&1 1>/dev/null`;           # either with backticks
    $pid = open(PH, "cmd 2>&1 1>/dev/null |");  # or with an open pipe
    while (<PH>) { }                            #    plus a read
```

To exchange a command's STDOUT and STDERR in order to capture the STDERR but leave its STDOUT to come out our old STDERR:

```
    $output = `cmd 3>&1 1>&2 2>&3 3>&-`;        # either with backticks
    $pid = open(PH, "cmd 3>&1 1>&2 2>&3 3>&-|");# or with an open pipe
    while (<PH>) { }                            #    plus a read
```

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");
```

Ordering is important in all these examples. That's because the shell processes file descriptor redirections in strictly left to right order.

```
    system("prog args 1>tmpfile 2>&1");
    system("prog args 2>&1 1>tmpfile");
```

The first command sends both standard out and standard error to the temporary file. The second command sends only the old standard output there, and the old standard error shows up on the old standard out.

24.1.26 Why doesn't open() return an error when a pipe open fails?

If the second argument to a piped open() contains shell metacharacters, perl fork()s, then exec()s a shell to decode the metacharacters and eventually run the desired program. If the program couldn't be run, it's the shell that gets the message, not Perl. All your Perl program can find out is whether the shell itself could be successfully started. You can still capture the shell's STDERR and check it for error messages. See §24.1.25 elsewhere in this document, or use the IPC::Open3 module.

If there are no shell metacharacters in the argument of open(), Perl runs the command directly, without using the shell, and can correctly report whether the command started.

24.1.27 What's wrong with using backticks in a void context?

Strictly speaking, nothing. Stylistically speaking, it's not a good way to write maintainable code. Perl has several operators for running external commands. Backticks are one; they collect the output from the command for use in your program. The `system` function is another; it doesn't do this.

Writing backticks in your program sends a clear message to the readers of your code that you wanted to collect the output of the command. Why send a clear message that isn't true?

Consider this line:

```
    `cat /etc/termcap`;
```

You forgot to check `$?` to see whether the program even ran correctly. Even if you wrote

```
    print `cat /etc/termcap`;
```

this code could and probably should be written as

```
    system("cat /etc/termcap") == 0
        or die "cat program failed!";
```

which will get the output quickly (as it is generated, instead of only at the end) and also check the return value.
system() also provides direct control over whether shell wildcard processing may take place, whereas backticks do not.

24.1.28 How can I call backticks without shell processing?

This is a bit tricky. You can't simply write the command like this:

```
    @ok = `grep @opts '$search_string' @filenames`;
```

As of Perl 5.8.0, you can use open() with multiple arguments. Just like the list forms of system() and exec(), no shell escapes happen.

```
   open( GREP, "-|", 'grep', @opts, $search_string, @filenames );
   chomp(@ok = <GREP>);
   close GREP;
```

You can also:

```
    my @ok = ();
    if (open(GREP, "-|")) {
        while (<GREP>) {
            chomp;
            push(@ok, $_);
        }
        close GREP;
    } else {
        exec 'grep', @opts, $search_string, @filenames;
    }
```

Just as with system(), no shell escapes happen when you exec() a list. Further examples of this can be found in Safe Pipe Opens in *perlipc*.
Note that if you're use Microsoft, no solution to this vexing issue is even possible. Even if Perl were to emulate fork(), you'd still be stuck, because Microsoft does not have a argc/argv-style API.

24.1.29 Why can't my script read from STDIN after I gave it EOF (ˆD on Unix, ˆZ on MS-DOS)?

Some stdio's set error and eof flags that need clearing. The POSIX module defines clearerr() that you can use. That is the technically correct way to do it. Here are some less reliable workarounds:

1. Try keeping around the seekpointer and go there, like this:

```
    $where = tell(LOG);
    seek(LOG, $where, 0);
```

2. If that doesn't work, try seeking to a different part of the file and then back.
3. If that doesn't work, try seeking to a different part of the file, reading something, and then seeking back.
4. If that doesn't work, give up on your stdio package and use sysread.

24.1.30 How can I convert my shell script to perl?

Learn Perl and rewrite it. Seriously, there's no simple converter. Things that are awkward to do in the shell are easy to do in Perl, and this very awkwardness is what would make a shell->perl converter nigh-on impossible to write. By rewriting it, you'll think about what you're really trying to do, and hopefully will escape the shell's pipeline datastream paradigm, which while convenient for some matters, causes many inefficiencies.

24.1.31 Can I use perl to run a telnet or ftp session?

Try the Net::FTP, TCP::Client, and Net::Telnet modules (available from CPAN). http://www.cpan.org/scripts/netstuff/telnet.emul.shar will also help for emulating the telnet protocol, but Net::Telnet is quite probably easier to use..

If all you want to do is pretend to be telnet but don't need the initial telnet handshaking, then the standard dual-process approach will suffice:

```
use IO::Socket;             # new in 5.004
$handle = IO::Socket::INET->new('www.perl.com:80')
        || die "can't connect to port 80 on www.perl.com: $!";
$handle->autoflush(1);
if (fork()) {               # XXX: undef means failure
    select($handle);
    print while <STDIN>;    # everything from stdin to socket
} else {
    print while <$handle>;  # everything from socket to stdout
}
close $handle;
exit;
```

24.1.32 How can I write expect in Perl?

Once upon a time, there was a library called chat2.pl (part of the standard perl distribution), which never really got finished. If you find it somewhere, *don't use it*. These days, your best bet is to look at the Expect module available from CPAN, which also requires two other modules from CPAN, IO::Pty and IO::Stty.

24.1.33 Is there a way to hide perl's command line from programs such as "ps"?

First of all note that if you're doing this for security reasons (to avoid people seeing passwords, for example) then you should rewrite your program so that critical information is never given as an argument. Hiding the arguments won't make your program completely secure.

To actually alter the visible command line, you can assign to the variable $0 as documented in *perlvar*. This won't work on all operating systems, though. Daemon programs like sendmail place their state there, as in:

```
$0 = "orcus [accepting connections]";
```

24.1.34 I {changed directory, modified my environment} in a perl script. How come the change disappeared when I exited the script? How do I get my changes to be visible?

Unix

In the strictest sense, it can't be done–the script executes as a different process from the shell it was started from. Changes to a process are not reflected in its parent–only in any children created after the change. There is shell magic that may allow you to fake it by eval()ing the script's output in your shell; check out the comp.unix.questions FAQ for details.

24.1.35 How do I close a process's filehandle without waiting for it to complete?

Assuming your system supports such things, just send an appropriate signal to the process (see kill in *perlfunc*). It's common to first send a TERM signal, wait a little bit, and then send a KILL signal to finish it off.

24.1.36 How do I fork a daemon process?

If by daemon process you mean one that's detached (disassociated from its tty), then the following process is reported to work on most Unixish systems. Non-Unix users should check their Your_OS::Process module for other solutions.

- Open /dev/tty and use the TIOCNOTTY ioctl on it. See *tty* for details. Or better yet, you can just use the POSIX::setsid() function, so you don't have to worry about process groups.
- Change directory to /
- Reopen STDIN, STDOUT, and STDERR so they're not connected to the old tty.
- Background yourself like this:

```
    fork && exit;
```

The Proc::Daemon module, available from CPAN, provides a function to perform these actions for you.

24.1.37 How do I find out if I'm running interactively or not?

Good question. Sometimes `-t STDIN` and `-t STDOUT` can give clues, sometimes not.

```
if (-t STDIN && -t STDOUT) {
    print "Now what? ";
}
```

On POSIX systems, you can test whether your own process group matches the current process group of your controlling terminal as follows:

```
use POSIX qw/getpgrp tcgetpgrp/;
open(TTY, "/dev/tty") or die $!;
$tpgrp = tcgetpgrp(fileno(*TTY));
$pgrp = getpgrp();
if ($tpgrp == $pgrp) {
    print "foreground\n";
} else {
    print "background\n";
}
```

24.1.38 How do I timeout a slow event?

Use the alarm() function, probably in conjunction with a signal handler, as documented in Signals in *perlipc* and the section on "Signals" in the Camel. You may instead use the more flexible Sys::AlarmCall module available from CPAN.

The alarm() function is not implemented on all versions of Windows. Check the documentation for your specific version of Perl.

24.1.39 How do I set CPU limits?

Use the BSD::Resource module from CPAN.

24.1.40 How do I avoid zombies on a Unix system?

Use the reaper code from Signals in *perlipc* to call wait() when a SIGCHLD is received, or else use the double-fork technique described in How do I start a process in the background? in *perlfaq8*.

24.1.41 How do I use an SQL database?

The DBI module provides an abstract interface to most database servers and types, including Oracle, DB2, Sybase, mysql, Postgresql, ODBC, and flat files. The DBI module accesses each database type through a database driver, or DBD. You can see a complete list of available drivers on CPAN: http://www.cpan.org/modules/by-module/DBD/ . You can read more about DBI on http://dbi.perl.org .

Other modules provide more specific access: Win32::ODBC, Alzabo, iodbc, and others found on CPAN Search: http://search.cpan.org .

24.1.42 How do I make a system() exit on control-C?

You can't. You need to imitate the system() call (see *perlipc* for sample code) and then have a signal handler for the INT signal that passes the signal on to the subprocess. Or you can check for it:

```
$rc = system($cmd);
if ($rc & 127) { die "signal death" }
```

24.1.43 How do I open a file without blocking?

If you're lucky enough to be using a system that supports non-blocking reads (most Unixish systems do), you need only to use the O_NDELAY or O_NONBLOCK flag from the Fcntl module in conjunction with sysopen():

```
use Fcntl;
sysopen(FH, "/foo/somefile", O_WRONLY|O_NDELAY|O_CREAT, 0644)
    or die "can't open /foo/somefile: $!":
```

24.1.44 How do I install a module from CPAN?

The easiest way is to have a module also named CPAN do it for you. This module comes with perl version 5.004 and later.

```
$ perl -MCPAN -e shell

cpan shell -- CPAN exploration and modules installation (v1.59_54)
ReadLine support enabled

cpan> install Some::Module
```

To manually install the CPAN module, or any well-behaved CPAN module for that matter, follow these steps:

1. Unpack the source into a temporary area.
2. `perl Makefile.PL`
3. `make`
4. `make test`
5. `make install`

If your version of perl is compiled without dynamic loading, then you just need to replace step 3 (**make**) with **make perl** and you will get a new *perl* binary with your extension linked in.

See *ExtUtils::MakeMaker* for more details on building extensions. See also the next question, "What's the difference between require and use?".

24.1.45 What's the difference between require and use?

Perl offers several different ways to include code from one file into another. Here are the deltas between the various inclusion constructs:

```
1)  do $file is like eval `cat $file`, except the former
    1.1: searches @INC and updates %INC.
    1.2: bequeaths an *unrelated* lexical scope on the eval'ed code.

2)  require $file is like do $file, except the former
    2.1: checks for redundant loading, skipping already loaded files.
    2.2: raises an exception on failure to find, compile, or execute $file.

3)  require Module is like require "Module.pm", except the former
    3.1: translates each "::" into your system's directory separator.
    3.2: primes the parser to disambiguate class Module as an indirect object.

4)  use Module is like require Module, except the former
    4.1: loads the module at compile time, not run-time.
    4.2: imports symbols and semantics from that package to the current one.
```

In general, you usually want use and a proper Perl module.

24.1.46 How do I keep my own module/library directory?

When you build modules, use the PREFIX and LIB options when generating Makefiles:

```
perl Makefile.PL PREFIX=/mydir/perl LIB=/mydir/perl/lib
```

then either set the PERL5LIB environment variable before you run scripts that use the modules/libraries (see *perlrun*) or say

```
use lib '/mydir/perl/lib';
```

This is almost the same as

```
BEGIN {
    unshift(@INC, '/mydir/perl/lib');
}
```

except that the lib module checks for machine-dependent subdirectories. See Perl's *lib* for more information.

24.1.47 How do I add the directory my program lives in to the module/library search path?

```
use FindBin;
use lib "$FindBin::Bin";
use your_own_modules;
```

24.1.48 How do I add a directory to my include path (@INC) at runtime?

Here are the suggested ways of modifying your include path:

```
the PERLLIB environment variable
the PERL5LIB environment variable
the perl -Idir command line flag
the use lib pragma, as in
    use lib "$ENV{HOME}/myown_perllib";
```

The latter is particularly useful because it knows about machine dependent architectures. The lib.pm pragmatic module was first included with the 5.002 release of Perl.

24.1.49 What is socket.ph and where do I get it?

It's a perl4-style file defining values for system networking constants. Sometimes it is built using h2ph when Perl is installed, but other times it is not. Modern programs `use Socket;` instead.

24.2 AUTHOR AND COPYRIGHT

Chapter 25

perlfaq9

Networking ($Revision: 1.15 $, $Date: 2003/01/31 17:36:57 $)

25.1 DESCRIPTION

This section deals with questions related to networking, the internet, and a few on the web.

25.1.1 What is the correct form of response from a CGI script?

(Alan Flavell <flavell+www@a5.ph.gla.ac.uk> answers...)

The Common Gateway Interface (CGI) specifies a software interface between a program ("CGI script") and a web server (HTTPD). It is not specific to Perl, and has its own FAQs and tutorials, and usenet group, comp.infosystems.www.authoring.cgi

The original CGI specification is at: http://hoohoo.ncsa.uiuc.edu/cgi/

Current best-practice RFC draft at: http://CGI-Spec.Golux.Com/

Other relevant documentation listed in: http://www.perl.org/CGI_MetaFAQ.html

These Perl FAQs very selectively cover some CGI issues. However, Perl programmers are strongly advised to use the CGI.pm module, to take care of the details for them.

The similarity between CGI response headers (defined in the CGI specification) and HTTP response headers (defined in the HTTP specification, RFC2616) is intentional, but can sometimes be confusing.

The CGI specification defines two kinds of script: the "Parsed Header" script, and the "Non Parsed Header" (NPH) script. Check your server documentation to see what it supports. "Parsed Header" scripts are simpler in various respects. The CGI specification allows any of the usual newline representations in the CGI response (it's the server's job to create an accurate HTTP response based on it). So "\n" written in text mode is technically correct, and recommended. NPH scripts are more tricky: they must put out a complete and accurate set of HTTP transaction response headers; the HTTP specification calls for records to be terminated with carriage-return and line-feed, i.e ASCII \015\012 written in binary mode.

Using CGI.pm gives excellent platform independence, including EBCDIC systems. CGI.pm selects an appropriate newline representation ($CGI::CRLF) and sets binmode as appropriate.

25.1.2 My CGI script runs from the command line but not the browser. (500 Server Error)

Several things could be wrong. You can go through the "Troubleshooting Perl CGI scripts" guide at

```
        http://www.perl.org/troubleshooting_CGI.html
```

If, after that, you can demonstrate that you've read the FAQs and that your problem isn't something simple that can be easily answered, you'll probably receive a courteous and useful reply to your question if you post it on comp.infosystems.www.authoring.cgi (if it's something to do with HTTP or the CGI protocols). Questions that appear to be Perl questions but are really CGI ones that are posted to comp.lang.perl.misc are not so well received.

The useful FAQs, related documents, and troubleshooting guides are listed in the CGI Meta FAQ:

```
        http://www.perl.org/CGI_MetaFAQ.html
```

25.1.3 How can I get better error messages from a CGI program?

Use the CGI::Carp module. It replaces `warn` and `die`, plus the normal Carp modules `carp`, `croak`, and `confess` functions with more verbose and safer versions. It still sends them to the normal server error log.

```
    use CGI::Carp;
    warn "This is a complaint";
    die "But this one is serious";
```

The following use of CGI::Carp also redirects errors to a file of your choice, placed in a BEGIN block to catch compile-time warnings as well:

```
    BEGIN {
        use CGI::Carp qw(carpout);
        open(LOG, ">>/var/local/cgi-logs/mycgi-log")
            or die "Unable to append to mycgi-log: $!\n";
        carpout(*LOG);
    }
```

You can even arrange for fatal errors to go back to the client browser, which is nice for your own debugging, but might confuse the end user.

```
    use CGI::Carp qw(fatalsToBrowser);
    die "Bad error here";
```

Even if the error happens before you get the HTTP header out, the module will try to take care of this to avoid the dreaded server 500 errors. Normal warnings still go out to the server error log (or wherever you've sent them with `carpout`) with the application name and date stamp prepended.

25.1.4 How do I remove HTML from a string?

The most correct way (albeit not the fastest) is to use HTML::Parser from CPAN. Another mostly correct way is to use HTML::FormatText which not only removes HTML but also attempts to do a little simple formatting of the resulting plain text.

Many folks attempt a simple-minded regular expression approach, like `s/<.*?>//g`, but that fails in many cases because the tags may continue over line breaks, they may contain quoted angle-brackets, or HTML comment may be present. Plus, folks forget to convert entities–like `&lt;` for example.

Here's one "simple-minded" approach, that works for most files:

```
    #!/usr/bin/perl -p0777
    s/<(?:[^>'"]*|(['"]).*?\1)*>//gs
```

If you want a more complete solution, see the 3-stage striphtml program in http://www.cpan.org/authors/Tom_Christiansen/scripts/striphtml.gz .

Here are some tricky cases that you should think about when picking a solution:

```
    <IMG SRC = "foo.gif" ALT = "A > B">

    <IMG SRC = "foo.gif"
         ALT = "A > B">

    <!-- <A comment> -->

    <script>if (a<b && a>c)</script>

    <# Just data #>

    <![INCLUDE CDATA [ >>>>>>>>>>>> ]]>
```

If HTML comments include other tags, those solutions would also break on text like this:

```
    <!-- This section commented out.
        <B>You can't see me!</B>
    -->
```

25.1.5 How do I extract URLs?

You can easily extract all sorts of URLs from HTML with `HTML::SimpleLinkExtor` which handles anchors, images, objects, frames, and many other tags that can contain a URL. If you need anything more complex, you can create your own subclass of `HTML::LinkExtor` or `HTML::Parser`. You might even use `HTML::SimpleLinkExtor` as an example for something specifically suited to your needs.

You can use URI::Find to extract URLs from an arbitrary text document.

Less complete solutions involving regular expressions can save you a lot of processing time if you know that the input is simple. One solution from Tom Christiansen runs 100 times faster than most module based approaches but only extracts URLs from anchors where the first attribute is HREF and there are no other attributes.

```
        #!/usr/bin/perl -n00
        # qxurl - tchrist@perl.com
        print "$2\n" while m{
            < \s*
              A \s+ HREF \s* = \s* (["']) (.*?) \1
            \s* >
        }gsix;
```

25.1.6 How do I download a file from the user's machine? How do I open a file on another machine?

In this case, download means to use the file upload feature of HTML forms. You allow the web surfer to specify a file to send to your web server. To you it looks like a download, and to the user it looks like an upload. No matter what you call it, you do it with what's known as **multipart/form-data** encoding. The CGI.pm module (which comes with Perl as part of the Standard Library) supports this in the start_multipart_form() method, which isn't the same as the startform() method.

See the section in the CGI.pm documentation on file uploads for code examples and details.

25.1.7 How do I make a pop-up menu in HTML?

Use the <**SELECT**> and <**OPTION**> tags. The CGI.pm module (available from CPAN) supports this widget, as well as many others, including some that it cleverly synthesizes on its own.

25.1.8 How do I fetch an HTML file?

One approach, if you have the lynx text-based HTML browser installed on your system, is this:

```
$html_code = ‘lynx -source $url‘;
$text_data = ‘lynx -dump $url‘;
```

The libwww-perl (LWP) modules from CPAN provide a more powerful way to do this. They don’t require lynx, but like lynx, can still work through proxies:

```
# simplest version
use LWP::Simple;
$content = get($URL);

# or print HTML from a URL
use LWP::Simple;
getprint "http://www.linpro.no/lwp/";

# or print ASCII from HTML from a URL
# also need HTML-Tree package from CPAN
use LWP::Simple;
use HTML::Parser;
use HTML::FormatText;
my ($html, $ascii);
$html = get("http://www.perl.com/");
defined $html
    or die "Can’t fetch HTML from http://www.perl.com/";
$ascii = HTML::FormatText->new->format(parse_html($html));
print $ascii;
```

25.1.9 How do I automate an HTML form submission?

If you’re submitting values using the GET method, create a URL and encode the form using the `query_form` method:

```
use LWP::Simple;
use URI::URL;

my $url = url(’http://www.perl.com/cgi-bin/cpan_mod’);
$url->query_form(module => ’DB_File’, readme => 1);
$content = get($url);
```

If you’re using the POST method, create your own user agent and encode the content appropriately.

```
use HTTP::Request::Common qw(POST);
use LWP::UserAgent;

$ua = LWP::UserAgent->new();
my $req = POST ’http://www.perl.com/cgi-bin/cpan_mod’,
               [ module => ’DB_File’, readme => 1 ];
$content = $ua->request($req)->as_string;
```

25.1.10 How do I decode or create those %-encodings on the web?

If you are writing a CGI script, you should be using the CGI.pm module that comes with perl, or some other equivalent module. The CGI module automatically decodes queries for you, and provides an escape() function to handle encoding.

The best source of detailed information on URI encoding is RFC 2396. Basically, the following substitutions do it:

```
s/([^\w()'*~!.-])/sprintf '%%%02x', ord $1/eg;   # encode

s/%([A-Fa-f\d]{2})/chr hex $1/eg;            # decode
```

However, you should only apply them to individual URI components, not the entire URI, otherwise you'll lose information and generally mess things up. If that didn't explain it, don't worry. Just go read section 2 of the RFC, it's probably the best explanation there is.

RFC 2396 also contains a lot of other useful information, including a regexp for breaking any arbitrary URI into components (Appendix B).

25.1.11 How do I redirect to another page?

Specify the complete URL of the destination (even if it is on the same server). This is one of the two different kinds of CGI "Location:" responses which are defined in the CGI specification for a Parsed Headers script. The other kind (an absolute URLpath) is resolved internally to the server without any HTTP redirection. The CGI specifications do not allow relative URLs in either case.

Use of CGI.pm is strongly recommended. This example shows redirection with a complete URL. This redirection is handled by the web browser.

```
use CGI qw/:standard/;

my $url = 'http://www.cpan.org/';
print redirect($url);
```

This example shows a redirection with an absolute URLpath. This redirection is handled by the local web server.

```
my $url = '/CPAN/index.html';
print redirect($url);
```

But if coded directly, it could be as follows (the final "\n" is shown separately, for clarity), using either a complete URL or an absolute URLpath.

```
print "Location: $url\n";   # CGI response header
print "\n";                 # end of headers
```

25.1.12 How do I put a password on my web pages?

To enable authentication for your web server, you need to configure your web server. The configuration is different for different sorts of web servers—apache does it differently from iPlanet which does it differently from IIS. Check your web server documentation for the details for your particular server.

25.1.13 How do I edit my .htpasswd and .htgroup files with Perl?

The HTTPD::UserAdmin and HTTPD::GroupAdmin modules provide a consistent OO interface to these files, regardless of how they're stored. Databases may be text, dbm, Berkeley DB or any database with a DBI compatible driver. HTTPD::UserAdmin supports files used by the 'Basic' and 'Digest' authentication schemes. Here's an example:

```
use HTTPD::UserAdmin ();
HTTPD::UserAdmin
      ->new(DB => "/foo/.htpasswd")
      ->add($username => $password);
```

25.1.14 How do I make sure users can't enter values into a form that cause my CGI script to do bad things?

See the security references listed in the CGI Meta FAQ

```
        http://www.perl.org/CGI_MetaFAQ.html
```

25.1.15 How do I parse a mail header?

For a quick-and-dirty solution, try this solution derived from split in *perlfunc*:

```
    $/ = '';
    $header = <MSG>;
    $header =~ s/\n\s+/ /g;      # merge continuation lines
    %head = ( UNIX_FROM_LINE, split /^([-\w]+):\s*/m, $header );
```

That solution doesn't do well if, for example, you're trying to maintain all the Received lines. A more complete approach is to use the Mail::Header module from CPAN (part of the MailTools package).

25.1.16 How do I decode a CGI form?

You use a standard module, probably CGI.pm. Under no circumstances should you attempt to do so by hand!

You'll see a lot of CGI programs that blindly read from STDIN the number of bytes equal to CONTENT_LENGTH for POSTs, or grab QUERY_STRING for decoding GETs. These programs are very poorly written. They only work sometimes. They typically forget to check the return value of the read() system call, which is a cardinal sin. They don't handle HEAD requests. They don't handle multipart forms used for file uploads. They don't deal with GET/POST combinations where query fields are in more than one place. They don't deal with keywords in the query string.

In short, they're bad hacks. Resist them at all costs. Please do not be tempted to reinvent the wheel. Instead, use the CGI.pm or CGI_Lite.pm (available from CPAN), or if you're trapped in the module-free land of perl1 .. perl4, you might look into cgi-lib.pl (available from http://cgi-lib.stanford.edu/cgi-lib/).

Make sure you know whether to use a GET or a POST in your form. GETs should only be used for something that doesn't update the server. Otherwise you can get mangled databases and repeated feedback mail messages. The fancy word for this is "idempotency". This simply means that there should be no difference between making a GET request for a particular URL once or multiple times. This is because the HTTP protocol definition says that a GET request may be cached by the browser, or server, or an intervening proxy. POST requests cannot be cached, because each request is independent and matters. Typically, POST requests change or depend on state on the server (query or update a database, send mail, or purchase a computer).

25.1.17 How do I check a valid mail address?

You can't, at least, not in real time. Bummer, eh?

Without sending mail to the address and seeing whether there's a human on the other hand to answer you, you cannot determine whether a mail address is valid. Even if you apply the mail header standard, you can have problems, because there are deliverable addresses that aren't RFC-822 (the mail header standard) compliant, and addresses that aren't deliverable which are compliant.

You can use the Email::Valid or RFC::RFC822::Address which check the format of the address, although they cannot actually tell you if it is a deliverable address (i.e. that mail to the address will not bounce). Modules like Mail::CheckUser and Mail::EXPN try to interact with the domain name system or particular mail servers to learn even more, but their methods do not work everywhere—especially for security conscious administrators.

Many are tempted to try to eliminate many frequently-invalid mail addresses with a simple regex, such as `/^[\w.-]+\@(?:[\w-]+\.)+\w+$/`. It's a very bad idea. However, this also throws out many valid ones, and says nothing about potential deliverability, so it is not suggested. Instead, see http://www.cpan.org/authors/Tom_Christiansen/scripts/ckaddr.gz , which actually checks against the full RFC spec (except for nested comments), looks for addresses you may not wish to accept mail to (say, Bill Clinton or your postmaster), and then makes sure that the hostname given can be looked up in the DNS MX records. It's not fast, but it works for what it tries to do.

Our best advice for verifying a person's mail address is to have them enter their address twice, just as you normally do to change a password. This usually weeds out typos. If both versions match, send mail to that address with a personal message that looks somewhat like:

```
    Dear someuser@host.com,

    Please confirm the mail address you gave us Wed May  6 09:38:41
    MDT 1998 by replying to this message.  Include the string
    "Rumpelstiltskin" in that reply, but spelled in reverse; that is,
    start with "Nik...".  Once this is done, your confirmed address will
    be entered into our records.
```

If you get the message back and they've followed your directions, you can be reasonably assured that it's real.

A related strategy that's less open to forgery is to give them a PIN (personal ID number). Record the address and PIN (best that it be a random one) for later processing. In the mail you send, ask them to include the PIN in their reply. But if it bounces, or the message is included via a "vacation" script, it'll be there anyway. So it's best to ask them to mail back a slight alteration of the PIN, such as with the characters reversed, one added or subtracted to each digit, etc.

25.1.18 How do I decode a MIME/BASE64 string?

The MIME-Base64 package (available from CPAN) handles this as well as the MIME/QP encoding. Decoding BASE64 becomes as simple as:

```
    use MIME::Base64;
    $decoded = decode_base64($encoded);
```

The MIME-Tools package (available from CPAN) supports extraction with decoding of BASE64 encoded attachments and content directly from email messages.

If the string to decode is short (less than 84 bytes long) a more direct approach is to use the unpack() function's "u" format after minor transliterations:

```
    tr#A-Za-z0-9+/##cd;                   # remove non-base64 chars
    tr#A-Za-z0-9+/# -_#;                  # convert to uuencoded format
    $len = pack("c", 32 + 0.75*length);   # compute length byte
    print unpack("u", $len . $_);         # uudecode and print
```

25.1.19 How do I return the user's mail address?

On systems that support getpwuid, the $< variable, and the Sys::Hostname module (which is part of the standard perl distribution), you can probably try using something like this:

```
    use Sys::Hostname;
    $address = sprintf('%s@%s', scalar getpwuid($<), hostname);
```

Company policies on mail address can mean that this generates addresses that the company's mail system will not accept, so you should ask for users' mail addresses when this matters. Furthermore, not all systems on which Perl runs are so forthcoming with this information as is Unix.

The Mail::Util module from CPAN (part of the MailTools package) provides a mailaddress() function that tries to guess the mail address of the user. It makes a more intelligent guess than the code above, using information given when the module was installed, but it could still be incorrect. Again, the best way is often just to ask the user.

25.1.20 How do I send mail?

Use the `sendmail` program directly:

```
    open(SENDMAIL, "|/usr/lib/sendmail -oi -t -odq")
                        or die "Can't fork for sendmail: $!\n";
    print SENDMAIL <<"EOF";
    From: User Originating Mail <me\@host>
    To: Final Destination <you\@otherhost>
    Subject: A relevant subject line

    Body of the message goes here after the blank line
    in as many lines as you like.
    EOF
    close(SENDMAIL)     or warn "sendmail didn't close nicely";
```

The **-oi** option prevents sendmail from interpreting a line consisting of a single dot as "end of message". The **-t** option says to use the headers to decide who to send the message to, and **-odq** says to put the message into the queue. This last option means your message won't be immediately delivered, so leave it out if you want immediate delivery.

Alternate, less convenient approaches include calling mail (sometimes called mailx) directly or simply opening up port 25 have having an intimate conversation between just you and the remote SMTP daemon, probably sendmail.

Or you might be able use the CPAN module Mail::Mailer:

```
    use Mail::Mailer;

    $mailer = Mail::Mailer->new();
    $mailer->open({ From    => $from_address,
                    To      => $to_address,
                    Subject => $subject,
                  })
        or die "Can't open: $!\n";
    print $mailer $body;
    $mailer->close();
```

The Mail::Internet module uses Net::SMTP which is less Unix-centric than Mail::Mailer, but less reliable. Avoid raw SMTP commands. There are many reasons to use a mail transport agent like sendmail. These include queuing, MX records, and security.

25.1.21 How do I use MIME to make an attachment to a mail message?

This answer is extracted directly from the MIME::Lite documentation. Create a multipart message (i.e., one with attachments).

```
    use MIME::Lite;

    ### Create a new multipart message:
    $msg = MIME::Lite->new(
                 From    =>'me@myhost.com',
                 To      =>'you@yourhost.com',
                 Cc      =>'some@other.com, some@more.com',
                 Subject =>'A message with 2 parts...',
                 Type    =>'multipart/mixed'
                 );
    ### Add parts (each "attach" has same arguments as "new"):
    $msg->attach(Type     =>'TEXT',
                 Data     =>"Here's the GIF file you wanted"
                 );
    $msg->attach(Type     =>'image/gif',
                 Path     =>'aaa000123.gif',
                 Filename =>'logo.gif'
                 );

    $text = $msg->as_string;
```

MIME::Lite also includes a method for sending these things.

```
    $msg->send;
```

This defaults to using *sendmail* but can be customized to use SMTP via *Net::SMTP*.

25.1.22 How do I read mail?

While you could use the Mail::Folder module from CPAN (part of the MailFolder package) or the Mail::Internet module from CPAN (part of the MailTools package), often a module is overkill. Here's a mail sorter.

```
    #!/usr/bin/perl

    my(@msgs, @sub);
    my $msgno = -1;
    $/ = '';                    # paragraph reads
    while (<>) {
        if (/^From /m) {
            /^Subject:\s*(?:Re:\s*)*(.*)/mi;
            $sub[++$msgno] = lc($1) || '';
        }
        $msgs[$msgno] .= $_;
    }
    for my $i (sort { $sub[$a] cmp $sub[$b] || $a <=> $b } (0 .. $#msgs)) {
        print $msgs[$i];
    }
```

Or more succinctly,

```
    #!/usr/bin/perl -n00
    # bysub2 - awkish sort-by-subject
    BEGIN { $msgno = -1 }
    $sub[++$msgno] = (/^Subject:\s*(?:Re:\s*)*(.*)/mi)[0] if /^From/m;
    $msg[$msgno] .= $_;
    END { print @msg[ sort { $sub[$a] cmp $sub[$b] || $a <=> $b } (0 .. $#msg) ] }
```

25.1.23 How do I find out my hostname/domainname/IP address?

The normal way to find your own hostname is to call the `` `hostname` `` program. While sometimes expedient, this has some problems, such as not knowing whether you've got the canonical name or not. It's one of those tradeoffs of convenience versus portability.

The Sys::Hostname module (part of the standard perl distribution) will give you the hostname after which you can find out the IP address (assuming you have working DNS) with a gethostbyname() call.

```
    use Socket;
    use Sys::Hostname;
    my $host = hostname();
    my $addr = inet_ntoa(scalar gethostbyname($host || 'localhost'));
```

Probably the simplest way to learn your DNS domain name is to grok it out of /etc/resolv.conf, at least under Unix. Of course, this assumes several things about your resolv.conf configuration, including that it exists.

(We still need a good DNS domain name-learning method for non-Unix systems.)

25.1.24 How do I fetch a news article or the active newsgroups?

Use the Net::NNTP or News::NNTPClient modules, both available from CPAN. This can make tasks like fetching the newsgroup list as simple as

```
    perl -MNews::NNTPClient
      -e 'print News::NNTPClient->new->list("newsgroups")'
```

25.1.25 How do I fetch/put an FTP file?

LWP::Simple (available from CPAN) can fetch but not put. Net::FTP (also available from CPAN) is more complex but can put as well as fetch.

25.1.26 How can I do RPC in Perl?

A DCE::RPC module is being developed (but is not yet available) and will be released as part of the DCE-Perl package (available from CPAN). The rpcgen suite, available from CPAN/authors/id/JAKE/, is an RPC stub generator and includes an RPC::ONC module.

25.2 AUTHOR AND COPYRIGHT

Part III

Reference Manual

Chapter 26

perlsyn

Perl syntax

26.1 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 mostly serves to separate tokens, unlike languages like Python where it is an important part of the syntax.

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 *perltrap* for information about how they differ.

26.1.1 Declarations

The only things you need to declare in Perl are report formats and subroutines (and sometimes not even subroutines). A 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:

```
    my $a;
    if ($a) {}
```

are exempt from warnings (because they care about truth rather than definedness). Operators such as `++`, `-`, `+=`, `-=`, and `.=`, that operate on undefined left values such as:

```
    my $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. Typically all the declarations are put at the beginning or the end of the script. However, if you're using lexically-scoped private variables created with `my()`, 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";
```

Note that myname() functions as a list operator, not as a unary operator; so be careful to use `or` instead of || in this case. However, if you were to declare the subroutine as `sub myname ($)`, then `myname` would function as a unary operator, so either `or` or || would work.

Subroutines declarations can also be loaded up with the `require` statement or both loaded and imported into your namespace with a `use` statement. See *perlmod* 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.

26.1.2 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.

26.1.3 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. (A semicolon is still encouraged if the block takes up more than one line, because you may eventually add another line.) Note that there are some operators like `eval {}` and `do {}` that look like compound statements, but aren't (they're just TERMs in an expression), and thus need an explicit termination if used as the last item in a statement.

26.1.4 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.

26.1.5 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
    foreach LIST
```

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 (i.e., if the condition is false).

```
    print "Basset hounds got long ears" if length $ear >= 10;
    go_outside() and play() unless $is_raining;
```

The `foreach` 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" foreach 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 deprecated `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 $line  eq ".\n";
```

See `do` in *perlfunc*. 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` statement modified with a statement modifier conditional or loop construct (e.g. `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.

26.1.6 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 ... else BLOCK
LABEL while (EXPR) BLOCK
LABEL while (EXPR) BLOCK continue BLOCK
LABEL for (EXPR; EXPR; EXPR) BLOCK
LABEL foreach VAR (LIST) BLOCK
LABEL foreach VAR (LIST) BLOCK continue BLOCK
LABEL BLOCK continue BLOCK
```

Note that, unlike C and Pascal, these are 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) or die "Can't open $FOO: $!";     # FOO or bust!
open(FOO) ? 'hi mom' : 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.

The `while` statement executes the block as long as the expression is true (does not evaluate to the null string `""` or `0` or `"0"`). 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.

26.1.7 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 short-hand 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 `?pat?` one-time matches:

```
    # inspired by :1,$g/fred/s//WILMA/
    while (<>) {
        ?(fred)?    && s//WILMA $1 WILMA/;
        ?(barney)?  && s//BETTY $1 BETTY/;
        ?(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.

The 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 §26.1.10.

The form `while/if BLOCK BLOCK`, available in Perl 4, is no longer available. Replace any occurrence of `if BLOCK` by `if (do BLOCK)`.

26.1.8 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).

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
}
```

26.1.9 Foreach Loops

The `foreach` loop iterates over a normal list value and sets the 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 localisation occurs *only* in a `foreach` loop.

The `foreach` keyword is actually a synonym for the `for` keyword, so you can use `foreach` for readability or `for` for brevity. (Or because the Bourne shell is more familiar to you than *csh*, so writing `for` comes more naturally.) 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.

Examples:

```
for (@ary) { s/foo/bar/ }

for my $elem (@elements) {
    $elem *= 2;
}
```

```
    for $count (10,9,8,7,6,5,4,3,2,1,'BOOM') {
        print $count, "\n"; sleep(1);
    }

    for (1..15) { print "Merry Christmas\n"; }

    foreach $item (split(/:[\\\n:]*/, $ENV{TERMCAP})) {
        print "Item: $item\n";
    }
```

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.

26.1.10 Basic BLOCKs and Switch Statements

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 is particularly nice for doing case structures.

```
    SWITCH: {
        if (/^abc/) { $abc = 1; last SWITCH; }
        if (/^def/) { $def = 1; last SWITCH; }
        if (/^xyz/) { $xyz = 1; last SWITCH; }
        $nothing = 1;
    }
```

There is no official `switch` statement in Perl, because there are already several ways to write the equivalent.

However, starting from Perl 5.8 to get switch and case one can use the Switch extension and say:

```
        use Switch;
```

after which one has switch and case. It is not as fast as it could be because it's not really part of the language (it's done using source filters) but it is available, and it's very flexible.

In addition to the above BLOCK construct, you could write

```
SWITCH: {
    $abc = 1, last SWITCH  if /^abc/;
    $def = 1, last SWITCH  if /^def/;
    $xyz = 1, last SWITCH  if /^xyz/;
    $nothing = 1;
}
```

(That's actually not as strange as it looks once you realize that you can use loop control "operators" within an expression. That's just the binary comma operator in scalar context. See Comma Operator in *perlop*.)

or

```
SWITCH: {
    /^abc/ && do { $abc = 1; last SWITCH; };
    /^def/ && do { $def = 1; last SWITCH; };
    /^xyz/ && do { $xyz = 1; last SWITCH; };
    $nothing = 1;
}
```

or formatted so it stands out more as a "proper" `switch` statement:

```
SWITCH: {
    /^abc/      && do {
                        $abc = 1;
                        last SWITCH;
                   };

    /^def/      && do {
                        $def = 1;
                        last SWITCH;
                   };

    /^xyz/      && do {
                        $xyz = 1;
                        last SWITCH;
                    };
    $nothing = 1;
}
```

or

```
SWITCH: {
    /^abc/ and $abc = 1, last SWITCH;
    /^def/ and $def = 1, last SWITCH;
    /^xyz/ and $xyz = 1, last SWITCH;
    $nothing = 1;
}
```

or even, horrors,

```
    if (/^abc/)
        { $abc = 1 }
    elsif (/^def/)
        { $def = 1 }
    elsif (/^xyz/)
        { $xyz = 1 }
    else
        { $nothing = 1 }
```

A common idiom for a `switch` statement is to use `foreach`'s aliasing to make a temporary assignment to `$_` for convenient matching:

```
    SWITCH: for ($where) {
                /In Card Names/     && do { push @flags, '-e'; last; };
                /Anywhere/          && do { push @flags, '-h'; last; };
                /In Rulings/        && do {                    last; };
                die "unknown value for form variable where: `$where'";
            }
```

Another interesting approach to a switch statement is arrange for a `do` block to return the proper value:

```
    $amode = do {
        if     ($flag & O_RDONLY) { "r" }       # XXX: isn't this 0?
        elsif  ($flag & O_WRONLY) { ($flag & O_APPEND) ? "a" : "w" }
        elsif  ($flag & O_RDWR)   {
            if ($flag & O_CREAT)  { "w+" }
            else                  { ($flag & O_APPEND) ? "a+" : "r+" }
        }
    };
```

Or

```
        print do {
            ($flags & O_WRONLY) ? "write-only"          :
            ($flags & O_RDWR)   ? "read-write"          :
                                  "read-only";
        };
```

Or if you are certain that all the `&&` clauses are true, you can use something like this, which "switches" on the value of the `HTTP_USER_AGENT` environment variable.

```
    #!/usr/bin/perl
    # pick out jargon file page based on browser
    $dir = 'http://www.wins.uva.nl/~mes/jargon';
    for ($ENV{HTTP_USER_AGENT}) {
        $page  =    /Mac/            && 'm/Macintrash.html'
                 || /Win(dows )?NT/  && 'e/evilandrude.html'
                 || /Win|MSIE|WebTV/ && 'm/MicroslothWindows.html'
                 || /Linux/          && 'l/Linux.html'
                 || /HP-UX/          && 'h/HP-SUX.html'
                 || /SunOS/          && 's/ScumOS.html'
                 ||                     'a/AppendixB.html';
    }
    print "Location: $dir/$page\015\012\015\012";
```

That kind of switch statement only works when you know the `&&` clauses will be true. If you don't, the previous `?:` example should be used.

You might also consider writing a hash of subroutine references instead of synthesizing a `switch` statement.

26.1.11 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 `goto`s 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.

26.1.12 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 *perlpod*.

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.

26.1.13 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 the same as for most C preprocessors: it matches the regular expression

```
# example: '# line 42 "new_filename.plx"'
/^\#   \s*
  line \s+ (\d+)   \s*
  (?:\s("?)([^"]+)\2)? \s*
 $/x
```

with `$1` being the line number for the next line, and `$3` being the optional filename (specified with or without quotes).

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.
```

Chapter 27

perldata

Perl data types

27.1 DESCRIPTION

27.1.1 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 *perlref*). 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 Packages in *perlmod* for details). 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 *perlref*.

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 *perlop* and *perlre*). 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 *perlvar*.

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 like 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 *perlref*.

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.)

27.1.2 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 righthand 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 wantarray in *perlfunc* for how you would dynamically discern your function's calling context.

27.1.3 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 TRUE in the Boolean sense if it is not the null string or the number 0 (or its string equivalent, "0"). 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 *perlref*. 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 *perlre*).

```
    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. (It used to do so in Perl 4, but we had to break this to make sure destructors were called when expected.)

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;
```

Version 5 of Perl changed the semantics of `$[`: files that don't set the value of `$[` no longer need to worry about whether another file changed its value. (In other words, use of `$[` is deprecated.) So in general you can assume that

```
    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.

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
```

27.1.4 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
    0b011011            # binary
```

You are allowed to use underscores (underbars) in numeric literals between digits for legibility. 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 Quote and Quote-like Operators in *perlop* 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 hex in *perlfunc* and oct in *perlfunc* for more details.

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.

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, an identifier within such curlies is forced to be a string, as is any simple identifier 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.

Version Strings

Note: Version Strings (v-strings) have been deprecated. They will not be available after Perl 5.8. The marginal benefits of v-strings were greatly outweighed by the potential for Surprise and Confusion.

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 UTF-8 encoded 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. The `$^V` special variable also contains the running Perl interpreter's version in this form. See $^V in *perlvar*. 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.

Special Literals

The special literals __FILE__, __LINE__, and __PACKAGE__ represent the current filename, line number, and package name at that point in your program. 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.

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__ but 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 contents after __DATA__. It is the program's responsibility to `close DATA` when it is done reading from it. For compatibility with older scripts written before __DATA__ was introduced, __END__ behaves like __DATA__ in the toplevel 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.

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. 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'`.

Array Joining Delimiter

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 Quote and Quote-like Operators in *perlop*.

27.1.5 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 *perlref*.

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);
```

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 sort in *perlfunc* for examples of how to arrange for an output ordering.

27.1.6 Subscripts

An array is subscripted by specifying a dollary sign ($), then the name of the array (without the leading @), then the subscript inside square brackets. For example:

```
    @myarray = (5, 50, 500, 5000);
    print "Element Number 2 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";
```

27.1.7 Slices

A common way to access an array or a hash is one scalar element at a time. You can also subscript a list to get a single element from it.

```
    $whoami = $ENV{"USER"};             # one element from the hash
    $parent = $ISA[0];                  # one element from the array
    $dir    = (getpwnam("daemon"))[7];  # likewise, but with list
```

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
    }
```

A slice of an empty list is still an empty list. Thus:

```
    @a = ()[1,0];           # @a has no elements
    @b = (@a)[0,1];         # @b has no elements
    @c = (0,1)[2,3];        # @c has no elements
```

But:

```
    @a = (1)[1,0];          # @a has two elements
    @b = (1,undef)[1,0,2];  # @b has three 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.

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).

27.1.8 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 Symbol Tables in *perlmod* 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 *perlsub* 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 open() in *perlfunc* for an example.

27.2 SEE ALSO

See *perlvar* for a description of Perl's built-in variables and a discussion of legal variable names. See *perlref*, *perlsub*, and Symbol Tables in *perlmod* for more discussion on typeglobs and the `*foo{THING}` syntax.

Chapter 28

perlop

Perl operators and precedence

28.1 DESCRIPTION

28.1.1 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. 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.
    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*.

28.1.2 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 *perlfunc*.

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 Named Unary Operators 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 Quote and Quote-like Operators toward the end of this section, as well as §28.1.30.

28.1.3 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 *perlreftut* and *perlref*.

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 *perlobj*.

28.1.4 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 behaviour. 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.

28.1.5 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.)

28.1.6 Symbolic Unary Operators

Unary "!" performs logical negation, i.e., "not". See also `not` for a lower precedence version of this.

Unary "-" performs arithmetic negation if the operand is numeric. 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 `"-bareword"`.

Unary "˜" performs bitwise negation, i.e., 1's complement. For example, `0666 & ˜027` is 0640. (See also Integer Arithmetic and Bitwise String Operators.) 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.

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 *perlreftut* and *perlref*. 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.

28.1.7 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. Behavior in list context depends on the particular operator. See §28.1.28 for details.

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.

Binary "!˜" is just like "=˜" except the return value is negated in the logical sense.

28.1.8 Multiplicative Operators

Binary "*" multiplies two numbers.

Binary "/" divides two numbers.

Binary "%" computes the modulus of two numbers. Given integer operands `$a` and `$b`: If `$b` is positive, then `$a % $b` is `$a` minus the largest multiple of `$b` that is not greater than `$a`. If `$b` is negative, then `$a % $b` is `$a` minus the smallest multiple of `$b` that is not less than `$a` (i.e. the result will be less than or equal to zero). Note that when `use integer` is in scope, "%" gives you direct access to the modulus 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, it repeats the list. If the right operand is zero or 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
```

28.1.9 Additive Operators

Binary "+" returns the sum of two numbers.

Binary "-" returns the difference of two numbers.

Binary "." concatenates two strings.

28.1.10 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 Integer Arithmetic.)

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 Integer Arithmetic.)

Note that both "<<" and ">>" in Perl are implemented directly using "<<" and ">>" in C. If `use integer` (see Integer Arithmetic) 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.

28.1.11 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 §28.1.2.

28.1.12 Relational Operators

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.

28.1.13 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 NaNs (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 NaNs then NaN is just a string with numeric value 0.

```
    perl -le '$a = NaN; print "No NaN support here" if $a == $a'
    perl -le '$a = NaN; print "NaN support here" if $a != $a'
```

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.

"lt", "le", "ge", "gt" and "cmp" use the collation (sort) order specified by the current locale if `use locale` is in effect. See *perllocale*.

28.1.14 Bitwise And

Binary "&" returns its operands ANDed together bit by bit. (See also Integer Arithmetic and Bitwise String Operators.)

Note that "&" has lower priority than relational operators, so for example the brackets are essential in a test like

```
        print "Even\n" if ($x & 1) == 0;
```

28.1.15 Bitwise Or and Exclusive Or

Binary "|" returns its operands ORed together bit by bit. (See also Integer Arithmetic and Bitwise String Operators.)

Binary "^" returns its operands XORed together bit by bit. (See also Integer Arithmetic and Bitwise String Operators.)

Note that "|" and "^" have lower priority than relational operators, so for example the brackets are essential in a test like

```
        print "false\n" if (8 | 2) != 10;
```

28.1.16 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.

28.1.17 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.

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 more readable alternatives to `&&` and `||` when used for control flow, Perl provides `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);
```

Using "or" for assignment is unlikely to do what you want; see below.

28.1.18 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. 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 till 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 .. $. == 200) ...
    next line if (1 .. /^$/);  # skip header lines, short for
                               #   ... if ($. == 1 .. /^$/);
    s/^/> / if (/^$/ .. eof()); # quote body

    # parse mail messages
    while (<>) {
        $in_header =   1  .. /^$/;
        $in_body   = /^$/ .. eof;
        if ($in_header) {
            # ...
        } else { # in body
            # ...
        }
    } 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.
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);
```

28.1.19 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.

```
    $a = $ok ? $b : $c;  # get a scalar
    @a = $ok ? @b : @c;  # get an array
    $a = $ok ? @b : @c;  # 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):

```
    ($a_or_b ? $a : $b) = $c;
```

Because this operator produces an assignable result, using assignments without parentheses will get you in trouble. For example, this:

```
    $a % 2 ? $a += 10 : $a += 2
```

Really means this:

```
    (($a % 2) ? ($a += 10) : $a) += 2
```

Rather than this:

```
    ($a % 2) ? ($a += 10) : ($a += 2)
```

That should probably be written more simply as:

```
    $a += ($a % 2) ? 10 : 2;
```

28.1.20 Assignment Operators

"=" is the ordinary assignment operator.

Assignment operators work as in C. That is,

```
    $a += 2;
```

is equivalent to

```
    $a = $a + 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.

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 [A-Z] [a-z];
```

Likewise,

```
    ($a += 2) *= 3;
```

is equivalent to

```
    $a += 2;
    $a *= 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.

28.1.21 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.

The => operator is a synonym for the comma, but forces any word to its left to be interpreted as a string (as of 5.001). It is helpful in documenting the correspondence between keys and values in hashes, and other paired elements in lists.

28.1.22 List Operators (Rightward)

On the right side of a list operator, it 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 extra parentheses:

```
    open HANDLE, "filename"
        or die "Can't open: $!\n";
```

See also discussion of list operators in Terms and List Operators (Leftward).

28.1.23 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.

28.1.24 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: i.e., the right expression is evaluated only if the left expression is true.

28.1.25 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: i.e., the right expression is evaluated only if the left expression is false. Due to its precedence, you should probably avoid using this for assignment, only for control flow.

```
    $a = $b or $c;              # bug: this is wrong
    ($a = $b) or $c;            # really means this
    $a = $b || $c;              # 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.

28.1.26 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.

28.1.27 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)
<<EOF		here-doc	yes*

```
        * unless the delimiter is ''.
```

Non-bracketing delimiters use the same character fore and aft, but the four sorts of brackets (round, angle, square, curly) will 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:

```
        $s = q{ if($a eq "}") ... }; # WRONG
```

is a syntax error. The `Text::Balanced` module (from CPAN, and starting from Perl 5.8 part of the standard distribution) 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.

```
    \t          tab             (HT, TAB)
    \n          newline         (NL)
    \r          return          (CR)
    \f          form feed       (FF)
    \b          backspace       (BS)
    \a          alarm (bell)    (BEL)
    \e          escape          (ESC)
    \033        octal char      (ESC)
    \x1b        hex char        (ESC)
    \x{263a}    wide hex char   (SMILEY)
    \c[         control char    (ESC)
    \N{name}    named Unicode character
```

NOTE: Unlike C and other languages, Perl has no \v escape sequence for the vertical tab (VT - ASCII 11).

The following escape sequences are available in constructs that interpolate but not in transliterations.

```
    \l          lowercase next char
    \u          uppercase next char
    \L          lowercase till \E
    \U          uppercase till \E
    \E          end case modification
    \Q          quote non-word characters till \E
```

If `use locale` is in effect, the case map used by `\l`, `\L`, `\u` and `\U` is taken from the current locale. See *perllocale*. If Unicode (for example, `\N{}` or wide hex characters of 0x100 or beyond) is being used, the case map used by `\l`, `\L`, `\u` and `\U` is as defined by Unicode. For documentation of `\N{name}`, see *charnames*.

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 a Mac, these are reversed, and on systems without line terminator, printing `"\n"` may 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 only interpolated if the name is enclosed in braces `@{+}`.

You cannot include a literal `$` or `@` within a `\Q` sequence. An unescaped `$` or `@` interpolates the corresponding variable, while escaping will cause the literal string `\$` to be inserted. You'll need to write something like `m/\Quser\E\@\Qhost/`.

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.

28.1.28 Regexp Quote-Like Operators

Here are the quote-like operators that apply to pattern matching and related activities.

?PATTERN?

This is just like the `/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 `??` patterns local to the current package are reset.

```
while (<>) {
    if (?^$?) {
                        # blank line between header and body
    }
} continue {
    reset if eof;       # clear ?? status for next file
}
```

This usage is vaguely deprecated, which means it just might possibly be removed in some distant future version of Perl, perhaps somewhere around the year 2168.

m/PATTERN/cgimosx

/PATTERN/cgimosx

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 *perlre*. See *perllocale* for discussion of additional considerations that apply when `use locale` is in effect.

Options are:

```
c   Do not reset search position on a failed match when /g is in effect.
g   Match globally, i.e., find all occurrences.
i   Do case-insensitive pattern matching.
```

```
    m   Treat string as multiple lines.
    o   Compile pattern only once.
    s   Treat string as single line.
    x   Use extended regular expressions.
```

If "/" is the delimiter then the initial `m` is optional. With the `m` you can use any pair of non-alphanumeric, non-whitespace characters as delimiters. This is particularly useful for matching path names that contain "/", to avoid LTS (leaning toothpick syndrome). If "?" is the delimiter, then the match-only-once rule of `?PATTERN?` applies. If "'" is the delimiter, no interpolation is performed on the PATTERN.

PATTERN may contain variables, which will be interpolated (and the pattern recompiled) 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.) If you want such a pattern to be compiled only once, add a `/o` after the trailing delimiter. This avoids expensive run-time recompilations, and is useful when the value you are interpolating won't change over the life of the script. However, mentioning `/o` constitutes a promise that you won't change the variables in the pattern. If you change them, Perl won't even notice. See also §**??**.

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 is honoured - 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).

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, i.e., (`$1`, `$2`, `$3`...). (Note that here `$1` etc. are also set, and that this differs from Perl 4's behavior.) 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');
    <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
    }

    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, i.e., 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 pos in *perlfunc*. A failed match normally resets the search position to the beginning of the string, but you can avoid that by adding the `/c` modifier (e.g. `m//gc`). Modifying the target string also resets the search position.

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(), 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
    $/ = "";
    while (defined($paragraph = <>)) {
        while ($paragraph =~ /[a-z]['")]*[.!?]+['")]*\s/g) {
            $sentences++;
        }
    }
    print "$sentences\n";

    # using 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 an older (pre-5.6.0) 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 = new URI::URL "http://www/";   die if $url eq "xXx";
 EOL
 LOOP:
    {
      print(" digits"),         redo LOOP if /\G\d+\b[,.;]?\s*/gc;
      print(" lowercase"),      redo LOOP if /\G[a-z]+\b[,.;]?\s*/gc;
      print(" UPPERCASE"),      redo LOOP if /\G[A-Z]+\b[,.;]?\s*/gc;
      print(" Capitalized"),    redo LOOP if /\G[A-Z][a-z]+\b[,.;]?\s*/gc;
      print(" MiXeD"),          redo LOOP if /\G[A-Za-z]+\b[,.;]?\s*/gc;
      print(" alphanumeric"),   redo LOOP if /\G[A-Za-z0-9]+\b[,.;]?\s*/gc;
      print(" line-noise"),     redo LOOP if /\G[^A-Za-z0-9]+/gc;
      print ". That's all!\n";
    }
```

Here is the output (split into several lines):

```
line-noise lowercase line-noise lowercase UPPERCASE line-noise
UPPERCASE line-noise lowercase line-noise lowercase line-noise
lowercase lowercase line-noise lowercase lowercase line-noise
MiXeD line-noise. That’s all!
```

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
```

qr/STRING/imosx

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/imosx` expression.

For example,

```
    $rex = qr/my.STRING/is;
    s/$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 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 a 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 are:

```
    i   Do case-insensitive pattern matching.
    m   Treat string as multiple lines.
    o   Compile pattern only once.
    s   Treat string as single line.
    x   Use extended regular expressions.
```

See *perlre* for additional information on valid syntax for STRING, and for a detailed look at the semantics of regular expressions.

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");
```

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 *perlsec* 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 (e.g. `;` on many Unix shells; `&` on the Windows NT `cmd` shell).

Beginning with v5.6.0, 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 *perlport*). 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 §28.1.30 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 comma 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.

s/PATTERN/REPLACEMENT/egimosx

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 no string is specified via the =˜ or !˜ operator, the $_ variable is searched and modified. (The string specified with =˜ must be scalar variable, an array element, a hash element, or an assignment to one of those, i.e., an 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 *perlre* for further explanation on these. See *perllocale* for discussion of additional considerations that apply when use locale is in effect.

Options are:

```
e   Evaluate the right side as an expression.
g   Replace globally, i.e., all occurrences.
i   Do case-insensitive pattern matching.
m   Treat string as multiple lines.
o   Compile pattern only once.
s   Treat string as single line.
x   Use extended regular expressions.
```

Any non-alphanumeric, non-whitespace delimiter may replace the slashes. If single quotes are used, no interpretation is done on the replacement string (the /e modifier overrides this, however). Unlike Perl 4, Perl 5 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, e.g., 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 evaled 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

$count = ($paragraph =~ s/Mister\b/Mr./g);  # get change-count

$_ = 'abc123xyz';
s/\d+/$&*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

# 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;

    # 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 white space in $_, expensively

    for ($variable) {           # trim white space 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 in only 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;
```

tr/SEARCHLIST/REPLACEMENTLIST/cds

y/SEARCHLIST/REPLACEMENTLIST/cds

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. (The string specified with =˜ must be a scalar variable, an array element, a hash element, or an assignment to one of those, i.e., 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, e.g., `tr[A-Z][a-z]` or `tr(+\-*/)/ABCD/`.

Note that `tr` does **not** do regular expression character classes such as `\d` or `[:lower:]`. The <tr> operator is not equivalent to the tr(1) utility. If you want to map strings between lower/upper cases, see lc in *perlfunc* and uc in *perlfunc*, and in general consider using the `s` operator if you need regular expressions.

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.
```

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

    $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/;

    tr/a-zA-Z/ /cs;             # change non-alphas to single space

    tr [\200-\377]
       [\000-\177];             # 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. If quoted, the type of quotes you use determines the treatment of the text, just as in regular quoting. An unquoted identifier works like double quotes. There must be no space between the << and the identifier, unless the identifier is 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.

```
   print <<EOF;
The price is $Price.
EOF

   print << "EOF"; # same as above
The price is $Price.
EOF

   print << `EOC`; # execute commands
echo hi there
echo lo there
EOC

   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 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 come on the lines following the final delimiter. So instead of

```
s/this/<<E . 'that'
the other
E
 . 'more '/eg;
```

you have to write

```
s/this/<<E . 'that'
 . 'more '/eg;
the other
E
```

If the terminating identifier is on the last line of the program, you must be sure there is a newline after it; otherwise, Perl will give the warning **Can't find string terminator "END" anywhere before EOF...**.

Additionally, the quoting rules for the identifier are not related 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.

28.1.29 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 five, but these passes are always performed in the same order.

Finding the end

The first pass is finding the end of the quoted construct, whether it be a multicharacter delimiter `"\nEOF\n"` in the `<<EOF` construct, a `/` that terminates a `qq//` construct, a `]` which terminates `qq[]` construct, or a `>` which terminates a fileglob started with `<`.

When searching for single-character non-pairing delimiters, such as `/`, combinations of `\\` and `\/` are skipped. However, when searching for single-character pairing delimiter like `[`, combinations of `\\`, `\]`, and `\[` are all skipped, and nested `[`, `]` are skipped as well. When searching for multicharacter delimiters, nothing is skipped.

For constructs with three-part delimiters (`s///`, `y///`, and `tr///`), the search is repeated once more.

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 `#`.

Removal of backslashes before delimiters

During the second pass, text between the starting and ending delimiters is copied to a safe location, and the \ is removed from combinations consisting of \ and delimiter–or delimiters, meaning both starting and ending delimiters will should these differ. This removal does not happen for multi-character delimiters. Note that the combination \\ is left intact, just as it was.

Starting from this step no information about the delimiters is used in parsing.

Interpolation

The next step is interpolation in the text obtained, which is now delimiter-independent. There are four different cases.

<<'EOF', m'', s''', tr///, y///

No interpolation is performed.

'', q//

The only interpolation is removal of \ from pairs \\.

"", ``, qq//, qx//, <file*glob>

`\Q`, `\U`, `\u`, `\L`, `\l` (possibly paired with `\E`) are converted to corresponding Perl constructs. Thus, `"$foo\Qbaz$bar"` is converted to `$foo . (quotemeta("baz" . $bar))` internally. The other combinations 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 became `"\\\$"`; 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 $b -> {c}"` really means:

```
"a " . $b . " -> {c}";
```

or:

```
"a " . $b -> {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.

?RE?, /RE/, m/RE/, s/RE/foo/,

Processing of `\Q`, `\U`, `\u`, `\L`, `\l`, and interpolation happens (almost) as with `qq//` constructs, but the substitution of \ followed by RE-special chars (including \) is not performed. 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 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.

It is at this step that `\1` is begrudgingly converted to `$1` in the replacement text of `s///` to correct the incorrigible *sed* hackers who haven't picked up the saner idiom yet. A warning is emitted if the `use warnings` pragma or the **-w** command-line flag (that is, the `$^W` variable) was set.

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 backslash-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.

Interpolation of 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 catenation, 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 *perlre*, 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 to 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. Similarly, the terminator of `(?{...})` is found using the same rules as for finding the terminator of a `{}`-delimited construct.

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 Command Switches in *perlrun*.

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`.

28.1.30 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 *perlvar* 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 *perlsec* 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 avoids $_ :

```
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 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 elicit 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 *perlopentut* and open in *perlfunc* 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 readline in *perlfunc*.

The null filehandle <> is special: it can be used to emulate the behavior of **sed** and **awk**. 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.)

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 eof in *perlfunc* 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 (e.g., <$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.

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 (e.g. 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]);
```

28.1.31 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 will precompute the number which that expression represents so that the interpreter won't have to.

28.1.32 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
```

See vec in *perlfunc* for information on how to manipulate individual bits in a bit vector.

28.1.33 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 that it's okay to use integer operations (if it feels like it) 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 only an integer, merely that Perl may use integer operations if it is so inclined. 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 ("&", "|", "^", "~", "<<", and ">>") always produce integral results. (But see also Bitwise String Operators.) 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 twos-complement machines.

28.1.34 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 equality of 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 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.

28.1.35 Bigger Numbers

The standard Math::BigInt and Math::BigFloat modules 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 Math::BigInt;
    $x = Math::BigInt->new('123456789123456789');
    print $x * $x;

    # prints +15241578780673678515622620750190521
```

There are several modules that let you calculate with (bound only by memory and cpu-time) unlimited or fixed precision. There are also some non-standard modules that provide faster implementations via external C libraries.

Here is a short, but incomplete summary:

```
        Math::Fraction          big, unlimited fractions like 9973 / 12967
        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::BigInteger        uses an external C library
        Math::Cephes            uses external Cephes C library (no big numbers)
        Math::Cephes::Fraction  fractions via the Cephes library
        Math::GMP               another one using an external C library
```

Choose wisely.

Chapter 29

perlsub

Perl subroutines

29.1 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 : ATTRS BLOCK          #  with attributes
    sub NAME(PROTO) : ATTRS BLOCK   #  with prototypes and attributes
```

To define an anonymous subroutine at runtime:

```
    $subref = sub BLOCK;                 # no proto
    $subref = sub (PROTO) BLOCK;         # with proto
    $subref = sub : ATTRS BLOCK;         # with attributes
    $subref = sub (PROTO) : ATTRS BLOCK; # with proto 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.
```

29.2 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 `@_`. 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.

The return value of a subroutine is the value of the last expression evaluated by that sub, or the empty list in the case of an empty sub. More explicitly, a `return` statement may be used to exit the 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.

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 §29.2.1 and §29.2.3. To create protected environments for a set of functions in a separate package (and probably a separate file), see Packages in *perlmod*.

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 Pass by Reference 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 *perlref* 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 *Prototypes* below.

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 that do special, pre-defined things include `AUTOLOAD`, `CLONE`, `DESTROY` plus all functions mentioned in *perltie* and *PerlIO::via*.

The `BEGIN`, `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 explicitely. See BEGIN, CHECK, INIT and END in *perlmod*

29.2.1 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 `local`ized 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 *perlref*.

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 Simple statements in *perlsyn* 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 *perlref*. 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
    my $_;              # also illegal (currently)
```

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 Function Templates in *perlref* for something of a work-around to this.

29.2.2 Persistent Private Variables

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 BEGIN, CHECK, INIT and END in *perlmod* about the special triggered code blocks, `BEGIN`, `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.

29.2.3 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 $hash{key} = "val";  # sets a local value for this hash entry
    local ($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.

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.

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 5.9.0, with an error *Modification of a read-only value attempted*, because the $1 variable is magical and read-only :

```
    local $1 = 2;
```

Similarly, but in a way more difficult to spot, the following snippet will die in perl 5.9.0 :

```
    sub f { local $_ = "foo"; print }
    for ($1) {
        # now $_ is aliased to $1, thus is magic and readonly
        f();
    }
```

See next section for an alternative to this situation.

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 behaviour of localising tied arrays or hashes (localising individual elements is still okay). See Localising Tied Arrays and Hashes Is Broken in *perl58delta* for more details.

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.

Notably, if you want to work with a brand new value of the default scalar $_, and avoid the potential problem listed above about $_ previously carrying a magic value, you should use `local *_` instead of `local $_`.

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.

29.2.4 Lvalue subroutines

WARNING: Lvalue subroutines are still experimental and the implementation may change in future versions of Perl.

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 {
        # return $val; this doesn't work, don't say "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 EXPERIMENTAL

They appear to be convenient, but there are several reasons to be circumspect.

You can't use the return keyword, you must pass out the value before falling out of subroutine scope. (see comment in example above). This is usually not a problem, but it disallows an explicit return out of a deeply nested loop, which is sometimes a nice way out.

They violate encapsulation. A normal mutator can check the supplied argument before setting the attribute it is protecting, an lvalue subroutine never gets that chance. Consider;

```
    my $some_array_ref = [];    # protected by mutators ??

    sub set_arr {               # normal mutator
        my $val = shift;
        die("expected array, you supplied ", ref $val)
           unless ref $val eq 'ARRAY';
        $some_array_ref = $val;
    }
    sub set_arr_lv : lvalue {   # lvalue mutator
        $some_array_ref;
    }

    # set_arr_lv cannot stop this !
    set_arr_lv() = { a => 1 };
```

29.2.5 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 Typeglobs and Filehandles in *perldata*.

29.2.6 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 `local`ized 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 `local`ize $_ 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 exists
       grow();                 # really calls shrink()
       move();                 # if move() grow()s, it shrink()s too
   }
   grow();                     # get the real grow() again
   ```

   See Function Templates in *perlref* 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 `local`ize 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. Prior to 5.005, this operation could on occasion misbehave.

29.2.7 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 *perlref*. 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 `locals`) 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;
    }
```

29.2.8 Prototypes

Perl supports a very limited kind of compile-time argument checking using function prototyping. If you declare

```
    sub mypush (\@@)
```

then `mypush()` takes arguments exactly like `push()` does. 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, @array, 0, @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 absolutely must start with that character. 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 also backslash several argument types simultaneously by using the `\[]` notation:

```
    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);
        ...
    }
```

A semicolon separates mandatory arguments from optional arguments. It is redundant before `@` or `%`, which gobble up everything else.

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.

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!

This is all very powerful, of course, and should be used only in moderation to make the world a better place.

29.2.9 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 }
```

Be aware that these will not be inlined; as they contain inner scopes, the constant folding doesn't reduce them to a single constant:

```
    sub foo_set () { if (FLAG_MASK & FLAG_FOO) { 1 } }

    sub baz_val () {
        if (OPT_BAZ) {
            return 23;
        }
        else {
            return 42;
        }
    }
```

If you redefine a subroutine that was eligible for inlining, you'll get a mandatory warning. (You can use this warning to tell whether or not a particular subroutine is considered constant.) The warning is considered severe enough not to be optional 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, such as

```
    sub not_inlined () {
        23 if $];
    }
```

29.2.10 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: you can't take a reference to it, such as the incorrect `\&CORE::open` might appear to produce.

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 $pat = shift;
        my @got;
        local *D;
        if (opendir D, '.') {
            @got = grep /$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 prototype in *perlfunc*).

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 require in *perlfunc*.

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>.

Finally, some built-ins (e.g. exists or grep) can't be overridden.

29.2.11 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...

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 standard Shell module, 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*.

29.2.12 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*.

29.3 SEE ALSO

See Function Templates in *perlref* for more about references and closures. See *perlxs* if you'd like to learn about calling C subroutines from Perl. See *perlembed* if you'd like to learn about calling Perl subroutines from C. See *perlmod* to learn about bundling up your functions in separate files. See *perlmodlib* to learn what library modules come standard on your system. See *perltoot* to learn how to make object method calls.

Chapter 30

perlfunc

Perl builtin functions

30.1 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 *perlop*.) 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 a scalar context to its argument, while a list operator may provide either scalar or list contexts for its arguments. If it does both, the scalar arguments will be first, and the list argument will follow. (Note that there can ever be only 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 the 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. Elements of the LIST should be separated by commas.

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 the 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. And whitespace between the function and left parenthesis doesn't count–so you need to be careful sometimes:

```
    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 a scalar context by returning the undefined value, and in a list context by returning the null 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 it 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 of the same name (like chown(2), fork(2), closedir(2), etc.) all 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 are `wait`, `waitpid`, and `syscall`. System calls also set the special `$!` variable on failure. Other functions do not, except accidentally.

30.1.1 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`, `hex`, `index`, `lc`, `lcfirst`, `length`, `oct`, `ord`, `pack`, `q/STRING/`, `qq/STRING/`, `reverse`, `rindex`, `sprintf`, `substr`, `tr///`, `uc`, `ucfirst`, `y///`

Regular expressions and pattern matching

`m//`, `pos`, `quotemeta`, `s///`, `split`, `study`, `qr//`

Numeric functions

`abs`, `atan2`, `cos`, `exp`, `hex`, `int`, `log`, `oct`, `rand`, `sin`, `sqrt`, `srand`

Functions for real @ARRAYs

`pop`, `push`, `shift`, `splice`, `unshift`

Functions for list data

`grep`, `join`, `map`, `qw/STRING/`, `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`, `rewinddir`, `seek`, `seekdir`, `select`, `syscall`, `sysread`, `sysseek`, `syswrite`, `tell`, `telldir`, `truncate`, `warn`, `write`

Functions for fixed length data or records

`pack`, `read`, `syscall`, `sysread`, `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

`caller`, `continue`, `die`, `do`, `dump`, `eval`, `exit`, `goto`, `last`, `next`, `redo`, `return`, `sub`, `wantarray`

Keywords related to scoping

caller, import, local, my, our, package, use

Miscellaneous functions

defined, dump, eval, formline, local, my, our, reset, scalar, undef, wantarray

Functions for processes and process groups

alarm, exec, fork, getpgrp, getppid, getpriority, kill, pipe, qx/STRING/, setpgrp, setpriority, sleep, system, times, wait, waitpid

Keywords related to perl modules

do, import, no, package, require, use

Keywords related to classes and object-orientedness

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

Functions new in perl5

abs, bless, chomp, chr, exists, formline, glob, import, lc, lcfirst, map, my, no, our, prototype, qx, qw, readline, readpipe, ref, sub*, sysopen, tie, tied, uc, ucfirst, untie, use

* - sub was a keyword in perl4, but in perl5 it is an operator, which can be used in expressions.

Functions obsoleted in perl5

dbmclose, dbmopen

30.1.2 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, getprgp, 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 *perlport* and other available platform-specific documentation.

30.1.3 Alphabetical Listing of Perl Functions

-X FILEHANDLE

-X EXPR

-X

A file test, where X is one of the letters listed below. This unary operator takes one argument, either a filename or a filehandle, 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, or the undefined value if the file doesn't exist. Despite the funny names, precedence is the same as any other named unary operator, and the argument may be parenthesized like any other 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.
    -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 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
        #...
    }
```

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. Such reasons may be for example network filesystem access controls, ACLs (access control lists), read-only filesystems, and unrecognized executable formats.

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 the `use filetest 'access'` the above-mentioned filetests will test whether the permission can (not) be granted using the access() 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. Read the documentation for the `filetest` pragma for more information.

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.

The `-T` and `-B` switches work as follows. The first block or so of the file is examined for odd characters such as strange control codes or characters with the high bit set. If too many strange characters (>30%) are found, it's a `-B` file, otherwise it's a `-T` file. Also, any file containing null in the first block is considered a binary file. 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 a null 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` operators) are 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` will leave values in the stat structure for the symbolic link, not the real file.) (Also, if the stat buffer was filled by a `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 _;
```

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 the accept(2) system call does. Returns the packed address if it succeeded, false otherwise. See the example in Sockets: Client/Server Communication in *perlipc*.

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 $^F in *perlvar*.

alarm SECONDS

alarm

Arranges to have a SIGALRM delivered to this process after the specified number of wallclock seconds have 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, you may 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. The Time::HiRes module (from CPAN, and starting from Perl 5.8 part of the standard distribution) may also prove useful.

It is usually a mistake to intermix `alarm` and `sleep` calls. (`sleep` may be internally implemented in 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 Signals in *perlipc*.

```
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 *perlipc*.

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])  }
```

bind SOCKET,NAME

Binds a network address to a socket, just as the bind system call 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 Sockets: Client/Server Communication in *perlipc*.

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 to always use it when appropriate, and to never use it when it isn't appropriate. Also, people can set their I/O to be by default UTF-8 encoded Unicode, not bytes.

In other words: regardless of platform, use binmode() on binary data, like for example images.

If LAYER is present it is a single string, but may contain multiple directives. The directives alter the behaviour of the file handle. When LAYER is present using binmode on 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) or elsewhere, `:raw` is *not* the simply inverse of `:crlf` – other layers which would affect binary nature of the stream are *also* disabled. See *PerlIO*, *perlrun* and the discussion about the PERLIO environment variable.

The `:bytes`, `:crlf`, and `: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`.

In general, binmode() should be called after open() but before any I/O is done on the filehandle. Calling binmode() will normally flush 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 *open*. 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 will operate on UTF-8 encoded Unicode characters.

The operating system, device drivers, C libraries, and Perl run-time system all work together to let the programmer treat a single character (`\n`) as the line terminator, irrespective of the 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.

Mac OS, all variants of Unix, 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 Mac OS and 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 contains `\cZ`, the I/O subsystem will regard it as the end of the file, unless you use binmode().

binmode() is not only important for readline() and print() operations, but also when using read(), seek(), sysread(), syswrite() and tell() (see *perlport* for more details). See the `$/` and `$\` variables in *perlvar* for how to manually set your input and output line-termination sequences.

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 the function doing the blessing might be inherited by a derived class. See *perltoot* and *perlobj* 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, so to prevent confusion, you may wish to avoid such package names as well. Make sure that CLASSNAME is a true value.

See Perl Modules in *perlmod*.

caller EXPR

caller

Returns the context of the current 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. In list context, returns

```
($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.

```
($package, $filename, $line, $subroutine, $hasargs,
$wantarray, $evaltext, $is_require, $hints, $bitmask) = caller($i);
```

Here $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, $filename 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. The `$hints` and `$bitmask` values are subject to change between versions of Perl, and are not meant for external use.

Furthermore, when called from within the DB package, 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 do, for `N > 1`. In particular, `@DB::args` might have information from the previous time `caller` was called.

chdir EXPR

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 upon success, false otherwise. See the example under `die`.

chmod LIST

Changes the permissions of a list of files. The first element of the list must be the numerical mode, which should probably be an octal number, and which definitely should *not* a string of octal digits: `0644` is okay, `'0644'` is not. Returns the number of files successfully changed. See also oct, 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
```

You can also import the symbolic `S_I*` constants from the Fcntl module:

```
use Fcntl ':mode';

chmod S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH, @executables;
# This is identical to the chmod 0755 of the above example.
```

chomp VARIABLE

chomp(LIST)

chomp

This safer version of chop 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 *perlvar*) 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.

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.

If the `encoding` pragma is in scope then the lengths returned are calculated from the length of `$/` in Unicode characters, which is not always the same as the length of `$/` in the native encoding.

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.

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.

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;
```

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);
```

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. Note that characters from 128 to 255 (inclusive) are by default not encoded in UTF-8 Unicode for backward compatibility reasons (but see *encoding*).

If NUMBER is omitted, uses `$_`.

For the reverse, use ord.

Note that under the `bytes` pragma the NUMBER is masked to the low eight bits.

See *perlunicode* and *encoding* 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 `$_`.

close FILEHANDLE

close

Closes the file or pipe associated with the file handle, returning true only if IO buffers are successfully flushed and closes the system file descriptor. 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` will close it for you. (See `open`.) However, an explicit `close` on an input file resets the line counter (`$.`), while the implicit close done by `open` does not.

If the file handle came from a piped open, `close` will additionally return false if one of the other system calls involved fails, or if the program exits with non-zero status. (If the only problem was that the program exited non-zero, `$!` will be set to `0`.) Closing a pipe also waits for the process executing on the pipe to complete, in case you want to look at the output of the pipe afterwards, and implicitly puts the exit status value of that command into `$?`.

Prematurely closing the read end of a pipe (i.e. before the process writing to it at the other end has closed it) will result in a SIGPIPE being delivered to the writer. 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.

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 as the connect system call 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 Sockets: Client/Server Communication in *perlipc*.

continue BLOCK

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` will 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 semantically equivalent to using an empty one, logically enough. In that case, `next` goes directly back to check the condition at the top of the loop.

cos EXPR

cos

Returns the cosine of EXPR (expressed in radians). If EXPR is omitted, takes 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

Encrypts a 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). This can prove useful for checking the password file for lousy passwords, amongst other things. Only the guys wearing white hats should do this.

Note that crypt is intended to be a one-way function, much like breaking eggs to make an omelette. There is no (known) corresponding decrypt function (in other words, the crypt() is a one-way hash function). As a result, this function isn't all that useful for cryptography. (For that, see your nearby CPAN mirror.)

When verifying an existing encrypted string you should use the encrypted text as the salt (like `crypt($plain, $crypted) eq $crypted`). This allows your code to work with the standard crypt and with more exotic implementations. In other words, do not assume anything about the returned string itself, or how many bytes in the encrypted string 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 the encrypted string 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 own 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 crypt function is unsuitable for encrypting large quantities of data, not least of all because you can't get the information back. Look at the *by-module/Crypt* and *by-module/PGP* directories on your favorite CPAN mirror for a slew of potentially useful modules.

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) 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`.

dbmclose HASH

[This function has been largely superseded by the `untie` function.]

Breaks the binding between a DBM file and a hash.

dbmopen HASH,DBNAME,MASK

[This function has been largely superseded by the `tie` 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`). If your system supports only the older DBM functions, you may perform only one `dbmopen` 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`, which will 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: $!";
```

defined EXPR

defined

Returns a Boolean value telling whether EXPR has a value other than the undefined value `undef`. If EXPR is not present, `$_` will be 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`. Note that a subroutine which 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 *perlsub*.

Use of `defined` on aggregates (hashes and arrays) is deprecated. It used to report whether memory for that aggregate has 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 exists 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 then are 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, despite the fact that it matched "nothing". But it didn't really match nothing–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 you're 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, exists, ref.

delete EXPR

Given an expression that specifies a hash element, array element, hash slice, or array slice, deletes the specified element(s) from the hash or array. In the case of an array, if the array elements happen to be at the end, the size of the array will shrink to the highest element that tests true for exists() (or 0 if no such element exists).

Returns a list with the same number of elements as the number of elements for which deletion was attempted. Each element of that list consists of either the value of the element deleted, or the undefined value. In scalar context, this means that you get the value of the last element deleted (or the undefined value if that element did not exist).

```
    %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 bar baz)}; # @array  is (undef,undef,33)
```

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.

Deleting an array element effectively returns that position of the array to its initial, uninitialized state. Subsequently testing for the same element with exists() will return false. Note that deleting array elements in the middle of an array will not shift the index of the ones after them down–use splice() for that. See exists.

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 of these are slower than just assigning the empty list or undefining %HASH or @ARRAY:

```
    %HASH = ();         # completely empty %HASH
    undef %HASH;        # forget %HASH ever existed

    @ARRAY = ();        # completely empty @ARRAY
    undef @ARRAY;       # forget @ARRAY ever existed
```

Note that the EXPR can be arbitrarily complicated as long as the final operation is a hash element, array element, hash slice, or array slice lookup:

```
    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

Outside an `eval`, prints the value of LIST to `STDERR` and exits with the current value of `$!` (errno). If `$!` is `0`, exits with the value of `($? >> 8)` (backtick ‘command‘ status). If `($? >> 8)` is `0`, exits with `255`. Inside an `eval(),` the error message is stuffed into `$@` and the `eval` is terminated with the undefined value. This makes `die` the way to raise an exception.

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 $/ in *perlvar* and $. in *perlvar*.

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.
```

See also exit(), warn(), and the Carp module.

If LIST 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 LIST is empty and `$@` contains an object reference that has a `PROPAGATE` method, that method will be called with additional file and line number parameters. The return value replaces the value in `$@`. ie. as if `$@ = eval { $@->PROPAGATE(__FILE__, __LINE__) };` were called.

If `$@` is empty then the string `"Died"` is used.

die() can also be called with a reference argument. If this happens to be trapped within an eval(), $@ contains the reference. This behavior permits a more elaborate exception handling implementation using objects that maintain arbitrary state about the nature of the exception. Such a scheme is sometimes preferable to matching particular string values of $@ using regular expressions. Here's an example:

```
    eval { ... ; die Some::Module::Exception->new( FOO => "bar" ) };
    if ($@) {
        if (ref($@) && UNIVERSAL::isa($@,"Some::Module::Exception")) {
            # handle Some::Module::Exception
        }
        else {
            # handle all other possible exceptions
        }
    }
```

Because perl will stringify uncaught exception messages before displaying them, you may want to overload stringification operations on such custom 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 will be called with the error text and can change the error message, if it sees fit, by calling `die` again. See $SIG{expr} in *perlvar* for details on setting `%SIG` entries, and §**??** for some examples. Although this feature was meant to be run only right before your program was to exit, this is not currently the case–the `$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 $ˆS in *perlvar*). Because this promotes strange action at a distance, this counterintuitive behavior may be fixed in a future release.

do BLOCK

Not really a function. Returns the value of the last command in the sequence of commands indicated by BLOCK. When modified by a 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 *perlsyn* for alternative strategies.

do SUBROUTINE(LIST)

A deprecated form of subroutine call. See *perlsub*.

do EXPR

Uses the value of EXPR as a filename and executes the contents of the file as a Perl script. Its primary use is to include subroutines from a Perl subroutine library.

```
    do 'stat.pl';
```

is just like

```
    eval `cat stat.pl`;
```

except that it's more efficient and concise, keeps track of the current filename for error messages, searches the @INC libraries, and updates `%INC` if the file is found. See Predefined Names in *perlvar* 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 cannot read the file, it returns undef and sets `$!` to the error. If do can read the file but cannot compile it, it returns undef and sets an error message in `$@`. If the file is successfully compiled, do returns the value of the last expression evaluated.

Note that 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

This function causes an immediate core dump. See also the **-u** command-line switch in *perlrun*, 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.

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 on the part of Perl.

This function is now largely obsolete, partly because it's very hard to convert a core file into an executable, and because the real compiler backends for generating portable bytecode and compilable C code have superseded it. That's why you should now invoke it as `CORE::dump()`, if you don't want to be warned against a possible typo.

If you're looking to use *dump* to speed up your program, consider generating bytecode or native C code as described in *perlcc*. If you're just trying to accelerate a CGI script, consider using the `mod_perl` extension to **Apache**, or the CPAN module, CGI::Fast. You might also consider autoloading or selfloading, which at least make your program *appear* to run faster.

each HASH

When called in list context, returns a 2-element list consisting of the key and value for the next element of a hash, so that you can iterate over it. When called in scalar context, returns only the key for the next element in the hash.

Entries are returned in an apparently random order. The actual random order is subject to change in future versions of perl, but it is guaranteed to be in the same order as either the `keys` or `values` function would produce on the same (unmodified) hash. Since Perl 5.8.1 the ordering is different even between different runs of Perl for security reasons (see Algorithmic Complexity Attacks in *perlsec*).

When the hash is entirely read, a null array is returned in list context (which when assigned produces a false (`0`) value), and `undef` in scalar context. The next call to `each` after that will start iterating again. There is a single iterator for each hash, shared by all `each`, `keys`, and `values` function calls in the program; it can be reset by reading all the elements from the hash, or by evaluating `keys HASH` or `values HASH`. If you add or delete elements of a hash while you're iterating over it, you may get entries skipped or duplicated, so don't. Exception: It is always safe to delete the item most recently returned by `each()`, which means that the following code will work:

```
        while (($key, $value) = each %hash) {
          print $key, "\n";
          delete $hash{$key};   # This is safe
        }
```

The following prints out your environment like the printenv(1) program, only in a different order:

```
    while (($key,$value) = each %ENV) {
        print "$key=$value\n";
    }
```

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 very 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 very 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 I/O Operators in *perlop*.

In a `while (<>)` loop, `eof` or `eof(ARGV)` can be used to detect the end of each file, `eof()` will only detect the end of the last file. 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 if there was an error.

eval EXPR

eval BLOCK

In the first form, 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 weren't any errors, executed in the lexical context of the current Perl program, so that any 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.

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 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, an undefined value is returned by `eval`, and `$@` is set to the error message. If there was no error, `$@` is guaranteed to be a null string. 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 warn, *perlvar*, *warnings* and *perllexwarn*.

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 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 $@
```

Due to the current arguably broken state of `__DIE__` hooks, when using the `eval{}` form as an exception trap in libraries, 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 shown in this example:

```
# a very 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.

`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.

Note that as a very special case, an `eval ''` executed within 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.

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 there is a following statement which isn't `die`, `warn`, or `exit` (if `-w` is set - but you always do that). 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, or if LIST is an array with more than one value, calls execvp(3) with the arguments in LIST. If there is only one scalar argument or an array with one element in it, 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. (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 will be subject to its quirks and capabilities. See 'STRING' in *perlop* 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 literally called *"echo surprise"*, didn't find it, and set `$?` to a non-zero value indicating failure.

Beginning with v5.6.0, Perl will attempt to flush all files opened for output before the exec, but this may not be supported on some platforms (see *perlport*). To be safe, you may need to set `$|` ($AUTOFLUSH in English) or call the `autoflush()` method of `IO::Handle` on any open handles in order to avoid lost output.

Note that `exec` will not call your `END` blocks, nor will it call any `DESTROY` methods in your objects.

exists EXPR

Given an expression that specifies a hash element or array element, returns true if the specified element in the hash or array has ever been initialized, even if the corresponding value is undefined. The element is not autovivified if it doesn't exist.

```
    print "Exists\n"    if exists $hash{$key};
    print "Defined\n"   if defined $hash{$key};
    print "True\n"      if $hash{$key};

    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 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 which 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 *perlsub*.

```
    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 deepest nested array or hash 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:

```
    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.

See Pseudo-hashes: Using an array as a hash in *perlref* for specifics on how exists() acts when used on a pseudo-hash.

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

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. If this is a problem, you can call `POSIX:_exit($status)` to avoid END and destructor processing. See *perlmod* for details.

exp EXPR

exp

Returns *e* (the natural logarithm base) to the power of EXPR. If EXPR is omitted, gives `exp($_)`.

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 return works 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` will produce a fatal error 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";
```

fileno FILEHANDLE

Returns the file descriptor for a filehandle, or undefined if the filehandle is not open. 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) == fileno(THAT)) {
        print "THIS and THAT are dups\n";
    }
```

(Filehandles connected to memory objects via new features of `open` may return undefined even though they are open.)

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 only entire files, not records.

Two potentially non-obvious but traditional `flock` semantics are that it waits indefinitely until the lock is granted, and that its locks **merely advisory**. Such discretionary locks are more flexible, but offer fewer guarantees. This means that files locked with `flock` may be modified by programs that do not also use `flock`. See *perlport*, your port's specific documentation, or 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` will return 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 perl.

Here's a mailbox appender for BSD systems.

```
    use Fcntl ':flock'; # import LOCK_* constants

    sub lock {
        flock(MBOX,LOCK_EX);
        # and, in case someone appended
        # while we were waiting...
        seek(MBOX, 0, 2);
    }

    sub unlock {
        flock(MBOX,LOCK_UN);
    }

    open(MBOX, ">>/usr/spool/mail/$ENV{'USER'}")
            or die "Can't open mailbox: $!";

    lock();
    print MBOX $msg,"\n\n";
    unlock();
```

On systems that support a real flock(), locks are inherited across fork() calls, whereas those that must resort to the more capricious fcntl() function lose the locks, making it harder to write servers.

See also DB_File for other flock() examples.

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.

Beginning with v5.6.0, Perl will attempt to flush all files opened for output before forking the child process, but this may not be supported on some platforms (see *perlport*). To be safe, you may need to set `$|` ($AUTOFLUSH in English) or call the `autoflush()` method of `IO::Handle` on any open handles in order 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 *perlipc* 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.

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 *perlform* for many details and examples.

formline PICTURE,LIST

This is an internal function used by `formats`, though you may call it, too. It formats (see *perlform*) 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 `$^A` are written to some filehandle, but you could also read `$^A` yourself 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 will 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 *perlform* for other examples.

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 null
    }
    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 site; details on CPAN can be found on CPAN in *perlmodlib*.

getlogin

Implements the C library function of the same name, which on most systems returns the current login from */etc/utmp*, if any. If null, use `getpwuid`.

```
    $login = getlogin || getpwuid($<) || "Kilroy";
```

Do not consider `getlogin` for authentication: it is not as secure as `getpwuid`.

getpeername SOCKET

Returns the packed sockaddr address of 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 process group of current process. Note that the POSIX version of `getpgrp` does not accept a PID argument, so only `PID==0` is truly portable.

getppid

Returns the process id of the parent process.

Note for Linux users: on Linux, the C functions `getpid()` and `getppid()` return different values from different threads. In order to be portable, this behavior is not reflected by the perl-level function `getppid()`, that returns a consistent value across threads. If you want to call the underlying `getppid()`, you may use the CPAN module `Linux::Pid`.

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).

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 perform the same functions as their counterparts in the system library. In list context, the return values from the various get routines are as follows:

```
($name,$passwd,$uid,$gid,
   $quota,$comment,$gcos,$dir,$shell,$expire) = getpw*
($name,$passwd,$gid,$members) = getgr*
($name,$aliases,$addrtype,$length,@addrs) = gethost*
($name,$aliases,$addrtype,$net) = getnet*
($name,$aliases,$proto) = getproto*
($name,$aliases,$port,$proto) = getserv*
```

(If the entry doesn't exist you get a null list.)

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 *perlsec*). The $passwd and $shell, user's encrypted password and login shell, are also tainted, because of 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 cases in the sense that in many systems they are unsupported. 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 your getpwnam(3) documentation and your *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 only supported 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 which 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 the raw addresses returned by the corresponding system library call. In the Internet domain, each address is four bytes long and you can unpack it by saying something like:

```
($a,$b,$c,$d) = unpack('C4',$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);
```

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 like they're the same method calls (uid), they aren't, because a `File::stat` object is different from a `User::pwent` object.

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 call returns a packed string representing the requested socket option, or `undef` if there is an error (the error reason will be in $!). What exactly is in the packed string depends in the LEVEL and OPTNAME, consult your system documentation for details. A very common case however is that the option is an integer, in which case the result will be an packed integer which you can decode using unpack with the `i` (or `I`) format.

An example testing if Nagle's algorithm is turned on on a socket:

```
use Socket;

defined(my $tcp = getprotobyname("tcp"))
    or die "Could not determine the protocol number for tcp";
# my $tcp = Socket::IPPROTO_TCP; # Alternative
my $packed = getsockopt($socket, $tcp, Socket::TCP_NODELAY)
    or die "Could not query TCP_NODELAY SOCKEt option: $!";
my $nodelay = unpack("I", $packed);
print "Nagle's algorithm is turned ", $nodelay ? "off\n" : "on\n";
```

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 I/O Operators in *perlop*.

Beginning with v5.6.0, this operator is implemented using the standard `File::Glob` extension. See *File::Glob* for details.

gmtime EXPR

Converts a time as returned by the time function to an 8-element list with the time localized for the standard Greenwich time zone. Typically used as follows:

```
    #  0    1    2     3     4    5     6     7
    ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday) =
                                            gmtime(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 is the month itself, in the range `0..11` with 0 indicating January and 11 indicating December. $year is the number of years since 1900. That is, $year is `123` in year 2023. $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.)

Note that the $year element is *not* simply the last two digits of the year. If you assume it is, then you create non-Y2K-compliant programs–and you wouldn't want to do that, would you?

The proper way to get a complete 4-digit year is simply:

```
        $year += 1900;
```

And to get the last two digits of the year (e.g., '01' in 2001) do:

```
        $year = sprintf("%02d", $year % 100);
```

If EXPR is omitted, `gmtime()` uses the current time (`gmtime(time)`).

In scalar context, `gmtime()` returns the ctime(3) value:

```
    $now_string = gmtime;  # e.g., "Thu Oct 13 04:54:34 1994"
```

If you need local time instead of GMT use the localtime builtin. See also the `timegm` function provided by the `Time::Local` module, and the strftime(3) and mktime(3) functions available via the *POSIX* module.

This scalar value is **not** locale dependent (see *perllocale*), but is instead a Perl builtin. To get somewhat similar but locale dependent date strings, see the example in localtime.

goto LABEL

goto EXPR

goto &NAME

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, or 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 being 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 a label name, whose scope will be resolved dynamically. This allows for computed `goto`s per FORTRAN, but isn't necessarily recommended if you're optimizing for maintainability:

```
    goto ("FOO", "BAR", "GLARCH")[$i];
```

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 which 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.

See also map 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.) 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().

import

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 use, *perlmod*, 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. The return value is based at `0` (or whatever you've set the `$[` variable to–but don't do that). If the substring is not found, returns one less than the base, ordinarily `-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 "ioctl.ph"; # probably in /usr/local/lib/perl/ioctl.ph
```

to get the correct function definitions. If *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 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.

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.

keys HASH

Returns a list consisting of all the keys of the named hash. (In scalar context, returns the number of keys.)

The keys are returned in an apparently random order. The actual random order is subject to change in future versions of perl, but it is guaranteed to be the same order as either the `values` or `each` function produces (given that the hash has not been modified). Since Perl 5.8.1 the ordering is different even between different runs of Perl for security reasons (see Algorithmic Complexity Attacks in *perlsec*).

As a side effect, calling keys() resets the HASH's internal iterator, see each. (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 values.

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;
}
```

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).

See also `each`, `values` and `sort`.

kill SIGNAL, LIST

Sends a signal to a list of processes. Returns the number of processes successfully signaled (which is not necessarily the same as the number actually killed).

```
$cnt = kill 1, $child1, $child2;
kill 9, @goners;
```

If SIGNAL is zero, no signal is sent to the process. This is a useful way to check that a child process is alive and hasn't changed its UID. See *perlport* for notes on the portability of this construct.

Unlike in the shell, if SIGNAL is negative, it kills process groups instead of processes. (On System V, a negative *PROCESS* number will also kill process groups, but that's not portable.) That means you usually want to use positive not negative signals. You may also use a signal name in quotes.

See Signals in *perlipc* for more details.

last LABEL

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 `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 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 `last` can be used to effect an early exit out of such a block.

See also continue for an illustration of how `last`, `next`, and `redo` work.

lc EXPR

lc

Returns a lowercased version of EXPR. This is the internal function implementing the `\L` escape in double-quoted strings. Respects current LC_CTYPE locale if `use locale` in force. See *perllocale* and *perlunicode* for more details about locale and Unicode support.

If EXPR is omitted, uses `$_`.

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. Respects current LC_CTYPE locale if `use locale` in force. See *perllocale* and *perlunicode* for more details about locale and Unicode support.

If EXPR is omitted, uses `$_`.

length EXPR

length

Returns the length in *characters* of the value of EXPR. If EXPR is omitted, returns length of `$_`. Note that this 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.

Note the *characters*: if the EXPR is in Unicode, you will get the number of characters, not the number of bytes. To get the length in bytes, use `do { use bytes; length(EXPR) }`, see *bytes*.

link OLDFILE,NEWFILE

Creates a new filename linked to the old filename. Returns true for success, false otherwise.

listen SOCKET,QUEUESIZE

Does the same thing that the listen system call does. Returns true if it succeeded, false otherwise. See the example in Sockets: Client/Server Communication in *perlipc*.

local EXPR

You really probably want to be using `my` instead, because `local` isn't what most people think of as "local". See Private Variables via my() in *perlsub* 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 Temporary Values via local() in *perlsub* for details, including issues with tied arrays and hashes.

localtime EXPR

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 is the month itself, in the range `0..11` with 0 indicating January and 11 indicating December. $year is the number of years since 1900. That is, $year is `123` in year 2023. $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 savings time, false otherwise.

Note that the $year element is *not* simply the last two digits of the year. If you assume it is, then you create non-Y2K-compliant programs–and you wouldn't want to do that, would you?

The proper way to get a complete 4-digit year is simply:

```
        $year += 1900;
```

And to get the last two digits of the year (e.g., ’01’ in 2001) do:

```
        $year = sprintf("%02d", $year % 100);
```

If EXPR is omitted, `localtime()` uses the current time (`localtime(time)`).

In scalar context, `localtime()` returns the ctime(3) value:

```
    $now_string = localtime;  # e.g., "Thu Oct 13 04:54:34 1994"
```

This scalar value is **not** locale dependent but is a Perl builtin. For GMT instead of local time use the gmtime builtin. See also the `Time::Local` module (to convert the second, minutes, hours, ... 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 *perllocale*) 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.

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.

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. (However, if you’ve said `use threads`, lock() is always a keyword.) See *threads*.

log EXPR

log

Returns the natural logarithm (base *e*) of EXPR. If EXPR is omitted, returns 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 for the inverse operation.

lstat EXPR

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 `$_`.

m//

The match operator. See *perlop*.

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, @nums);
```

translates a list of numbers to the corresponding characters. And

```
    %hash = map { getkey($_) => $_ } @array;
```

is just a funny way to write

```
    %hash = ();
    foreach $_ (@array) {
        $hash{getkey($_)} = $_;
    }
```

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 grep for an array composed of those items of the original list for which the BLOCK or EXPR evaluates to true.

`{` 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 its dealing with based 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 + 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$_", 1) } @array  # this also works
    %hash = map {  lc($_), 1  } @array  # as does 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 , at end
```

and you get list of anonymous hashes each with only 1 entry.

mkdir FILENAME,MASK

mkdir FILENAME

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). If omitted, MASK defaults to 0777.

In general, it is better to create directories with 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.

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 which 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 SysV IPC in *perlipc*, `IPC::SysV`, and `IPC::Semaphore` documentation.

msgget KEY,FLAGS

Calls the System V IPC function msgget(2). Returns the message queue id, or the undefined value if there is an error. See also SysV IPC in *perlipc* and `IPC::SysV` and `IPC::Msg` documentation.

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, or false if there is an error. See also SysV IPC in *perlipc*, `IPC::SysV`, and `IPC::SysV::Msg` documentation.

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, and be followed by the length of the actual message, and finally the message itself. This kind of packing can be achieved with `pack("l! a*", $type, $message)`. Returns true if successful, or false if there is an error. See also `IPC::SysV` and `IPC::SysV::Msg` documentation.

my EXPR

my TYPE EXPR

my EXPR : ATTRS

my TYPE EXPR : ATTRS

A `my` declares the listed variables to be local (lexically) to the enclosing block, file, or `eval`. If more than one value is listed, the list must be placed in parentheses.

The exact semantics and interface of TYPE and ATTRS are still evolving. TYPE is currently bound to the use of `fields` pragma, and attributes are handled using the `attributes` pragma, or starting from Perl 5.8.0 also via the `Attribute::Handlers` module. See Private Variables via my() in *perlsub* for details, and *fields*, *attributes*, and *Attribute::Handlers*.

next LABEL

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 the LABEL is omitted, the command refers to the innermost enclosing loop.

`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 for an illustration of how `last`, `next`, and `redo` work.

no Module VERSION LIST

no Module VERSION

no Module LIST

no Module

See the `use` function, which `no` is the opposite of.

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 the standard Perl or C 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():

```
    $perms = (stat("filename"))[2] & 07777;
    $oct_perms = sprintf "%lo", $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 will automatically convert strings into numbers as needed, this automatic conversion assumes base 10.)

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.

(The following is a comprehensive reference to open(): for a gentler introduction you may consider *perlopentut*.)

If FILEHANDLE is an undefined scalar variable (or array or hash element) the variable is assigned a reference to a new anonymous filehandle, otherwise if FILEHANDLE is an expression, its value is used as the name of the real filehandle wanted. (This is considered a symbolic reference, so `use strict 'refs'` should *not* be in effect.)

If EXPR is omitted, the scalar variable of the same name as the FILEHANDLE contains the filename. (Note that lexical variables–those declared with `my`–will not work for this purpose; so if you're using `my`, specify EXPR in your call to open.)

If three or more arguments are specified then the mode of opening and the file name are separate. If MODE is `'<'` or nothing, the file is opened for input. If MODE is `'>'`, the file is truncated and opened for output, being created if necessary. 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 *perlrun* for a better approach. The file is created with permissions of `0666` modified by the process' `umask` value.

These various prefixes correspond to the fopen(3) modes of `'r'`, `'r+'`, `'w'`, `'w+'`, `'a'`, and `'a+'`.

In the 2-arguments (and 1-argument) form of the call the mode and filename should be concatenated (in this order), possibly separated by spaces. It is possible to omit the mode in these forms if the mode is `'<'`.

If the filename begins with `'|'`, the filename is interpreted as a command to which output is to be piped, and if the filename ends with a `'|'`, the filename is interpreted as a command which pipes output to us. See Using open() for IPC in *perlipc* 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 Bidirectional Communication with Another Process in *perlipc* for alternatives.)

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 which pipes output to us. In the 2-arguments (and 1-argument) form one should replace dash (`'-'`) with the command. See Using open() for IPC in *perlipc* 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 Bidirectional Communication in *perlipc* for alternatives.)

In the three-or-more argument form of pipe opens, 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 specified. Experimental "layers" may give extra LIST arguments meaning.

In the 2-arguments (and 1-argument) form opening `'-'` opens STDIN and opening `'>-'` opens STDOUT.

You may use the three-argument form of open to specify IO "layers" (sometimes also referred to as "disciplines") to be applied to the handle that affect how the input and output are processed (see *open* and *PerlIO* for more details). For example

```
open(FH, "<:utf8", "file")
```

will open the UTF-8 encoded file containing Unicode characters, see *perluniintro*. (Note that if layers are specified in the three-arg form then default layers set by the `open` pragma are ignored.)

Open returns nonzero upon 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 binmode 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, which delimit lines with a single character, and which encode that character in C as `"\n"`, do not need `binmode`. The rest need it.

When opening a file, it's usually a bad idea to continue normal execution if the request failed, so `open` is frequently used in connection with `die`. Even if `die` won't do what you want (say, in a CGI script, where you want to make a nicely formatted error message (but there are modules that can help with that problem)) you should always check the return value from opening a file. The infrequent exception is when working with an unopened filehandle is actually what you want to do.

As a special case the 3 arg form with a read/write mode and the third argument being `undef`:

```
    open(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.

File handles can be opened to "in memory" files held in Perl scalars via:

```
    open($fh, '>', \$variable) || ..
```

Though if you try to re-open `STDOUT` or `STDERR` as an "in memory" file, you have to close it first:

```
    close STDOUT;
    open STDOUT, '>', \$variable or die "Can't open STDOUT: $!";
```

Examples:

```
    $ARTICLE = 100;
    open ARTICLE or die "Can't find article $ARTICLE: $!\n";
    while (<ARTICLE>) {...
    open(LOG, '>>/usr/spool/news/twitlog');     # (log is reserved)
    # if the open fails, output is discarded

    open(DBASE, '+<', 'dbase.mine')             # open for update
        or die "Can't open 'dbase.mine' for update: $!";

    open(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 end up 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
        }
    }
```

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 3 arg 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), and vice versa. But with `open(A, '>>&=B')` the filehandles will share the same file descriptor.

Note that if you are using Perls older than 5.8.0, Perl will be using the standard C libraries' 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 Perl has been compiled with PerlIO or not by running `perl -V` and looking for `useperlio=` line. If `useperlio` is `define`, you have PerlIO, otherwise you don't.

If you open a pipe on the command `'-'`, i.e., either `'|-'` or `'-|'` with 2-arguments (or 1-argument) form of open(), then there is an implicit fork done, and the return value of open is the pid of the child within the parent process, and `0` within the child process. (Use `defined($pid)` to determine whether the open was successful.) 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 or 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 you are running setuid, and don't want to have to scan shell commands for metacharacters. The following triples 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 example in each block shows the pipe as "list form", which is not yet supported on all platforms. A good rule of thumb is that if your platform has true `fork()` (in other words, if your platform is UNIX) you can use the list form.

See Safe Pipe Opens in *perlipc* for more examples of this.

Beginning with v5.6.0, 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 *perlport*). 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 $^F. See `$^F` in *perlvar*.

Closing any piped filehandle causes the parent process to wait for the child to finish, and returns the status value in `$?`.

The filename passed to 2-argument (or 1-argument) form of open() will have leading and trailing whitespace deleted, and the 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 3-argument form to open a file with arbitrary weird characters in it,

```
    open(FOO, '<', $file);
```

otherwise it's necessary to protect any leading and trailing whitespace:

```
    $file =~ s#^(\s)#./$1#;
    open(FOO, "< $file\0");
```

(this may not work on some bizarre filesystems). One should conscientiously choose between the *magic* and 3-arguments form of open():

```
    open IN, $ARGV[0];
```

will allow the user to specify an argument of the form "`rsh cat file |`", but will not work on a filename which happens to have a trailing space, while

```
    open IN, '<', $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>;
```

Using the constructor from the `IO::Handle` package (or one of its subclasses, such as `IO::File` or `IO::Socket`), you can generate anonymous filehandles that have the scope of whatever variables hold references to them, and automatically close whenever and however you leave that scope:

```
    use IO::File;
    #...
    sub read_myfile_munged {
        my $ALL = shift;
        my $handle = new IO::File;
        open($handle, "myfile") or die "myfile: $!";
        $first = <$handle>
            or return ();     # Automatically closed here.
        mung $first or die "mung failed";       # Or here.
        return $first, <$handle> if $ALL;       # Or here.
        $first;                                 # Or here.
    }
```

See seek for some details about mixing reading and writing.

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. DIRHANDLEs have their own namespace separate from FILEHANDLEs.

ord EXPR

ord

Returns the numeric (the native 8-bit encoding, like ASCII or EBCDIC, or Unicode) value of the first character of EXPR. If EXPR is omitted, uses `$_`.

For the reverse, see chr. See *perlunicode* and *encoding* for more about Unicode.

our EXPR

our EXPR TYPE

our EXPR : ATTRS

our TYPE EXPR : ATTRS

An `our` declares the listed variables to be valid globals within the enclosing block, file, or `eval`. That is, it has the same scoping rules as a "my" declaration, but does not create a local variable. If more than one value is listed, the list must be placed in parentheses. The `our` declaration has no semantic effect unless "use strict vars" is in effect, in which case it lets you use the declared global variable without qualifying it with a package name. (But only within the lexical scope of the `our` declaration. In this it differs from "use vars", which is package scoped.)

An `our` declaration declares a global 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
```

Multiple `our` declarations in the same lexical scope are allowed if they are in different packages. If they happened to be in the same package, Perl will emit warnings if you have asked for them.

```
    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
```

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 `fields` pragma, and attributes are handled using the `attributes` pragma, or starting from Perl 5.8.0 also via the `Attribute::Handlers` module. See Private Variables via my() in *perlsub* for details, and *fields*, *attributes*, and *Attribute::Handlers*.

The only currently recognized `our()` attribute is `unique` which indicates that a single copy of the global is to be used by all interpreters should the program happen to be running in a multi-interpreter environment. (The default behaviour would be for each interpreter to have its own copy of the global.) Examples:

```
    our @EXPORT : unique = qw(foo);
    our %EXPORT_TAGS : unique = (bar => [qw(aa bb cc)]);
    our $VERSION : unique = "1.00";
```

Note that this attribute also has the effect of making the global readonly when the first new interpreter is cloned (for example, when the first new thread is created).

Multi-interpreter environments can come to being either through the fork() emulation on Windows platforms, or by embedding perl in a multi-threaded application. The `unique` attribute does nothing in all other environments.

Warning: the current implementation of this attribute operates on the typeglob associated with the variable; this means that `our $x : unique` also has the effect of `our @x : unique; our %x : unique`. This may be subject to change.

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 a converted integer may be represented by a sequence of 4 bytes.

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 value.
C   An unsigned char value.  Only does bytes.  See U for Unicode.

s   A signed short value.
S   An unsigned short value.
      (This 'short' is _exactly_ 16 bits, which may differ from
       what a local C compiler calls 'short'.  If you want
       native-length shorts, use the '!' suffix.)

i   A signed integer value.
I   An unsigned integer value.
      (This 'integer' is _at_least_ 32 bits wide.  Its exact
       size depends on what a local C compiler calls 'int',
       and may even be larger than the 'long' described in
       the next item.)

l   A signed long value.
L   An unsigned long value.
      (This 'long' is _exactly_ 32 bits, which may differ from
       what a local C compiler calls 'long'.  If you want
       native-length longs, use the '!' suffix.)

n   An unsigned short in "network" (big-endian) order.
N   An unsigned long in "network" (big-endian) order.
v   An unsigned short in "VAX" (little-endian) order.
V   An unsigned long in "VAX" (little-endian) order.
      (These 'shorts' and 'longs' are _exactly_ 16 bits and
       _exactly_ 32 bits, respectively.)

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.
       Causes a fatal error otherwise.)

j   A signed integer value (a Perl internal integer, IV).
J   An unsigned integer value (a Perl internal unsigned integer, UV).

f   A single-precision float in the native format.
d   A double-precision float in the native format.

F   A floating point value in the native native format
       (a Perl internal floating point value, NV).
D   A long double-precision float in the native format.
      (Long doubles are available only if your system supports long
       double values _and_ if Perl has been compiled to support those.
       Causes a fatal error otherwise.)

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 UTF-8 internally
    (or UTF-EBCDIC in EBCDIC platforms).

w   A BER compressed integer.  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.
X   Back up a byte.
@   Null fill to absolute position, counted from the start of
    the innermost ()-group.
(   Start of a ()-group.
```

The following rules apply:

- Each letter may optionally be followed by a number giving a repeat count. With all types except `a`, `A`, `Z`, `b`, `B`, `h`, `H`, `@`, `x`, `X` and `P` the pack function will gobble up that many values from the LIST. A * for the repeat count means to use however many items are left, except for `@`, `x`, `X`, where it is equivalent to `0`, and `u`, where it is equivalent to 1 (or 45, what is the same). A numeric repeat count may optionally be enclosed in brackets, as in `pack 'C[80]', @arr`.

  One can replace the numeric repeat count by a template enclosed in brackets; then the packed length of this template in bytes is used as a count. For example, `x[L]` skips a long (it skips the number of bytes in a long); the template `$t X[$t] $t` unpack()s twice what $t 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 has the maximal possible alignment.

  When used with `Z`, * results in the addition of a trailing null byte (so the packed result will be one longer than the byte `length` of the item).

  The repeat count for `u` is interpreted as the maximal number of bytes to encode per line of output, with 0 and 1 replaced by 45.

- 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 necessary. When unpacking, `A` strips trailing spaces and nulls, `Z` strips everything after the first null, and `a` returns data verbatim. When packing, `a`, and `Z` are equivalent.

  If the value-to-pack is too long, it is truncated. If 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 byte under all circumstances.

- Likewise, the `b` and `B` fields pack a string that many bits long. Each byte of the input field of pack() generates 1 bit of the result. Each result bit is based on the least-significant bit of the corresponding input byte, i.e., on `ord($byte)%2`. In particular, bytes `"0"` and `"1"` generate bits 0 and 1, as do bytes `"\0"` and `"\1"`.

  Starting from the beginning of the input string of pack(), each 8-tuple of bytes is converted to 1 byte of output. With format `b` the first byte of the 8-tuple determines the least-significant bit of a byte, and with format `B` it determines the most-significant bit of a byte.

  If the length of the input string is not exactly divisible by 8, the remainder is packed as if the input string were padded by null bytes at the end. Similarly, during unpack()ing the "extra" bits are ignored.

  If the input string of pack() is longer than needed, extra bytes are ignored. A * for the repeat count of pack() means to use all the bytes of the input field. On unpack()ing the bits are converted to a string of `"0"`s and `"1"`s.

- The `h` and `H` fields pack a string that many nybbles (4-bit groups, representable as hexadecimal digits, 0-9a-f) long.

  Each byte of the input field of pack() generates 4 bits of the result. For non-alphabetical bytes the result is based on the 4 least-significant bits of the input byte, i.e., on `ord($byte)%16`. In particular, bytes `"0"` and `"1"` generate nybbles 0 and 1, as do bytes `"\0"` and `"\1"`. For bytes `"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`. The result for bytes `"g".."z"` and `"G".."Z"` is not well-defined.

Starting from the beginning of the input string of pack(), each pair of bytes is converted to 1 byte of output. With format `h` the first byte of the pair determines the least-significant nybble of the output byte, and 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 byte at the end. Similarly, during unpack()ing the "extra" nybbles are ignored.

If the input string of pack() is longer than needed, extra bytes are ignored. A `*` for the repeat count of pack() means to use all the bytes of the input field. On unpack()ing the bits are converted to a string of hexadecimal digits.

- The `p` type packs a pointer to a null-terminated string. You are responsible for ensuring the string is not a temporary value (which can potentially get deallocated before you get around to using the packed result). The `P` type 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 for unpack().
- The `/` template character allows packing and unpacking of strings where the packed structure contains a byte count followed by the string itself. You write *length-item*`/`*string-item*.

  The *length-item* can be any `pack` template letter, and describes how the length value is packed. The ones 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`, the *string-item* must, at present, be `"A*"`, `"a*"` or `"Z*"`. For `unpack` the length of the string is obtained from the *length-item*, but if you put in the '*' it will be ignored. For all other codes, `unpack` applies the length value to the next item, which must not have a repeat count.

```
    unpack 'C/a', "\04Gurusamy";        gives 'Guru'
    unpack 'a3/A* A*', '007 Bond  J ';  gives (' Bond','J')
    pack 'n/a* w/a*','hello,','world';  gives "\000\006hello,\005world"
```

  The *length-item* is not returned explicitly from `unpack`.

  Adding a count to the *length-item* letter is unlikely to do anything useful, unless that letter is `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 immediately followed by a `!` suffix to signify native shorts or longs–as you can see from above for example a bare `l` does mean exactly 32 bits, the native `long` (as seen by the local C compiler) may be larger. This is an issue mainly in 64-bit platforms. You can see whether using `!` makes any difference by

```
        print length(pack("s")), " ", length(pack("s!")), "\n";
        print length(pack("l")), " ", length(pack("l!")), "\n";
```

  `i!` and `I!` also work but only because of completeness; 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 via *Config*:

```
        use Config;
        print $Config{shortsize},    "\n";
        print $Config{intsize},      "\n";
        print $Config{longsize},     "\n";
        print $Config{longlongsize}, "\n";
```

  (The `$Config{longlongsize}` will be undefined if your system does not support long longs.)
- The integer formats `s`, `S`, `i`, `I`, `l`, `L`, `j`, and `J` are inherently non-portable between processors and operating systems because they obey the 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, the Intel and VAX CPUs are little-endian, while everybody else, for example Motorola m68k/88k, PPC, Sparc, HP PA, Power, and Cray are big-endian. Alpha and MIPS can be either: Digital/Compaq used/uses them in little-endian mode; SGI/Cray uses them in big-endian mode.

The names 'big-endian' and 'little-endian' are comic references to the classic "Gulliver's Travels" (via the paper "On Holy Wars and a Plea for Peace" by Danny Cohen, USC/ISI IEN 137, April 1, 1980) and the egg-eating habits of the Lilliputians.

Some systems may have even weirder byte orders such as

```
0x56 0x78 0x12 0x34
0x34 0x12 0x78 0x56
```

You can see your system's preference with

```
print join(" ", map { sprintf "%#02x", $_ }
                    unpack("C*",pack("L",0x12345678))), "\n";
```

The byteorder on the platform where Perl was built is also available via *Config*:

```
use Config;
print $Config{byteorder}, "\n";
```

Byteorders `'1234'` and `'12345678'` are little-endian, `'4321'` and `'87654321'` are big-endian.

If you want portable packed integers use the formats `n`, `N`, `v`, and `V`, their byte endianness and size are known. See also *perlport*.

- Real numbers (floats and doubles) are in the native machine format only; due to the multiplicity of floating formats around, and the lack of a standard "network" representation, 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 (as the endian-ness of the memory representation is not part of the IEEE spec). See also *perlport*.

  Note that Perl uses doubles internally for all numeric calculation, and converting from double into float and thence back to double again will lose precision (i.e., `unpack("f", pack("f", $foo))` will not in general equal $foo).

- If the pattern begins with a `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 `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 `C0` (or anything else) to force Perl not to UTF-8 encode your string, and then follow this with a `U*` somewhere in your pattern.

- You must yourself do any alignment or padding by inserting for example enough `'x'`es while packing. There is no way to pack() and unpack() could know where the bytes are going to or coming from. Therefore `pack` (and `unpack`) handle their output and input as flat sequences of bytes.

- A ()-group is a sub-TEMPLATE enclosed in parentheses. A group may take a repeat count, both as postfix, and for unpack() also via the `/` template character. Within each repetition of a group, positioning with `@` starts again at 0. Therefore, the result of

  ```
  pack( '@1A((@2A)@3A)', 'a', 'b', 'c' )
  ```

  is the string "\0a\0\0bc".

- `x` and `X` accept `!` modifier. In this case they act as alignment commands: they jump forward/back to the closest position aligned at a multiple of `count` bytes. For example, to pack() or unpack() C's `struct {char c; 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 on the double's size.

  For alignment commands `count` of 0 is equivalent to `count` of 1; both result in no-ops.

- A comment in a TEMPLATE starts with `#` and goes to the end of line. White space may be used to separate pack codes from each other, but a `!` modifier and a repeat count must follow immediately.

- If TEMPLATE requires more arguments to pack() than actually given, pack() assumes additional `""` arguments. If TEMPLATE requires less arguments to pack() than actually given, extra arguments are ignored.

Examples:

```
$foo = pack("CCCC",65,66,67,68);
# foo eq "ABCD"
$foo = pack("C4",65,66,67,68);
# same thing
$foo = pack("U4",0x24b6,0x24b7,0x24b8,0x24b9);
# same thing with Unicode circled letters

$foo = pack("ccxxcc",65,66,67,68);
# foo eq "AB\0\0CD"

# note: the above examples featuring "C" and "c" are true
# only on ASCII and ASCII-derived systems such as ISO Latin 1
# and UTF-8.  In EBCDIC the first example would be
# $foo = pack("CCCC",193,194,195,196);

$foo = pack("s2",1,2);
# "\1\0\2\0" on little-endian
# "\0\1\0\2" 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
```

The same template may generally also be used in unpack().

package NAMESPACE

package

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, file, or eval (the same as the `my` operator). All further unqualified dynamic identifiers will be in this namespace. A package statement affects only dynamic variables–including those you've used `local` on–but *not* lexical variables, which are created with `my`. Typically it would be the first declaration in a file to be included by the `require` or `use` operator. 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: `$Package::Variable`. If the package name is null, the `main` package as assumed. That is, `$::sail` is equivalent to `$main::sail` (as well as to `$main'sail`, still seen in older code).

If NAMESPACE is omitted, then there is no current package, and all identifiers must be fully qualified or lexicals. However, you are strongly advised not to make use of this feature. Its use can cause unexpected behaviour, even crashing some versions of Perl. It is deprecated, and will be removed from a future release.

See Packages in *perlmod* for more information about packages, modules, and classes. See *perlsub* for other scoping issues.

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.

See *IPC::Open2*, *IPC::Open3*, and Bidirectional Communication in *perlipc* for examples of such things.

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 $^F in *perlvar*.

pop ARRAY

pop

Pops and returns the last value of the array, shortening the array by one element. Has an effect similar to

```
$ARRAY[$#ARRAY--]
```

If there are no elements in the array, returns the undefined value (although this may happen at other times as well). If ARRAY is omitted, pops the `@ARGV` array in the main program, and the `@_` array in subroutines, just like `shift`.

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). May be modified to change that offset. Such modification will also influence the `\G` zero-width assertion in regular expressions. See *perlre* and *perlop*.

print FILEHANDLE LIST

print LIST

print

Prints a string or a list of strings. Returns true if successful. FILEHANDLE may be a scalar variable name, in which case the variable contains 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 by default to standard output (or to the last selected output channel–see select). If LIST is also omitted, prints `$_` to the currently selected output channel. To set the default output channel to something other than STDOUT use the select operation. The current value of `$,` (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, and any subroutine that you call will have one or more of its expressions evaluated in list context. Also 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–interpose a + or put parentheses around all the arguments.

Note that if you're storing FILEHANDLES in an array or other expression, you will have to use a block returning its value instead:

```
    print { $files[$i] } "stuff\n";
    print { $OK ? STDOUT : STDERR } "stuff\n";
```

printf FILEHANDLE FORMAT, LIST

printf FORMAT, LIST

Equivalent to `print FILEHANDLE sprintf(FORMAT, LIST)`, except that `$\` (the output record separator) is not appended. The first argument of the list will be interpreted as the `printf` format. See `sprintf` for an explanation of the format argument. If `use locale` is in effect, the character used for the decimal point in formatted real numbers is affected by the LC_NUMERIC locale. See *perllocale*.

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

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 a string starting with `CORE::`, the rest is taken as a name for Perl builtin. If the builtin is not *overridable* (such as `qw//`) or its arguments cannot be expressed by a prototype (such as `system`) 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

Treats ARRAY as a stack, and pushes the values of LIST onto 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 new number of elements in the array.

q/STRING/

qq/STRING/

qr/STRING/

qx/STRING/

qw/STRING/

Generalized quotes. See Regexp Quote-Like Operators in *perlop*.

quotemeta EXPR

quotemeta

Returns the value of EXPR with all non-"word" characters backslashed. (That is, all 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.

If EXPR is omitted, uses `$_`.

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 has not been documented 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.)

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 actually implemented in terms of either Perl's or system's fread() call. To get a true read(2) system call, see `sysread`.

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 open, and 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 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 an undefined value in scalar context or a null 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(DIR, $some_dir) || die "can't opendir $some_dir: $!";
    @dots = grep { /^\./ && -f "$some_dir/$_" } readdir(DIR);
    closedir DIR;
```

readline EXPR

Reads from the filehandle whose typeglob is contained in EXPR. 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 however you may have defined it with `$/` or `$INPUT_RECORD_SEPARATOR`). See $/ in *perlvar*.

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 I/O Operators in *perlop*.

```
    $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 takes the necessary steps to ensure that `readline` was successful.

```
    for (;;) {
        undef $!;
        unless (defined( $line = <> )) {
            die $! if $!;
            last; # reached EOF
        }
        # ...
    }
```

readlink EXPR

readlink

Returns the value of a symbolic link, if symbolic links are implemented. If not, gives a fatal error. If there is some system error, returns the undefined value and sets `$!` (errno). If EXPR is omitted, uses `$_`.

readpipe EXPR

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 I/O Operators in *perlop*.

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 UDP: Message Passing in *perlipc* 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 `:utf8` I/O layer (see 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 read.

redo LABEL

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. This command is normally used by programs that want to lie to themselves about what was just input:

```
    # 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 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 `redo` inside such a block will effectively turn it into a looping construct.

See also continue for an illustration of how `last`, `next`, and `redo` work.

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
```

If the referenced object has been blessed into a package, then that package name is returned instead. 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";
    }
    if (UNIVERSAL::isa($r, "HASH")) {  # for subclassing
        print "r is a reference to something that isa hash.\n";
    }
```

See also *perlref*.

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 *perlport* and either the rename(2) manpage or equivalent system documentation for details.

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). A fatal error is produced at run time if VERSION is greater than the version of the current Perl interpreter. Compare with use, 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 which 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, 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`. Has semantics similar to the following subroutine:

```
sub require {
   my ($filename) = @_;
   if (exists $INC{$filename}) {
       return 1 if $INC{$filename};
       die "Compilation failed in require";
   }
   my ($realfilename,$result);
   ITER: {
       foreach $prefix (@INC) {
           $realfilename = "$prefix/$filename";
           if (-f $realfilename) {
               $INC{$filename} = $realfilename;
               $result = do $realfilename;
               last ITER;
           }
       }
       die "Can't find $filename in \@INC";
   }
   if ($@) {
       $INC{$filename} = undef;
       die $@;
   } elsif (!$result) {
       delete $INC{$filename};
       die "$filename did not return true value";
   } else {
       return $result;
   }
}
```

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 in the case of 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 filename with a "*.pmc*" extension. A file with this extension is assumed to be Perl bytecode generated by B::Bytecode. If this file is found, and it's modification time is newer than a coinciding "*.pm*" non-compiled file, it will be loaded in place of that non-compiled file ending in a "*.pm*" extension.

You can also insert hooks into the import facility, by putting directly Perl code 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 being a reference to itself, and the second the name of the file to be included (e.g. "*Foo/Bar.pm*"). The subroutine should return `undef` or a filehandle, from which the file to include will be read. If `undef` is returned, `require` will look at the remaining elements of @INC.

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 enables to pass indirectly some 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 it 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, new Foo(...);
```

Note that these hooks are also permitted to set the %INC entry corresponding to the files they have loaded. See %INC in *perlvar*.

For a yet-more-powerful import facility, see use and *perlmod*.

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. Resets only 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 my.

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`). 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 a void context.

(Note that in the absence of an explicit `return`, a subroutine, eval, or do FILE will automatically return the value of the last expression evaluated.)

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 reverse <>;           # line tac, last line first

    undef $/;                   # for efficiency of <>
    print scalar reverse <>;    # character tac, last line tsrif
```

Used without arguments in scalar context, reverse() reverses `$_`.

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.

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 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 `$!` (errno). If FILENAME is omitted, uses `$_`.

s///

The substitution operator. See *perlop*.

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 unary operator, if you accidentally use for EXPR a parenthesized list, 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 *perlop* 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` upon success, `0` otherwise.

Note the *in bytes*: even if the filehandle has been set to operate on characters (for example by using the `: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 file for `sysread` or `syswrite`, don't use `seek`–buffering makes its effect on the file's system 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 might have to stick in a seek() to reset things. The `seek` doesn't change the current 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 IO implementations are particularly cantankerous), then you may need something more 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`. Has the same caveats about possible directory compaction as the corresponding system library routine.

select FILEHANDLE

select

Returns the currently selected filehandle. Sets the current default filehandle for output, if FILEHANDLE is supplied. This has two effects: first, a `write` or a `print` without a filehandle will default to this FILEHANDLE. Second, references to variables related to output will refer to this output channel. For example, if you have 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);
```

select RBITS,WBITS,EBITS,TIMEOUT

This calls the select(2) system call 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 might wish to write a subroutine:

```
    sub fhbits {
        my(@fhlist) = split(' ',$_[0]);
        my($bits);
        for (@fhlist) {
            vec($bits,fileno($_),1) = 1;
        }
        $bits;
    }
    $rin = fhbits('STDIN TTY SOCK');
```

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 *perlport* for notes on the portability of `select`.

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.

semctl ID,SEMNUM,CMD,ARG

Calls the System V IPC function `semctl`. 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 which 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 SysV IPC in *perlipc*, `IPC::SysV`, `IPC::Semaphore` documentation.

semget KEY,NSEMS,FLAGS

Calls the System V IPC function semget. Returns the semaphore id, or the undefined value if there is an error. See also SysV IPC in *perlipc*, `IPC::SysV`, `IPC::SysV::Semaphore` documentation.

semop KEY,OPSTRING

Calls the System V IPC function semop to perform 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 number of semaphore operations is implied by the length of OPSTRING. Returns true if successful, or false if there is an 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 SysV IPC in *perlipc*, IPC::SysV, and IPC::SysV::Semaphore documentation.

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 C sendto. Returns the number of characters sent, or the undefined value if there is an error. The C system call sendmsg(2) is currently unimplemented. See UDP: Message Passing in *perlipc* 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 :utf8 I/O layer (see open, 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. Will produce a fatal error if 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().

setpriority WHICH,WHO,PRIORITY

Sets the current priority for a process, a process group, or a user. (See setpriority(2).) Will produce a fatal error if used on a machine that doesn't implement setpriority(2).

setsockopt SOCKET,LEVEL,OPTNAME,OPTVAL

Sets the socket option requested. Returns undefined if there is an error. OPTVAL may be specified as undef if you don't want to pass an argument.

shift ARRAY

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 at file scopes or within the lexical scopes established by the eval '', BEGIN {}, INIT {}, CHECK {}, and END {} constructs.

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 which will hold the returned shmid_ds structure. Returns like ioctl: the undefined value for error, "0 but true" for zero, or the actual return value otherwise. See also SysV IPC in *perlipc* and IPC::SysV documentation.

shmget KEY,SIZE,FLAGS

Calls the System V IPC function shmget. Returns the shared memory segment id, or the undefined value if there is an error. See also SysV IPC in *perlipc* and IPC::SysV documentation.

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, or false if there is an error. shmread() taints the variable. See also SysV IPC in *perlipc*, `IPC::SysV` documentation, and the `IPC::Shareable` module from CPAN.

shutdown SOCKET,HOW

Shuts down a socket connection in the manner indicated by HOW, which has the same interpretation as in the system call 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.

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 EXPR seconds, or forever if no EXPR. May be interrupted if the process receives a signal such as `SIGALRM`. Returns the number of seconds actually slept. 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, you may use Perl's `syscall` interface to access setitimer(2) if your system supports it, or else see select above. The Time::HiRes module (from CPAN, and starting from Perl 5.8 part of the standard distribution) may also help.

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 system call of the same name. You should `use Socket` first to get the proper definitions imported. See the examples in Sockets: Client/Server Communication in *perlipc*.

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 $^F. See `$^F` in *perlvar*.

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 system call of the same name. If unimplemented, yields a fatal error. 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 $^F. See `$^F` in *perlvar*.

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 *perlipc* 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.

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 counter-productive to declare $a and $b as lexicals.

In either case, the subroutine may not be recursive. The values to be compared are always passed by reference, so don't modify them.

You also cannot exit out of the sort block or subroutine using any of the loop control operators described in *perlsyn* or with `goto`.

When `use locale` is in effect, `sort LIST` sorts LIST according to the current collation locale. See *perllocale*.

Perl 5.6 and earlier used a quicksort algorithm to implement sort. That algorithm was not stable, and *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 {uc($a) cmp uc($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

@new = sort {
    ($b =~ /=(\d+)/)[0] <=> ($a =~ /=(\d+)/)[0]
                        ||
                uc($a)  cmp  uc($b)
} @old;

# same thing, but much more efficiently;
# we'll build auxiliary indices instead
# for speed
@nums = @caps = ();
for (@old) {
    push @nums, /=(\d+)/;
    push @caps, uc($_);
}

@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+)/, uc($_)] } @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;
```

If you're using strict, you *must not* declare $a and $b as lexicals. They are package globals. That means 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), and because `sort` will trigger a fatal error unless the result of a comparison is defined, when sorting with a comparison function like `$a <=> $b`, be careful about lists that might contain a `NaN`. The following example takes advantage of the fact that `NaN != NaN` to eliminate any `NaN`s from the input.

```
@result = sort { $a <=> $b } grep { $_ == $_ } @input;
```

splice ARRAY,OFFSET,LENGTH,LIST

splice ARRAY,OFFSET,LENGTH

splice ARRAY,OFFSET

splice ARRAY

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, perl issues a warning, and splices at the end of the array.

The following equivalences hold (assuming `$[ == 0 and $#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)
```

Example, assuming array lengths are passed before arrays:

```
    sub aeq {   # compare two list values
        my(@a) = splice(@_,0,shift);
        my(@b) = splice(@_,0,shift);
        return 0 unless @a == @b;       # same len?
        while (@a) {
            return 0 if pop(@a) ne pop(@b);
        }
        return 1;
    }
    if (&aeq($len,@foo[1..$len],0+@bar,@bar)) { ... }
```

split /PATTERN/,EXPR,LIMIT

split /PATTERN/,EXPR

split /PATTERN/

split

Splits the string EXPR into a list of strings and returns that list. By default, empty leading fields are preserved, and empty trailing ones are deleted. (If all fields are empty, they are considered to be trailing.)

In scalar context, returns the number of fields found and splits into the `@_` array. Use of split in scalar context is deprecated, however, because it clobbers your subroutine arguments.

If EXPR is omitted, splits the `$_` string. If PATTERN is also omitted, splits on whitespace (after skipping any leading whitespace). Anything matching PATTERN is taken to be a delimiter separating the fields. (Note that the delimiter may be longer than one character.)

If LIMIT is specified and positive, it represents the maximum number of fields the EXPR will be split into, though the actual number of fields returned depends on the number of times PATTERN matches within EXPR. If LIMIT is unspecified or zero, trailing null fields are stripped (which potential users of `pop` would do well to remember). If LIMIT is negative, it is treated as if an arbitrarily large LIMIT had been specified. Note that splitting an EXPR that evaluates to the empty string always returns the empty list, regardless of the LIMIT specified.

A pattern matching the null string (not to be confused with a null pattern `//`, which is just one member of the set of patterns matching a null string) will split the value of EXPR into separate characters at each point it matches that way. For example:

```
    print join(':', split(/ */, 'hi there'));
```

produces the output 'h:i:t:h:e:r:e'.

Using the empty pattern `//` specifically matches the null string, and is not be confused with the use of `//` to mean "the last successful pattern match".

Empty leading (or trailing) fields are produced when there are positive width matches at the beginning (or end) of the string; a zero-width match at the beginning (or end) of the string does not produce an empty field. For example:

```
   print join(':', split(/(?=\w)/, 'hi there!'));
```

produces the output 'h:i :t:h:e:r:e!'.

The LIMIT parameter can be used to split a line partially

```
    ($login, $passwd, $remainder) = split(/:/, $_, 3);
```

When assigning to a list, if LIMIT is omitted, or zero, Perl supplies a LIMIT one larger than the number of variables in the list, to avoid unnecessary work. For the list above LIMIT would have been 4 by default. In time critical applications it behooves you not to split into more fields than you really need.

If the PATTERN contains parentheses, additional list elements are created from each matching substring in the delimiter.

```
    split(/([,-])/, "1-10,20", 3);
```

produces the list value

```
    (1, '-', 10, ',', 20)
```

If you had the entire header of a normal Unix email message in $header, you could split it up into fields and their values this way:

```
    $header =~ s/\n\s+/ /g;  # fix continuation lines
    %hdrs   =  (UNIX_FROM => split /^(\S*?):\s*/m, $header);
```

The pattern `/PATTERN/` may be replaced with an expression to specify patterns that vary at runtime. (To do runtime compilation only once, use `/$variable/o`.)

As a special case, specifying a PATTERN of space (`' '`) will split on white space just as `split` with no arguments does. Thus, `split(' ')` can be used to emulate **awk**'s default behavior, whereas `split(/ /)` will give you as many null initial fields as there are leading spaces. A `split` on `/\s+/` is like a `split(' ')` except that any leading whitespace produces a null first field. A `split` with no arguments really does a `split(' ', $_)` internally.

A PATTERN of `/^/` is treated as if it were `/^/m`, since it isn't much use otherwise.

Example:

```
    open(PASSWD, '/etc/passwd');
    while (<PASSWD>) {
        chomp;
        ($login, $passwd, $uid, $gid,
         $gcos, $home, $shell) = split(/:/);
        #...
    }
```

As with regular pattern matching, any capturing parentheses that are not matched in a `split()` will be set to `undef` when returned:

```
    @fields = split /(A)|B/, "1A2B3";
    # @fields is (1, 'A', 2, undef, 3)
```

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`, but it doesn't use it (except for floating-point numbers, and even then only the standard modifiers are allowed). As a result, any non-standard extensions in your local `sprintf` are not available 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
   %p   a pointer (outputs the Perl value's address in hexadecimal)
   %n   special: *stores* the number of characters output so far
        into the next variable in the parameter list
```

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".

Between the `%` and the format letter, you may specify a number of 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. Eg:

```
  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 positive number with a space + prefix positive number with a plus sign - left-justify within the field 0 use zeros, not spaces, to right-justify # prefix non-zero octal with "0", non-zero hex with "0x", non-zero binary with "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 '<%#x>', 12;   # prints "<0xc>"
```

vector flag

The vector flag v, optionally specifying the join string to use. This flag tells perl to interpret the supplied string as a vector of integers, one for each character in the string, separated by a given string (a dot . by default). This can be useful for displaying ordinal values of characters in arbitrary strings:

```
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 eg *2$v:

```
printf '%*4$vX %*4$vX %*4$vX', @addr[1..3], ":";   # 3 IPv6 addresses
```

(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 (with eg *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, with the exception of 'g' and 'G', this specifies the number of decimal places to show (the default being 6), eg:

```
# 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 prior to the decimal point as well as after it, eg:

```
# 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:

```
printf '<%.6x>', 1;      # prints "<000001>"
printf '<%#.6x>', 1;     # prints "<0x000001>"
printf '<%-10.6x>', 1;   # prints "<000001    >"
```

For string conversions, specifying a precision truncates the string to fit in 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>"
```

You cannot currently get the precision from a specified number, but it is intended that this will be possible in the future using eg `.*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:

```
l           interpret integer as C type "long" or "unsigned long"
h           interpret integer as C type "short" or "unsigned short"
q, L or ll  interpret integer as C type "long long", "unsigned long long".
            or "quads" (typically 64-bit integers)
```

The last will produce errors if Perl does not understand "quads" in your installation. (This requires that either the platform natively supports quads or Perl was specifically compiled to support quads.) You can find out whether your Perl supports quads via *Config*:

```
use Config;
($Config{use64bitint} eq 'define' || $Config{longsize} >= 8) &&
        print "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;
$Config{d_longdbl} eq 'define' && print "long doubles\n";
```

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;
($Config{uselongdouble} eq 'define') &&
        print "long doubles by default\n";
```

It can also be the case 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 it is supported for compatibility with XS code; it means 'use the standard size for a Perl integer (or floating-point number)', which is already the default for Perl code.

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 in which they appear in the format specification *before* the value to format. Where an argument is specified using 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 in any case).

So:

```
printf '<%*.*s>', $a, $b, $c;
```

would use `$a` for the width, `$b` for the precision and `$c` as the value to format, while:

```
print '<%*1$.*s>', $a, $b;
```

would use `$a` for the width and the precision, and `$b` as the value to format.

Here are some more examples - beware that when using an explicit index, the `$` may need to be escaped:

```
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` is in effect, the character used for the decimal point in formatted real numbers is affected by the LC_NUMERIC locale. See *perllocale*.

sqrt EXPR

sqrt

Return the square root of EXPR. If EXPR is omitted, returns square root of `$_`. Only works on non-negative operands, unless you've loaded the standard Math::Complex module.

```
use Math::Complex;
print sqrt(-2);    # prints 1.4142135623731i
```

srand EXPR

srand

Sets 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.

If srand() is not called explicitly, it is called implicitly at the first use of the `rand` operator. However, this was not the case in versions of Perl before 5.004, so if your script will run under older Perl versions, it should call `srand`.

Most programs won't even call srand() at all, except those that need a cryptographically-strong starting point rather than the generally acceptable default, which is based on time of day, process ID, and memory allocation, or the */dev/urandom* device, if available.

You can call srand($seed) with the same $seed to reproduce the *same* sequence from rand(), but this is usually reserved for generating predictable results for testing or debugging. Otherwise, don't call srand() more than once in your program.

Do **not** call srand() (i.e. without an argument) more than once in a script. 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.

In versions of Perl prior to 5.004 the default seed was just the current `time`. This isn't a particularly good seed, so many old programs supply their own seed value (often `time ^ $$` or `time ^ ($$ + ($$ << 15))`), but that isn't necessary any more.

Note that you need something much more random than the default seed for cryptographic purposes. Checksumming the compressed output of one or more rapidly changing operating system status programs is the usual method. For example:

```
srand (time ^ $$ ^ unpack "%L*", `ps axww | gzip`);
```

If you're particularly concerned with this, see the `Math::TrulyRandom` module in CPAN.

Frequently called programs (like CGI scripts) that simply use

```
    time ^ $$
```

for a seed can fall prey to the mathematical property that

```
    a^b == (a+1)^(b+1)
```

one-third of the time. So don't do that.

stat FILEHANDLE

stat EXPR

stat

Returns a 13-element list giving the status info for a file, either the file opened via FILEHANDLE, or named by EXPR. If EXPR is omitted, it stats `$_`. Returns a null list if the 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 block size for file system I/O
 12 blocks   actual number of blocks allocated
```

(The epoch was at 00:00 January 1, 1970 GMT.)

(*) The ctime field is non-portable. In particular, you cannot expect it to be a "creation time", see Files and Filesystems in *perlport* 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;
    $is_setgid     =  S_ISDIR($mode);
```

You could write the last two using the -u and -d operators. The 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 necessarily all are available on your system.

    S_IFREG S_IFDIR S_IFLNK S_IFBLK S_ISCHR S_IFIFO S_IFSOCK S_IFWHT S_ENFMT

    # The following are compatibility aliases for S_IRUSR, S_IWUSR, 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 e.g. 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.

study SCALAR

study

Takes 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 on, and on 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 runs faster. Those loops which scan for many short constant strings (including the constant parts of more complex patterns) will benefit most. You may have only one `study` active at a time–if you study a different scalar the first is "unstudied". (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 those 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 very 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, and does actually return a value: the CODE ref of the closure you just created.

See *perlsub* and *perlref* for details about subroutines and references, and *attributes* and *Attribute::Handlers* for more information about attributes.

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 `0`, or whatever you've set `$[` to (but don't do that). If OFFSET is negative (or more precisely, less than `$[`), starts that far from the end of the string. If LENGTH is omitted, returns everything to the end of the string. If LENGTH is negative, leaves that many characters off the end of the string.

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 is a fatal error. 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';         # fatal error
```

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().

If the lvalue returned by substr is used after the EXPR is changed in any way, the behaviour may not be as expected and is subject to change. This caveat includes code such as `print(substr($foo,$a,$b)=$bar)` or `(substr($foo,$a,$b)=$bar)=$fud` (where $foo is changed via the substring assignment, and then the substr is used again), or where a substr() is aliased via a `foreach` loop or passed as a parameter or a reference to it is taken and then the alias, parameter, or deref'd reference either is used after the original EXPR has been changed or is assigned to and then used a second time.

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, produces a fatal error at run time. To check for that, use eval:

```
$symlink_exists = eval { symlink("",""); 1 };
```

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, produces a fatal error. 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 system call, 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 and 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. There is no way to retrieve the file number of the other end. You can avoid this problem by using `pipe` instead.

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 name of the real filehandle wanted. This function calls the underlying operating system's `open` function with the parameters FILENAME, MODE, 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` 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: zero means read-only, one means write-only, and two means read/write. We know that these values do *not* work under OS/390 & VM/ESA Unix 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 rebuilding Perl to use the `sfio` library, or perhaps using the POSIX::open() function.

See *perlopentut* for a kinder, gentler explanation of opening files.

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 system call 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 very 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 binmode, open, and the `open` pragma, *open*.

sysseek FILEHANDLE,POSITION,WHENCE

Sets FILEHANDLE's system position in bytes using the system call 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 `:utf8` I/O layer), tell() will return byte offsets, not character offsets (because implementing that would render sysseek() very slow).

sysseek() bypasses normal buffered IO, so mixing this with reads (other than `sysread`, for example &gt;&lt 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 complete. 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.

Beginning with v5.6.0, 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 *perlport*). 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 exec. 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 'STRING' in *perlop*. Return value of -1 indicates a failure to start the program (inspect $! for the reason).

Like `exec`, `system` allows you to lie to a program about its name if you use the `system PROGRAM LIST` syntax. Again, see exec.

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: $?"
```

You can check all the failure possibilities 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;
}
```

or more portably by using the W*() calls of the POSIX extension; see *perlport* for more information.

When the arguments get executed via the system shell, results and return codes will be subject to its quirks and capabilities. See 'STRING' in *perlop* and exec for details.

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 the system call 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 buffers 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 available data 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. In the case the SCALAR is empty you can use OFFSET but only zero offset.

Note that if the filehandle has been marked as `:utf8`, Unicode characters are written instead of bytes (the LENGTH, OFFSET, and the return value of syswrite() are in UTF-8 encoded Unicode characters). The `:encoding(...)` layer implicitly introduces the `:utf8` layer. See binmode, open, 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 `: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() on a filehandle that has been opened using sysopen(), use sysseek() for that as described above. Why? Because sysopen() creates unbuffered, "raw", filehandles, while open() creates buffered filehandles. sysseek() make sense only on the first kind, tell() only makes sense on the second kind.

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. 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 `new` 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 `new` method 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
DESTROY this
UNTIE this
```

A class implementing a file handle 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 *perltie, 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 *perltie*, §**??**.

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, you may use either the Time::HiRes module (from CPAN, and starting from Perl 5.8 part of the standard distribution), or if you have gettimeofday(2), you may be able to use the `syscall` interface of Perl. See *perlfaq8* for details.

times

Returns a four-element list giving the user and system times, in seconds, for this process and the children of this process.

```
    ($user,$system,$cuser,$csystem) = times;
```

In scalar context, `times` returns `$user`.

tr///

The transliteration operator. Same as `y///`. See *perlop*.

truncate FILEHANDLE,LENGTH

truncate EXPR,LENGTH

Truncates the file opened on FILEHANDLE, or named by EXPR, to the specified length. Produces a fatal error if truncate isn't implemented on your system. Returns true if successful, the undefined value otherwise.

The behavior is undefined if LENGTH is greater than the length of the file.

uc EXPR

uc

Returns an uppercased version of EXPR. This is the internal function implementing the `\U` escape in double-quoted strings. Respects current LC_CTYPE locale if `use locale` in force. See *perllocale* and *perlunicode* for more details about locale and Unicode support. It does not attempt to do titlecase mapping on initial letters. See `ucfirst` for that.

If EXPR is omitted, uses `$_`.

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. Respects current LC_CTYPE locale if `use locale` in force. See *perllocale* and *perlunicode* for more details about locale and Unicode support.

If EXPR is omitted, uses `$_`.

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` (`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), produces a fatal error at run time. 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, if all you have is a string.

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 *delete*.) 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. Returns the number of files successfully deleted.

```
    $cnt = unlink 'a', 'b', 'c';
    unlink @goners;
    unlink <*.bak>;
```

Note: `unlink` will not 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. Use `rmdir` instead.

If LIST is omitted, uses `$_`.

unpack TEMPLATE,EXPR

`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.)

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 bytes 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("c",$_[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("%32C*",<>) % 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: in some cases, the repeat count is decreased, or `unpack()` will produce null strings or zeroes, or terminate with an error. If the input string is longer than one described by the TEMPLATE, the rest is ignored.

See pack for more examples and notes.

untie VARIABLE

Breaks the binding between a variable and a package. (See `tie`.) Has no effect if the variable is not tied.

unshift ARRAY,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.

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; import Module LIST; }
```

except that Module *must* be a bareword.

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. A fatal error is produced 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, which can do a similar check at run 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 which 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 have changed in incompatible ways from older versions of Perl. (We try not to do this more than we have to.)

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.

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).

There's a corresponding `no` command that unimports meanings imported by `use`, i.e., it calls `unimport Module LIST` instead of `import`.

```
no integer;
no strict 'refs';
no warnings;
```

See *perlmodlib* for a list of standard modules and pragmas. See *perlrun* 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 NUMERICAL 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*.

```
#!/usr/bin/perl
$atime = $mtime = time;
utime $atime, $mtime, @ARGV;
```

Since perl 5.7.2, if the first two elements of the list are `undef`, then the utime(2) function in the C library will be 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.)

```
utime undef, undef, @ARGV;
```

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.

Note that only passing one of the first two elements as `undef` will be equivalent of passing it as 0 and will not have the same effect as described when they are both `undef`. This case will also trigger an uninitialized warning.

values HASH

Returns a list consisting of all the values of the named hash. (In a scalar context, returns the number of values.)

The values are returned in an apparently random order. The actual random order is subject to change in future versions of perl, but it is guaranteed to be the same order as either the `keys` or `each` function would produce on the same (unmodified) hash. Since Perl 5.8.1 the ordering is different even between different runs of Perl for security reasons (see Algorithmic Complexity Attacks in *perlsec*).

As a side effect, calling values() resets the HASH's internal iterator, see each. (In particular, calling values() in void context resets the iterator with no other overhead.)

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
```

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 §**??** 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).

The string should not contain any character with the value > 255 (which can only happen if you're using UTF-8 encoding). If it does, it will be treated as something which is not UTF-8 encoded. When the `vec` was assigned to, other parts of your program will also no longer consider the string to be UTF-8 encoded. In other words, if you do have such characters in your string, vec() will operate on the actual byte string, and not the conceptual character string.

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 Bitwise String Operators in *perlop*.

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 is run, the above example 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 the wait(2) system call 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 `$?`. Note that a return value of `-1` could mean that child processes are being automatically reaped, as described in *perlipc*.

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 `$?`. If you say

```
use POSIX ":sys_wait_h";
#...
do {
    $kid = waitpid(-1, WNOHANG);
} until $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) system calls. 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 *perlipc* for details, and for other examples.

wantarray

Returns true if the context of the currently executing subroutine or eval() block 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";
```

This function should have been named wantlist() instead.

warn LIST

Produces a message on STDERR just like `die`, but doesn't exit or throw an exception.

If LIST 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 make arrangements 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 *perlvar* 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, a special top-of-page format is used to format the new page header, and then the record is written. By default the top-of-page format is the name of the filehandle with "_TOP" appended, but it may be dynamically set to the format of your choice by assigning the name to the `$^` variable while the filehandle is selected. The number of lines remaining on the current page is in variable `$-`, 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 *perlform*.

Note that write is *not* the opposite of `read`. Unfortunately.

y///

The transliteration operator. Same as `tr///`. See *perlop*.

Chapter 31

perlopentut

Tutorial on opening things in Perl

31.1 DESCRIPTION

Perl has two simple, built-in ways to open files: the shell way for convenience, and the C way for precision. The shell way also has 2- and 3-argument forms, which have different semantics for handling the filename. The choice is yours.

31.2 Open à la shell

Perl's `open` function was designed to mimic the way command-line redirection in the shell works. Here are some basic examples from the shell:

```
    $ myprogram file1 file2 file3
    $ myprogram    <  inputfile
    $ myprogram    >  outputfile
    $ myprogram    >> outputfile
    $ myprogram    |  otherprogram
    $ otherprogram |  myprogram
```

And here are some more advanced examples:

```
    $ otherprogram      | myprogram f1 - f2
    $ otherprogram 2>&1 | myprogram -
    $ myprogram     <&3
    $ myprogram     >&4
```

Programmers accustomed to constructs like those above can take comfort in learning that Perl directly supports these familiar constructs using virtually the same syntax as the shell.

31.2.1 Simple Opens

The `open` function takes two arguments: the first is a filehandle, and the second is a single string comprising both what to open and how to open it. `open` returns true when it works, and when it fails, returns a false value and sets the special variable `$!` to reflect the system error. If the filehandle was previously opened, it will be implicitly closed first.

For example:

```
open(INFO,      "datafile") || die("can't open datafile: $!");
open(INFO,   "<  datafile") || die("can't open datafile: $!");
open(RESULTS,">  runstats") || die("can't open runstats: $!");
open(LOG,    ">> logfile ") || die("can't open logfile:  $!");
```

If you prefer the low-punctuation version, you could write that this way:

```
open INFO,   "<  datafile"  or die "can't open datafile: $!";
open RESULTS,">  runstats"  or die "can't open runstats: $!";
open LOG,    ">> logfile "  or die "can't open logfile:  $!";
```

A few things to notice. First, the leading less-than is optional. If omitted, Perl assumes that you want to open the file for reading.

Note also that the first example uses the || logical operator, and the second uses `or`, which has lower precedence. Using || in the latter examples would effectively mean

```
open INFO, ( "<  datafile"  || die "can't open datafile: $!" );
```

which is definitely not what you want.

The other important thing to notice is that, just as in the shell, any white space before or after the filename is ignored. This is good, because you wouldn't want these to do different things:

```
open INFO,   "<datafile"
open INFO,   "< datafile"
open INFO,   "<  datafile"
```

Ignoring surrounding whitespace also helps for when you read a filename in from a different file, and forget to trim it before opening:

```
$filename = <INFO>;         # oops, \n still there
open(EXTRA, "< $filename") || die "can't open $filename: $!";
```

This is not a bug, but a feature. Because `open` mimics the shell in its style of using redirection arrows to specify how to open the file, it also does so with respect to extra white space around the filename itself as well. For accessing files with naughty names, see §31.4.2.

There is also a 3-argument version of `open`, which lets you put the special redirection characters into their own argument:

```
open( INFO, ">", $datafile ) || die "Can't create $datafile: $!";
```

In this case, the filename to open is the actual string in `$datafile`, so you don't have to worry about `$datafile` containing characters that might influence the open mode, or whitespace at the beginning of the filename that would be absorbed in the 2-argument version. Also, any reduction of unnecessary string interpolation is a good thing.

31.2.2 Indirect Filehandles

`open`'s first argument can be a reference to a filehandle. As of perl 5.6.0, if the argument is uninitialized, Perl will automatically create a filehandle and put a reference to it in the first argument, like so:

```
open( my $in, $infile )   or die "Couldn't read $infile: $!";
while ( <$in> ) {
    # do something with $_
}
close $in;
```

Indirect filehandles make namespace management easier. Since filehandles are global to the current package, two subroutines trying to open `INFILE` will clash. With two functions opening indirect filehandles like `my $infile`, there's no clash and no need to worry about future conflicts.

Another convenient behavior is that an indirect filehandle automatically closes when it goes out of scope or when you undefine it:

```
    sub firstline {
        open( my $in, shift ) && return scalar <$in>;
        # no close() required
    }
```

31.2.3 Pipe Opens

In C, when you want to open a file using the standard I/O library, you use the `fopen` function, but when opening a pipe, you use the `popen` function. But in the shell, you just use a different redirection character. That's also the case for Perl. The `open` call remains the same–just its argument differs.

If the leading character is a pipe symbol, `open` starts up a new command and opens a write-only filehandle leading into that command. This lets you write into that handle and have what you write show up on that command's standard input. For example:

```
    open(PRINTER, "| lpr -Plp1")    || die "can't run lpr: $!";
    print PRINTER "stuff\n";
    close(PRINTER)                  || die "can't close lpr: $!";
```

If the trailing character is a pipe, you start up a new command and open a read-only filehandle leading out of that command. This lets whatever that command writes to its standard output show up on your handle for reading. For example:

```
    open(NET, "netstat -i -n |")    || die "can't fork netstat: $!";
    while (<NET>) { }               # do something with input
    close(NET)                      || die "can't close netstat: $!";
```

What happens if you try to open a pipe to or from a non-existent command? If possible, Perl will detect the failure and set `$!` as usual. But if the command contains special shell characters, such as > or *, called 'metacharacters', Perl does not execute the command directly. Instead, Perl runs the shell, which then tries to run the command. This means that it's the shell that gets the error indication. In such a case, the `open` call will only indicate failure if Perl can't even run the shell. See How can I capture STDERR from an external command? in *perlfaq8* to see how to cope with this. There's also an explanation in *perlipc*.

If you would like to open a bidirectional pipe, the IPC::Open2 library will handle this for you. Check out Bidirectional Communication with Another Process in *perlipc*

31.2.4 The Minus File

Again following the lead of the standard shell utilities, Perl's `open` function treats a file whose name is a single minus, "-", in a special way. If you open minus for reading, it really means to access the standard input. If you open minus for writing, it really means to access the standard output.

If minus can be used as the default input or default output, what happens if you open a pipe into or out of minus? What's the default command it would run? The same script as you're currently running! This is actually a stealth `fork` hidden inside an `open` call. See Safe Pipe Opens in *perlipc* for details.

31.2.5 Mixing Reads and Writes

It is possible to specify both read and write access. All you do is add a "+" symbol in front of the redirection. But as in the shell, using a less-than on a file never creates a new file; it only opens an existing one. On the other hand, using a greater-than always clobbers (truncates to zero length) an existing file, or creates a brand-new one if there isn't an old one. Adding a "+" for read-write doesn't affect whether it only works on existing files or always clobbers existing ones.

```
open(WTMP, "+< /usr/adm/wtmp")
    || die "can't open /usr/adm/wtmp: $!";

open(SCREEN, "+> lkscreen")
    || die "can't open lkscreen: $!";

open(LOGFILE, "+>> /var/log/applog"
    || die "can't open /var/log/applog: $!";
```

The first one won't create a new file, and the second one will always clobber an old one. The third one will create a new file if necessary and not clobber an old one, and it will allow you to read at any point in the file, but all writes will always go to the end. In short, the first case is substantially more common than the second and third cases, which are almost always wrong. (If you know C, the plus in Perl's `open` is historically derived from the one in C's fopen(3S), which it ultimately calls.)

In fact, when it comes to updating a file, unless you're working on a binary file as in the WTMP case above, you probably don't want to use this approach for updating. Instead, Perl's **-i** flag comes to the rescue. The following command takes all the C, C++, or yacc source or header files and changes all their foo's to bar's, leaving the old version in the original filename with a ".orig" tacked on the end:

```
$ perl -i.orig -pe 's/\bfoo\b/bar/g' *.[Cchy]
```

This is a short cut for some renaming games that are really the best way to update textfiles. See the second question in *perlfaq5* for more details.

31.2.6 Filters

One of the most common uses for `open` is one you never even notice. When you process the ARGV filehandle using `<ARGV>`, Perl actually does an implicit open on each file in @ARGV. Thus a program called like this:

```
$ myprogram file1 file2 file3
```

Can have all its files opened and processed one at a time using a construct no more complex than:

```
while (<>) {
    # do something with $_
}
```

If @ARGV is empty when the loop first begins, Perl pretends you've opened up minus, that is, the standard input. In fact, $ARGV, the currently open file during `<ARGV>` processing, is even set to "-" in these circumstances.

You are welcome to pre-process your @ARGV before starting the loop to make sure it's to your liking. One reason to do this might be to remove command options beginning with a minus. While you can always roll the simple ones by hand, the Getopts modules are good for this:

```
use Getopt::Std;

# -v, -D, -o ARG, sets $opt_v, $opt_D, $opt_o
getopts("vDo:");
```

```
    # -v, -D, -o ARG, sets $args{v}, $args{D}, $args{o}
    getopts("vDo:", \%args);
```

Or the standard Getopt::Long module to permit named arguments:

```
    use Getopt::Long;
    GetOptions( "verbose"  => \$verbose,        # --verbose
                "Debug"    => \$debug,          # --Debug
                "output=s" => \$output );
            # --output=somestring or --output somestring
```

Another reason for preprocessing arguments is to make an empty argument list default to all files:

```
    @ARGV = glob("*") unless @ARGV;
```

You could even filter out all but plain, text files. This is a bit silent, of course, and you might prefer to mention them on the way.

```
    @ARGV = grep { -f && -T } @ARGV;
```

If you're using the **-n** or **-p** command-line options, you should put changes to @ARGV in a `BEGIN{}` block.

Remember that a normal `open` has special properties, in that it might call fopen(3S) or it might called popen(3S), depending on what its argument looks like; that's why it's sometimes called "magic open". Here's an example:

```
    $pwdinfo = `domainname` =~ /^(\(none\))?$/
                    ? '< /etc/passwd'
                    : 'ypcat passwd |';

    open(PWD, $pwdinfo)
                or die "can't open $pwdinfo: $!";
```

This sort of thing also comes into play in filter processing. Because `<ARGV>` processing employs the normal, shell-style Perl `open`, it respects all the special things we've already seen:

```
    $ myprogram f1 "cmd1|" - f2 "cmd2|" f3 < tmpfile
```

That 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.

Yes, this also means that if you have files named "-" (and so on) in your directory, they won't be processed as literal files by `open`. You'll need to pass them as "./-", much as you would for the *rm* program, or you could use `sysopen` as described below.

One of the more interesting applications is to change files of a certain name into pipes. For example, to autoprocess gzipped or compressed files by decompressing them with *gzip*:

```
    @ARGV = map { /^\.(gz|Z)$/ ? "gzip -dc $_ |" : $_  } @ARGV;
```

Or, if you have the *GET* program installed from LWP, you can fetch URLs before processing them:

```
    @ARGV = map { m#^\w+://# ? "GET $_ |" : $_ } @ARGV;
```

It's not for nothing that this is called magic `<ARGV>`. Pretty nifty, eh?

31.3 Open à la C

If you want the convenience of the shell, then Perl's `open` is definitely the way to go. On the other hand, if you want finer precision than C's simplistic fopen(3S) provides you should look to Perl's `sysopen`, which is a direct hook into the open(2) system call. That does mean it's a bit more involved, but that's the price of precision.

`sysopen` takes 3 (or 4) arguments.

```
    sysopen HANDLE, PATH, FLAGS, [MASK]
```

The HANDLE argument is a filehandle just as with `open`. The PATH is a literal path, one that doesn't pay attention to any greater-thans or less-thans or pipes or minuses, nor ignore white space. If it's there, it's part of the path. The FLAGS argument contains one or more values derived from the Fcntl module that have been or'd together using the bitwise "|" operator. The final argument, the MASK, is optional; if present, it is combined with the user's current umask for the creation mode of the file. You should usually omit this.

Although the traditional values of read-only, write-only, and read-write are 0, 1, and 2 respectively, this is known not to hold true on some systems. Instead, it's best to load in the appropriate constants first from the Fcntl module, which supplies the following standard flags:

```
    O_RDONLY            Read only
    O_WRONLY            Write only
    O_RDWR              Read and write
    O_CREAT             Create the file if it doesn't exist
    O_EXCL              Fail if the file already exists
    O_APPEND            Append to the file
    O_TRUNC             Truncate the file
    O_NONBLOCK          Non-blocking access
```

Less common flags that are sometimes available on some operating systems include `O_BINARY`, `O_TEXT`, `O_SHLOCK`, `O_EXLOCK`, `O_DEFER`, `O_SYNC`, `O_ASYNC`, `O_DSYNC`, `O_RSYNC`, `O_NOCTTY`, `O_NDELAY` and `O_LARGEFILE`. Consult your open(2) manpage or its local equivalent for details. (Note: starting from Perl release 5.6 the `O_LARGEFILE` flag, if available, is automatically added to the sysopen() flags because large files are the default.)

Here's how to use `sysopen` to emulate the simple `open` calls we had before. We'll omit the || `die $!` checks for clarity, but make sure you always check the return values in real code. These aren't quite the same, since `open` will trim leading and trailing white space, but you'll get the idea.

To open a file for reading:

```
    open(FH, "< $path");
    sysopen(FH, $path, O_RDONLY);
```

To open a file for writing, creating a new file if needed or else truncating an old file:

```
    open(FH, "> $path");
    sysopen(FH, $path, O_WRONLY | O_TRUNC | O_CREAT);
```

To open a file for appending, creating one if necessary:

```
    open(FH, ">> $path");
    sysopen(FH, $path, O_WRONLY | O_APPEND | O_CREAT);
```

To open a file for update, where the file must already exist:

```
    open(FH, "+< $path");
    sysopen(FH, $path, O_RDWR);
```

And here are things you can do with sysopen that you cannot do with a regular open. As you'll see, it's just a matter of controlling the flags in the third argument.

To open a file for writing, creating a new file which must not previously exist:

```
    sysopen(FH, $path, O_WRONLY | O_EXCL | O_CREAT);
```

To open a file for appending, where that file must already exist:

```
    sysopen(FH, $path, O_WRONLY | O_APPEND);
```

To open a file for update, creating a new file if necessary:

```
    sysopen(FH, $path, O_RDWR | O_CREAT);
```

To open a file for update, where that file must not already exist:

```
    sysopen(FH, $path, O_RDWR | O_EXCL | O_CREAT);
```

To open a file without blocking, creating one if necessary:

```
    sysopen(FH, $path, O_WRONLY | O_NONBLOCK | O_CREAT);
```

31.3.1 Permissions à la mode

If you omit the MASK argument to sysopen, Perl uses the octal value 0666. The normal MASK to use for executables and directories should be 0777, and for anything else, 0666.

Why so permissive? Well, it isn't really. The MASK will be modified by your process's current umask. A umask is a number representing *disabled* permissions bits; that is, bits that will not be turned on in the created files' permissions field.

For example, if your umask were 027, then the 020 part would disable the group from writing, and the 007 part would disable others from reading, writing, or executing. Under these conditions, passing sysopen 0666 would create a file with mode 0640, since 0666 & ~027 is 0640.

You should seldom use the MASK argument to sysopen(). That takes away the user's freedom to choose what permission new files will have. Denying choice is almost always a bad thing. One exception would be for cases where sensitive or private data is being stored, such as with mail folders, cookie files, and internal temporary files.

31.4 Obscure Open Tricks

31.4.1 Re-Opening Files (dups)

Sometimes you already have a filehandle open, and want to make another handle that's a duplicate of the first one. In the shell, we place an ampersand in front of a file descriptor number when doing redirections. For example, 2>&1 makes descriptor 2 (that's STDERR in Perl) be redirected into descriptor 1 (which is usually Perl's STDOUT). The same is essentially true in Perl: a filename that begins with an ampersand is treated instead as a file descriptor if a number, or as a filehandle if a string.

```
    open(SAVEOUT, ">&SAVEERR") || die "couldn't dup SAVEERR: $!";
    open(MHCONTEXT, "<&4")     || die "couldn't dup fd4: $!";
```

That means that if a function is expecting a filename, but you don't want to give it a filename because you already have the file open, you can just pass the filehandle with a leading ampersand. It's best to use a fully qualified handle though, just in case the function happens to be in a different package:

```
    somefunction("&main::LOGFILE");
```

This way if somefunction() is planning on opening its argument, it can just use the already opened handle. This differs from passing a handle, because with a handle, you don't open the file. Here you have something you can pass to open.

If you have one of those tricky, newfangled I/O objects that the C++ folks are raving about, then this doesn't work because those aren't a proper filehandle in the native Perl sense. You'll have to use fileno() to pull out the proper descriptor number, assuming you can:

```
    use IO::Socket;
    $handle = IO::Socket::INET->new("www.perl.com:80");
    $fd = $handle->fileno;
    somefunction("&$fd");  # not an indirect function call
```

It can be easier (and certainly will be faster) just to use real filehandles though:

```
    use IO::Socket;
    local *REMOTE = IO::Socket::INET->new("www.perl.com:80");
    die "can't connect" unless defined(fileno(REMOTE));
    somefunction("&main::REMOTE");
```

If the filehandle or descriptor number is preceded not just with a simple "&" but rather with a "&=" combination, then Perl will not create a completely new descriptor opened to the same place using the dup(2) system call. Instead, it will just make something of an alias to the existing one using the fdopen(3S) library call This is slightly more parsimonious of systems resources, although this is less a concern these days. Here's an example of that:

```
    $fd = $ENV{"MHCONTEXTFD"};
    open(MHCONTEXT, "<&=$fd")   or die "couldn't fdopen $fd: $!";
```

If you're using magic `<ARGV>`, you could even pass in as a command line argument in @ARGV something like `"<&=$MHCONTEXTFD"`, but we've never seen anyone actually do this.

31.4.2 Dispelling the Dweomer

Perl is more of a DWIMmer language than something like Java–where DWIM is an acronym for "do what I mean". But this principle sometimes leads to more hidden magic than one knows what to do with. In this way, Perl is also filled with *dweomer*, an obscure word meaning an enchantment. Sometimes, Perl's DWIMmer is just too much like dweomer for comfort.

If magic `open` is a bit too magical for you, you don't have to turn to `sysopen`. To open a file with arbitrary weird characters in it, it's necessary to protect any leading and trailing whitespace. Leading whitespace is protected by inserting a `"./"` in front of a filename that starts with whitespace. Trailing whitespace is protected by appending an ASCII NUL byte (`"\0"`) at the end of the string.

```
    $file =~ s#^(\s)#./$1#;
    open(FH, "< $file\0")   || die "can't open $file: $!";
```

This assumes, of course, that your system considers dot the current working directory, slash the directory separator, and disallows ASCII NULs within a valid filename. Most systems follow these conventions, including all POSIX systems as well as proprietary Microsoft systems. The only vaguely popular system that doesn't work this way is the proprietary Macintosh system, which uses a colon where the rest of us use a slash. Maybe `sysopen` isn't such a bad idea after all.

If you want to use `<ARGV>` processing in a totally boring and non-magical way, you could do this first:

```
#   "Sam sat on the ground and put his head in his hands.
#   'I wish I had never come here, and I don't want to see
#   no more magic,' he said, and fell silent."
for (@ARGV) {
    s#^([^./])#./$1#;
    $_ .= "\0";
}
while (<>) {
    # now process $_
}
```

But be warned that users will not appreciate being unable to use "-" to mean standard input, per the standard convention.

31.4.3 Paths as Opens

You've probably noticed how Perl's `warn` and `die` functions can produce messages like:

```
Some warning at scriptname line 29, <FH> line 7.
```

That's because you opened a filehandle FH, and had read in seven records from it. But what was the name of the file, rather than the handle?

If you aren't running with `strict refs`, or if you've turned them off temporarily, then all you have to do is this:

```
open($path, "< $path") || die "can't open $path: $!";
while (<$path>) {
    # whatever
}
```

Since you're using the pathname of the file as its handle, you'll get warnings more like

```
Some warning at scriptname line 29, </etc/motd> line 7.
```

31.4.4 Single Argument Open

Remember how we said that Perl's open took two arguments? That was a passive prevarication. You see, it can also take just one argument. If and only if the variable is a global variable, not a lexical, you can pass `open` just one argument, the filehandle, and it will get the path from the global scalar variable of the same name.

```
$FILE = "/etc/motd";
open FILE or die "can't open $FILE: $!";
while (<FILE>) {
    # whatever
}
```

Why is this here? Someone has to cater to the hysterical porpoises. It's something that's been in Perl since the very beginning, if not before.

31.4.5 Playing with STDIN and STDOUT

One clever move with STDOUT is to explicitly close it when you're done with the program.

```
END { close(STDOUT) || die "can't close stdout: $!" }
```

If you don't do this, and your program fills up the disk partition due to a command line redirection, it won't report the error exit with a failure status.

You don't have to accept the STDIN and STDOUT you were given. You are welcome to reopen them if you'd like.

```
open(STDIN, "< datafile")
    || die "can't open datafile: $!";

open(STDOUT, "> output")
    || die "can't open output: $!";
```

And then these can be accessed directly or passed on to subprocesses. This makes it look as though the program were initially invoked with those redirections from the command line.

It's probably more interesting to connect these to pipes. For example:

```
$pager = $ENV{PAGER} || "(less || more)";
open(STDOUT, "| $pager")
    || die "can't fork a pager: $!";
```

This makes it appear as though your program were called with its stdout already piped into your pager. You can also use this kind of thing in conjunction with an implicit fork to yourself. You might do this if you would rather handle the post processing in your own program, just in a different process:

```
head(100);
while (<>) {
    print;
}

sub head {
    my $lines = shift || 20;
    return if $pid = open(STDOUT, "|-");       # return if parent
    die "cannot fork: $!" unless defined $pid;
    while (<STDIN>) {
        last if --$lines < 0;
        print;
    }
    exit;
}
```

This technique can be applied to repeatedly push as many filters on your output stream as you wish.

31.5 Other I/O Issues

These topics aren't really arguments related to `open` or `sysopen`, but they do affect what you do with your open files.

31.5.1 Opening Non-File Files

When is a file not a file? Well, you could say when it exists but isn't a plain file. We'll check whether it's a symbolic link first, just in case.

```
    if (-l $file || ! -f _) {
        print "$file is not a plain file\n";
    }
```

What other kinds of files are there than, well, files? Directories, symbolic links, named pipes, Unix-domain sockets, and block and character devices. Those are all files, too–just not *plain* files. This isn't the same issue as being a text file. Not all text files are plain files. Not all plain files are text files. That's why there are separate `-f` and `-T` file tests.

To open a directory, you should use the `opendir` function, then process it with `readdir`, carefully restoring the directory name if necessary:

```
    opendir(DIR, $dirname) or die "can't opendir $dirname: $!";
    while (defined($file = readdir(DIR))) {
        # do something with "$dirname/$file"
    }
    closedir(DIR);
```

If you want to process directories recursively, it's better to use the File::Find module. For example, this prints out all files recursively and adds a slash to their names if the file is a directory.

```
    @ARGV = qw(.) unless @ARGV;
    use File::Find;
    find sub { print $File::Find::name, -d && '/', "\n" }, @ARGV;
```

This finds all bogus symbolic links beneath a particular directory:

```
    find sub { print "$File::Find::name\n" if -l && !-e }, $dir;
```

As you see, with symbolic links, you can just pretend that it is what it points to. Or, if you want to know *what* it points to, then `readlink` is called for:

```
    if (-l $file) {
        if (defined($whither = readlink($file))) {
            print "$file points to $whither\n";
        } else {
            print "$file points nowhere: $!\n";
        }
    }
```

31.5.2 Opening Named Pipes

Named pipes are a different matter. You pretend they're regular files, but their opens will normally block until there is both a reader and a writer. You can read more about them in Named Pipes in *perlipc*. Unix-domain sockets are rather different beasts as well; they're described in Unix-Domain TCP Clients and Servers in *perlipc*.

When it comes to opening devices, it can be easy and it can be tricky. We'll assume that if you're opening up a block device, you know what you're doing. The character devices are more interesting. These are typically used for modems, mice, and some kinds of printers. This is described in How do I read and write the serial port? in *perlfaq8* It's often enough to open them carefully:

```
    sysopen(TTYIN, "/dev/ttyS1", O_RDWR | O_NDELAY | O_NOCTTY)
                # (O_NOCTTY no longer needed on POSIX systems)
        or die "can't open /dev/ttyS1: $!";
    open(TTYOUT, "+>&TTYIN")
        or die "can't dup TTYIN: $!";

    $ofh = select(TTYOUT); $| = 1; select($ofh);

    print TTYOUT "+++at\015";
    $answer = <TTYIN>;
```

With descriptors that you haven't opened using `sysopen`, such as sockets, you can set them to be non-blocking using `fcntl`:

```
    use Fcntl;
    my $old_flags = fcntl($handle, F_GETFL, 0)
        or die "can't get flags: $!";
    fcntl($handle, F_SETFL, $old_flags | O_NONBLOCK)
        or die "can't set non blocking: $!";
```

Rather than losing yourself in a morass of twisting, turning `ioctl`s, all dissimilar, if you're going to manipulate ttys, it's best to make calls out to the stty(1) program if you have it, or else use the portable POSIX interface. To figure this all out, you'll need to read the termios(3) manpage, which describes the POSIX interface to tty devices, and then *POSIX*, which describes Perl's interface to POSIX. There are also some high-level modules on CPAN that can help you with these games. Check out Term::ReadKey and Term::ReadLine.

31.5.3 Opening Sockets

What else can you open? To open a connection using sockets, you won't use one of Perl's two open functions. See Sockets: Client/Server Communication in *perlipc* for that. Here's an example. Once you have it, you can use FH as a bidirectional filehandle.

```
    use IO::Socket;
    local *FH = IO::Socket::INET->new("www.perl.com:80");
```

For opening up a URL, the LWP modules from CPAN are just what the doctor ordered. There's no filehandle interface, but it's still easy to get the contents of a document:

```
    use LWP::Simple;
    $doc = get('http://www.linpro.no/lwp/');
```

31.5.4 Binary Files

On certain legacy systems with what could charitably be called terminally convoluted (some would say broken) I/O models, a file isn't a file–at least, not with respect to the C standard I/O library. On these old systems whose libraries (but not kernels) distinguish between text and binary streams, to get files to behave properly you'll have to bend over backwards to avoid nasty problems. On such infelicitous systems, sockets and pipes are already opened in binary mode, and there is currently no way to turn that off. With files, you have more options.

Another option is to use the `binmode` function on the appropriate handles before doing regular I/O on them:

```
    binmode(STDIN);
    binmode(STDOUT);
    while (<STDIN>) { print }
```

Passing sysopen a non-standard flag option will also open the file in binary mode on those systems that support it. This is the equivalent of opening the file normally, then calling binmode on the handle.

```
    sysopen(BINDAT, "records.data", O_RDWR | O_BINARY)
        || die "can't open records.data: $!";
```

Now you can use read and print on that handle without worrying about the non-standard system I/O library breaking your data. It's not a pretty picture, but then, legacy systems seldom are. CP/M will be with us until the end of days, and after.

On systems with exotic I/O systems, it turns out that, astonishingly enough, even unbuffered I/O using sysread and syswrite might do sneaky data mutilation behind your back.

```
    while (sysread(WHENCE, $buf, 1024)) {
        syswrite(WHITHER, $buf, length($buf));
    }
```

Depending on the vicissitudes of your runtime system, even these calls may need binmode or O_BINARY first. Systems known to be free of such difficulties include Unix, the Mac OS, Plan 9, and Inferno.

31.5.5 File Locking

In a multitasking environment, you may need to be careful not to collide with other processes who want to do I/O on the same files as you are working on. You'll often need shared or exclusive locks on files for reading and writing respectively. You might just pretend that only exclusive locks exist.

Never use the existence of a file -e $file as a locking indication, because there is a race condition between the test for the existence of the file and its creation. It's possible for another process to create a file in the slice of time between your existence check and your attempt to create the file. Atomicity is critical.

Perl's most portable locking interface is via the flock function, whose simplicity is emulated on systems that don't directly support it such as SysV or Windows. The underlying semantics may affect how it all works, so you should learn how flock is implemented on your system's port of Perl.

File locking *does not* lock out another process that would like to do I/O. A file lock only locks out others trying to get a lock, not processes trying to do I/O. Because locks are advisory, if one process uses locking and another doesn't, all bets are off.

By default, the flock call will block until a lock is granted. A request for a shared lock will be granted as soon as there is no exclusive locker. A request for an exclusive lock will be granted as soon as there is no locker of any kind. Locks are on file descriptors, not file names. You can't lock a file until you open it, and you can't hold on to a lock once the file has been closed.

Here's how to get a blocking shared lock on a file, typically used for reading:

```
    use 5.004;
    use Fcntl qw(:DEFAULT :flock);
    open(FH, "< filename")  or die "can't open filename: $!";
    flock(FH, LOCK_SH)      or die "can't lock filename: $!";
    # now read from FH
```

You can get a non-blocking lock by using LOCK_NB.

```
    flock(FH, LOCK_SH | LOCK_NB)
        or die "can't lock filename: $!";
```

This can be useful for producing more user-friendly behaviour by warning if you're going to be blocking:

```
use 5.004;
use Fcntl qw(:DEFAULT :flock);
open(FH, "< filename")  or die "can't open filename: $!";
unless (flock(FH, LOCK_SH | LOCK_NB)) {
    $| = 1;
    print "Waiting for lock...";
    flock(FH, LOCK_SH)  or die "can't lock filename: $!";
    print "got it.\n"
}
# now read from FH
```

To get an exclusive lock, typically used for writing, you have to be careful. We `sysopen` the file so it can be locked before it gets emptied. You can get a nonblocking version using `LOCK_EX | LOCK_NB`.

```
use 5.004;
use Fcntl qw(:DEFAULT :flock);
sysopen(FH, "filename", O_WRONLY | O_CREAT)
    or die "can't open filename: $!";
flock(FH, LOCK_EX)
    or die "can't lock filename: $!";
truncate(FH, 0)
    or die "can't truncate filename: $!";
# now write to FH
```

Finally, due to the uncounted millions who cannot be dissuaded from wasting cycles on useless vanity devices called hit counters, here's how to increment a number in a file safely:

```
use Fcntl qw(:DEFAULT :flock);

sysopen(FH, "numfile", O_RDWR | O_CREAT)
    or die "can't open numfile: $!";
# autoflush FH
$ofh = select(FH); $| = 1; select ($ofh);
flock(FH, LOCK_EX)
    or die "can't write-lock numfile: $!";

$num = <FH> || 0;
seek(FH, 0, 0)
    or die "can't rewind numfile : $!";
print FH $num+1, "\n"
    or die "can't write numfile: $!";

truncate(FH, tell(FH))
    or die "can't truncate numfile: $!";
close(FH)
    or die "can't close numfile: $!";
```

31.5.6 IO Layers

In Perl 5.8.0 a new I/O framework called "PerlIO" was introduced. This is a new "plumbing" for all the I/O happening in Perl; for the most part everything will work just as it did, but PerlIO also brought in some new features such as the ability to think of I/O as "layers". One I/O layer may in addition to just moving the data also do transformations on the data. Such transformations may include compression and decompression, encryption and decryption, and transforming between various character encodings.

Full discussion about the features of PerlIO is out of scope for this tutorial, but here is how to recognize the layers being used:

- The three-(or more)-argument form of `open` is being used and the second argument contains something else in addition to the usual `'<'`, `'>'`, `'>>'`, `'|'` and their variants, for example:

```
    open(my $fh, "<:utf8", $fn);
```

- The two-argument form of `binmode` is being used, for example

```
    binmode($fh, ":encoding(utf16)");
```

For more detailed discussion about PerlIO see *PerlIO*; for more detailed discussion about Unicode and I/O see *perluniintro*.

31.6 SEE ALSO

The `open` and `sysopen` functions in perlfunc(1); the system open(2), dup(2), fopen(3), and fdopen(3) manpages; the POSIX documentation.

31.7 AUTHOR and COPYRIGHT

31.8 HISTORY

First release: Sat Jan 9 08:09:11 MST 1999

Chapter 32

perlpacktut

Tutorial on `pack` and `unpack`

32.1 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.

32.2 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 $s = pack( 'H2' x 10, map { "3$_" } ( 0..9 ) );
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`.

32.3 Packing Text

Let's suppose you've got to read in a data file like this:

```
Date      |Description                | Income|Expenditure
01/24/2001 Ahmed'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 Ahmed'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 `pack` in *perlfunc* 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 Ahmed'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.

32.4 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.

32.4.1 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.

32.4.2 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 (signed) 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).

32.4.3 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 §32.7 before you really code this!)

32.4.4 Floating point Numbers

For packing floating point numbers you have the choice between the pack codes `f` and `d` which pack into (or unpack from) single-precision or double-precision representation as it is provided by your system. (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 ASCII representation, unless you're absolutely sure what's on the other end of the line.)

32.5 Exotic Templates

32.5.1 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 `vec` in *perlfunc*. 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.

32.5.2 Uuencoding

Another odd-man-out in the template alphabet is u, which packs an "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.

32.5.3 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 );
```

32.5.4 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 *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.

So what has this got to do with pack? Well, if you want to convert between a Unicode number and its UTF-8 representation you can do so by using template code U. As an example, let's produce the UTF-8 representation of the Euro currency symbol (code number 0x20AC):

```
$UTF8{Euro} = pack( 'U', 0x20AC );
```

Inspecting `$UTF8{Euro}` shows that it contains 3 bytes: "\xe2\x82\xac". The round trip can be completed with `unpack`:

```
$Unicode{Euro} = unpack( 'U', $UTF8{Euro} );
```

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 );
```

32.5.5 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 *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.

32.6 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 );
```

32.7 Lengths and Widths

32.7.1 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...

32.7.2 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.

32.7.3 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.

32.8 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.

32.8.1 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 );
```

32.8.2 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 );
```

32.8.3 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]

32.8.4 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 *perlfunc* explaining how to use `syscall` we can write this Perl function copying a file to standard output:

```
require 'syscall.ph';
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.

32.9 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" : '';
```

32.10 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' );
```

32.11 Authors

Simon Cozens and Wolfgang Laun.

Chapter 33

perlpod

The Plain Old Documentation format

33.1 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: §33.1.1, §33.1.2, and §33.1.3.

33.1.1 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`, hyperlinks, and more. Such codes are explained in the "§33.1.4" section, below.

33.1.2 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.

33.1.3 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

```
=head1 Heading Text
=head2 Heading Text
=head3 Heading Text
=head4 Heading Text
=over indentlevel
=item stuff
```

```
    =back
    =cut
    =pod
    =begin format
    =end format
    =for format text...
```

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. (Note that head3 and head4 are recent additions, not supported in older Pod translators.) The text in these heading commands can use formatting codes, as seen here:

```
  =head2 Possible Values for C<$/>
```

Such commands are explained in the "§33.1.4" 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 "§33.1.4" 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 inbetween 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 inbetween. (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 or Latin-1, then put a `=encoding` *`encodingname`* command 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. Examples:

```
=encoding utf8

=encoding koi8-r

=encoding ShiftJIS

=encoding big5
```

And don't forget, when using any command, that the command lasts 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.

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
```

33.1.4 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>` 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<SWITCH statements|perlsyn/"Basic BLOCKs and Switch Statements">`
- `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:...>`
  Links to an absolute URL. For example, `L<http://www.perl.org/>`. But note that there is no corresponding `L<text|scheme:...>` syntax, for various reasons.

`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 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. With the Pod formatters that are standard starting with perl5.5.660, 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>
```

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.

33.1.5 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.

33.1.6 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. Perl will ignore the Pod text. 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.

33.1.7 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.

33.2 SEE ALSO

perlpodspec, PODs: Embedded Documentation in *perlsyn*, *perlnewmod*, *perldoc*, *pod2html*, *pod2man*, *podchecker*.

33.3 AUTHOR

Larry Wall, Sean M. Burke

Chapter 34

perlpodspec

Plain Old Documentation: format specification and notes

34.1 DESCRIPTION

This document is detailed notes on the Pod markup language. Most people will only have to read perlpod 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.

34.2 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 wordcounting it, 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 data or verbatim paragraphs, if *identifier* doesn't begin with a colon. This is discussed in detail in the section About Data Paragraphs and "=begin/=end" Regions.

34.3 Pod Commands

This section is intended to supplement and clarify the discussion in Command Paragraph in *perlpod*. 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 §34.7 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 §34.7 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"

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 About Data Paragraphs and "=begin/=end" Regions.

It is advised that formatnames match the regexp `m/\A:?[-a-zA-Z0-9_]+\z/`. 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 About Data Paragraphs and "=begin/=end" Regions.

"=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 *Encoding* recognizes. (Encoding's list of supported encodings, in *Encoding::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 "=use utf8" line, and later on another "=use 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.

34.4 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?

X<C<chmod> 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.

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 Formatting Codes in *perlpod*.

`B<text>` – bold text

See the brief discussion in Formatting Codes in *perlpod*.

`C<code>` – code text

See the brief discussion in Formatting Codes in *perlpod*.

`F<filename>` – style for filenames

See the brief discussion in Formatting Codes in *perlpod*.

`X<topic name>` – an index entry

See the brief discussion in Formatting Codes in *perlpod*.

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 Formatting Codes in *perlpod*.

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 Formatting Codes in *perlpod*, and implementation details are discussed below, in §34.6. 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 Formatting Codes in *perlpod*, and several points in Notes on Implementing Pod Processors.

`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<lt>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.)

34.5 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 perlport.
- 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 Latin-1.

  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. If the file begins with the three literal byte values 0xEF 0xBB 0xBF, this is the BOM for UTF-8.

- A naive but sufficient heuristic 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 0xC0 - 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 Latin-1. 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.

- This document's requirements and suggestions about encodings do not apply to Pod processors running on non-ASCII platforms, notably EBCDIC platforms.

- 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 openquote 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>.

  Characters 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 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 (13, 13 10, or 10), and tab (9).

  Characters in the range 160-255 refer to Latin-1 characters (also defined there by Unicode, with the same meaning). Characters 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 (doublequote, "), "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 judgment 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.

34.6 About L<...> Codes

As you can tell from a glance at perlpod, 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 DESCRIPTION in *Getopt::Std*, "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" # orig. 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
```

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.

- Authors wanting to link to a particular (absolute) URL, must do so only with "L<scheme:...>" codes (like L<http://www.perl.org>), and must not attempt "L<Some Site Name|scheme:...>" codes. This restriction avoids many problems in parsing and rendering L<...> codes.

- 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 – no other formatting codes. 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. This syntax is no longer in the specification, and has been replaced by the `L<"section">` syntax (where the quotes were 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 may warn about this being deprecated syntax.

34.7 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 forseeable 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.

34.8 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" – i.e., 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.

34.9 SEE ALSO

perlpod, PODs: Embedded Documentation in *perlsyn*, *podchecker*

34.10 AUTHOR

Sean M. Burke

Chapter 35

perlrun

How to execute the Perl interpreter

35.1 SYNOPSIS

perl [**-sTtuUWX**] [**-hv**] [**-V**[:*configvar*]] [**-cw**] [**-d**[:*debugger*]] [**-D**[*number/list*]]
[**-pna**] [**-F***pattern*] [**-l**[*octal*]] [**-0**[*octal/hexadecimal*]] [**-I***dir*] [**-m**[**-**]*module*] [**-M**[**-**]*'module...'*] [**-P**] [**-S**]
[**-x**[*dir*]] [**-i**[*extension*]] [**-e** *'command'*] [**–**] [*programfile*] [*argument*]... [**-C [*number/list*]**]]>

35.2 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 *perldebug* 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** 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 Location of Perl.)
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 behavior 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 -*- -p
    eval 'exec perl -wS $0 ${1+"$@"}'
        if $running_under_some_shell;
```

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, perl5.005_57, you should place that directly in the #! line's path.

If the #! line does not contain the word "perl", 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.

35.2.1 #! 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.

Macintosh

A Macintosh perl program will have the appropriate Creator and Type, so that double-clicking them will invoke the perl application.

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** 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\""

# Macintosh
print "Hello world\n"
 (then Run "Myscript" or Shift-Command-R)

# 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.

Under the Macintosh, it depends which environment you are using. The MacPerl shell, or MPW, is much like Unix shells in its support for several quoting variants, except that it makes free use of the Macintosh's non-ASCII characters as control characters.

There is no general solution to all of this. It's just a mess.

35.2.2 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/perl5.00554
```

or if you just want to be running at least version, place a statement like this at the top of your program:

```
use 5.005_54;
```

35.2.3 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
```

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. The value 0777 will cause Perl to slurp files whole because there is no legal byte with that value.

If you want to specify any Unicode character, use the hexadecimal format: `-0xHHH...`, where the `H` are valid hexadecimal digits. (This means that you cannot use the `-x` with a directory name that consists of hexadecimal digits.)

-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**.

-C [*number/list*]

The `-C` flag controls some Unicode 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_TYPE, and LANG, in the order
               of decreasing precedence) -- if the variables indicate
               UTF-8, then the selected "IOEioA" are in effect
```

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) 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.

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 is variable is set during Perl startup and is thereafter read-only. If you want runtime effects, use the three-arg open() (see open in *perlfunc*), the two-arg binmode() (see binmode in *perlfunc*), 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".)

-c

causes Perl to check the syntax of the program and then exit without executing it. Actually, it *will* execute `BEGIN`, `CHECK`, and `use` blocks, because these are considered as occurring outside the execution of your program. `INIT` and `END` blocks, however, will be skipped.

-d

runs the program under the Perl debugger. See *perldebug*.

-d:*foo[=bar,baz]*

runs the program under the control of a debugging, profiling, or tracing module installed as Devel::foo. E.g., **-d:DProf** executes the program using the Devel::DProf profiler. As with the **-M** flag, options may be passed to the Devel::foo package where they will be received and interpreted by the Devel::foo::import routine. The comma-separated list of options must follow a = character. See *perldebug*.

-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 *perldebguts*.

As an alternative, specify a number instead of list of letters (e.g., **-D14** is equivalent to **-Dtls**):

```
        1  p  Tokenizing and parsing
        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, preprocessor command for -P, source file input state
      128  m  Memory allocation
      256  f  Format processing
      512  r  Regular expression parsing and execution
     1024  x  Syntax tree dump
     2048  u  Tainting checks
     4096     (Obsolete, previously used for LEAKTEST)
     8192  H  Hash dump -- usurps values()
```

```
    16384  X  Scratchpad allocation
    32768  D  Cleaning up
    65536  S  Thread synchronization
   131072  T  Tokenising
   262144  R  Include reference counts of dumped variables (eg when using -Ds)
   524288  J  Do not s,t,P-debug (Jump over) opcodes within package DB
  1048576  v  Verbose: use in conjunction with other flags
  2097152  C  Copy On Write
```

All these flags require **-DDEBUGGING** when you compile the Perl executable (but see *Devel::Peek*, *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 *perldebug* 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.

-F*pattern*

specifies the pattern to split on if **-a** is also in effect. The pattern may be surrounded by `//`, `""`, or ", otherwise it will be put in single quotes.

-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, no backup is made and the current file is overwritten.

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 eof in *perlfunc*).

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 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...
```

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`), and also tells the C preprocessor where to search for include files. The C preprocessor is invoked with **-P**; by default it searches /usr/include and /usr/lib/perl.

-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., `'-Mmodule 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 **-mmodule=foo,bar** or **-Mmodule=foo,bar** as a shortcut for `'-Mmodule qw(foo bar)'`. This avoids the need to use quotes when importing symbols. The actual code generated by **-Mmodule=foo,bar** is `use module split(/,/,q{foo,bar})`. Note that the = form removes the distinction between **-m** and **-M**.

-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** 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.

Here is an efficient way to delete all files that haven't been modifed 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**.

-P

NOTE: Use of -P is strongly discouraged because of its inherent problems, including poor portability.

This option causes your program to be run through the C preprocessor before compilation by Perl. Because both comments and **cpp** directives begin with the # character, you should avoid starting comments with any words recognized by the C preprocessor such as `"if"`, `"else"`, or `"define"`.

If you're considering using `-P`, you might also want to look at the Filter::cpp module from CPAN.

The problems of -P include, but are not limited to:

- The `#!` line is stripped, so any switches there don't apply.
- A `-P` on a `#!` line doesn't work.
- **All** lines that begin with (whitespace and) a # but do not look like cpp commands, are stripped, including anything inside Perl strings, regular expressions, and here-docs .
- In some platforms the C preprocessor knows too much: it knows about the C++ -style until-end-of-line comments starting with `"//"`. This will cause problems with common Perl constructs like

  ```
  s/foo//;
  ```

  because after -P this will became illegal code

  ```
  s/foo
  ```

  The workaround is to use some other quoting separator than `"/"`, like for example `"!"`:

```
        s!foo!!;
```

- It requires not only a working C preprocessor but also a working *sed*. If not on UNIX, you are probably out of luck on this.
- Script line numbers are not preserved.
- The `-x` does not work with `-P`.

-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 **–**). This means you can have switches with two leading dashes (**–help**). 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 **–help** creates the variable ${-help}, which is not compliant with `strict refs`.

-S

makes Perl use the PATH environment variable to search for the program (unless the name of the program contains directory 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 #!. 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 (i.e., 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 be controlled normally with `no warnings qw(taint)`.

NOTE: this is not a substitute for -T. This is meant only to be used 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

forces "taint" checks to be turned on 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 *perlsec* 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 obsolete 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.

This switch has been superseded in favor of the new Perl code generator backends to the compiler. See *B* and *B::Bytecode* for details.

-U

allows Perl to do unsafe operations. Currently the only "unsafe" operations are the unlinking of directories while running as superuser, and running setuid programs with fatal taint checks turned into warnings. Note that the **-w** switch (or the `$^W` variable) must be used 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:*name*

Prints to STDOUT the value of the named configuration variable(s), with multiples when your query looks like a regex. For example,

```
$ 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.

```
$ 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 that are mentioned only once and scalar variables that are used before being set, redefined subroutines, references to undefined filehandles or filehandles opened read-only that you are attempting to write on, values used as a number that doesn'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 internal `$^W` variable. You can disable or promote into fatal errors specific warnings using `__WARN__` hooks, as described in *perlvar* and warn in *perlfunc*. See also *perldiag* and *perltrap*. A new, fine-grained warning facility is also available if you want to manipulate entire classes of warnings; see *warnings* or *perllexwarn*.

-W

Enables all warnings regardless of `no warnings` or `$^W`. See *perllexwarn*.

-X

Disables all warnings regardless of `use warnings` or `$^W`. See *perllexwarn*.

-x

-x *directory*

tells Perl that the program is embedded in a larger chunk of unrelated ASCII 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. 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).

35.3 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 directories under the specified locations are automatically included if they exist. 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** switch was used), neither variable is used. The program should instead say:

```
    use lib "/my/directory";
```

PERL5OPT

Command-line options (switches). Switches in this variable are taken as if they were on every Perl command line. Only the **-[DIMUdmtw]** switches are allowed. When running taint checks (because the program was running setuid or setgid, or the **-T** switch was used), this variable is ignored. If PERL5OPT begins with **-T**, tainting will be enabled, and any subsequent options ignored.

PERLIO

A space (or colon) separated list of PerlIO layers. If perl is built to use PerlIO system for IO (the default) these layers effect perl's IO.

It is conventional to start layer names with a colon e.g. `:perlio` to emphasise 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 `:stdio`.

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 pragma" for how to add external encodings as defaults.

The layers that it makes sense to include in the PERLIO environment variable are briefly summarised below. For more details see *PerlIO*.

:bytes

A pseudolayer that turns *off* the `:utf8` flag 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 which implements "reading" of files by using `mmap()` to make (whole) 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 nitroglycerin.

: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 any translation. In particular CRLF translation, and/or :utf8 intuited from locale are disabled.

Unlike in the 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 PerlIO interface by wrapping system's ANSI C "stdio" library calls. The layer provides both buffering and IO. Note that `:stdio` layer does *not* do CRLF translation even if that is platforms normal behaviour. You will need a `:crlf` layer above it to do that.

:unix

Low level layer which calls `read`, `write` and `lseek` etc.

:utf8

A pseudolayer that turns on a flag on the layer below to tell perl that output should be in utf8 and that input should be regarded as already in utf8 form. May be useful in PERLIO environment variable to make UTF-8 the default. (To turn off that behaviour use `:bytes` layer.)

:win32

On Win32 platforms this *experimental* layer uses native "handle" IO rather than unix-like numeric file descriptor layer. Known to be buggy in this release.

On all platforms the default set of layers should give acceptable results.

For UNIX platforms that will equivalent of "unix perlio" or "stdio". Configure is setup to prefer "stdio" implementation if system's library provides for fast access to the buffer, otherwise it uses the "unix perlio" implementation.

On Win32 the default in this release is "unix crlf". Win32's "stdio" has a number of bugs/mis-features for perl IO which are somewhat C compiler vendor/version dependent. Using our own `crlf` layer as the buffer avoids those issues and makes things more uniform. The `crlf` layer provides CRLF to/from "\n" conversion as well as buffering.

This release uses `unix` as the bottom layer on Win32 and so still uses C compiler's numeric file descriptor routines. There is an experimental native `win32` layer which is expected to be enhanced and should eventually be the default under Win32.

PERLIO_DEBUG

If set to the name of a file or device then certain operations of PerlIO sub-system will be logged to that file (opened as append). Typical uses are UNIX:

```
PERLIO_DEBUG=/dev/tty perl script ...
```

and Win32 approximate equivalent:

```
set PERLIO_DEBUG=CON
perl script ...
```

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.

PERL5DB

The command used to load the debugger code. The default is:

```
    BEGIN { require 'perl5db.pl' }
```

PERL5SHELL (specific to the Win32 port)

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 to be space-separated. Precede any character that needs to be protected (like a space or backslash) with a 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).

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 causes memory statistics to be dumped after execution. If set to an integer greater than one, also causes memory statistics to be dumped after compilation.

PERL_DESTRUCT_LEVEL

Relevant only if your perl executable was built with **-DDEBUGGING**, this controls the behavior of global destruction of objects and other references. See PERL_DESTRUCT_LEVEL in *perlhack* for more information.

PERL_DL_NONLAZY

Set to one 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 it.

PERL_ENCODING

If using the `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.) Used to randomise Perl's internal hash function. To emulate the pre-5.8.1 behaviour, set to an integer (zero means exactly the same order as 5.8.0). "Pre-5.8.1" means, among other things, that hash keys will be ordered the same between different runs of Perl.

The default behaviour is to randomise unless the PERL_HASH_SEED is set. If Perl has been compiled with `-DUSE_HASH_SEED_EXPLICIT`, the default behaviour is **not** to randomise unless the PERL_HASH_SEED is set.

If PERL_HASH_SEED is unset or set to a non-numeric string, Perl uses the pseudorandom seed supplied by the operating system and libraries. This means that each different run of Perl will have a different ordering of the results of keys(), values(), and each().

Please note that 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 Algorithmic Complexity Attacks in *perlsec* and PERL_HASH_SEED_DEBUG for more information.

PERL_HASH_SEED_DEBUG

(Since Perl 5.8.1.) Set to one to display (to STDERR) the value of the hash seed at the beginning of execution. This, combined with PERL_HASH_SEED is intended to aid in debugging nondeterministic behavior caused by hash randomization.

Note that the hash seed is sensitive information: by knowing it one can craft a denial-of-service attack against Perl code, even remotely, see Algorithmic Complexity Attacks in *perlsec* for more information. **Do not disclose the hash seed** to people who don't need to know it. See also hash_seed() of *Hash::Util*.

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 *perlvms* and in *README.vms* in the Perl source distribution.

PERL_SIGNALS

In Perls 5.8.1 and later. If set to `unsafe` the pre-Perl-5.8.0 signals behaviour (immediate but unsafe) is restored. If set to `safe` the safe (or deferred) signals are used. See Deferred Signals (Safe signals) in *perlipc*.

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 *perllocale*.

Apart from these, Perl uses no other environment variables, except to make them available to the program being executed, and to child processes. 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)};
```

Chapter 36

perldiag

Various Perl diagnostics

36.1 DESCRIPTION

These messages are classified as follows (listed in increasing order of desperation):

```
    (W) A warning (optional).
    (D) A deprecation (optional).
    (S) A severe warning (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.

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 *perlvar*.

Default 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 eval in *perlfunc*. 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 accept in *perlfunc*.

Allocation too large: %lx

> (X) You can't allocate more than 64K on an MS-DOS machine.

'!' allowed only after types %s

> (F) The '!' is allowed in pack() or unpack() only after certain types. See pack in *perlfunc*.

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 Subroutine Attributes in *perlsub* 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

(W ambiguous)(S) 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.

'|' 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 grep in *perlfunc* and map in *perlfunc* for alternatives.

Args must match #! line

(F) The setuid emulator requires that the arguments Perl was invoked with match the arguments specified on the #! line. Since some systems impose a one-argument limit on the #! line, try combining switches; for example, turn `-w -U` into `-wU`.

Arg too short for msgsnd

(F) msgsnd() requires a string at least as long as sizeof(long).

%s argument is not a HASH or ARRAY element

(F) The argument to exists() must be a hash or array element, such as:

```
$foo{$bar}
$ref->{"susie"}[12]
```

%s 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"}
```

%s 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.

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.

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.

Array @%s missing the @ in argument %d of %s()

(D deprecated) Really old Perl let you omit the @ on array names in some spots. This is now heavily deprecated.

assertion botched: %s

(P) The malloc package that comes with Perl had an internal failure.

Assertion failed: file "%s"

(P) A general assertion failed. The file in question must be examined.

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.

A thread exited while %d threads were running

(W) When using threaded Perl, a thread (not necessarily the main thread) exited while there were still other threads running. Usually it's a good idea to first collect the return values of the created threads by joining them, and only then exit from the main thread. See *threads*.

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 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 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%lx

(P 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

(P 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

(W 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

(P internal) The reference counts got screwed up on symbol aliases.

Attempt to free unreferenced scalar

(W 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 join self

(F) You tried to join a thread from within itself, which is an impossible task. You may be joining the wrong thread, or you may need to move the join() to some other thread.

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 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 substr in *perlfunc*.

Bad arg length for %s, is %d, should be %s

(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.

Bad index while coercing array into hash

(F) The index looked up in the hash found as the 0'th element of a pseudo-hash is not legal. Index values must be at 1 or greater. See *perlref*.

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 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 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 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.

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.

\1 better written as $ 1

(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 *perlport* 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 bind in *perlfunc*.

binmode() on closed filehandle %s

(W unopened) You tried binmode() on a filehandle that was never opened. Check you 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 in %s

(P) Perl detected an attempt to copy an internal value that is not copyable.

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 *perlsub*.

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 Infinity or a very large number (> 1e308). See pack in *perlfunc*.

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 pack in *perlfunc*.

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 pack in *perlfunc*.

Can't bless non-reference value

(F) Only hard references may be blessed. This is how Perl "enforces" encapsulation of objects. See *perlobj*.

Can't call method "%s" in empty package "%s"

(F) You called a method correctly, and it correctly indicated a package functioning as a class, but that package doesn't have ANYTHING defined in it, let alone methods. See *perlobj*.

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 *perlobj*.

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 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 array into hash

(F) You used an array where a hash was expected, but the array has no information on how to map from keys to array indices. You can do that only with arrays that have a hash reference at index 0.

Can't coerce %s to integer 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 coerce %s to number in %s

(F) Certain types of SVs, in particular real symbol table entries (typeglobs), can't be forced to stop being what they are.

Can't coerce %s to string in %s

(F) Certain types of SVs, in particular real symbol table entries (typeglobs), can't be forced to stop being what they are.

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 class for non-scalar %s in "%s"

(F) Currently, only scalar variables can be declared with a specific class qualifier in a "my" or "our" declaration. The semantics may be extended for other types of variables in future.

Can't declare %s in "%s"

(F) Only scalar, array, and hash variables may be declared as "my" or "our" variables. They must have ordinary identifiers as names.

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, or a FIFO. 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 {n,m} with n > m in regex; marked by <– HERE in m/%s/

(F) Minima must be less than or equal to maxima. If you really want your regexp to match something 0 times, just put `{0}`. The <– HERE shows in the regular expression about where the problem was discovered. See *perlre*.

Can't do setegid!

(P) The setegid() call failed for some reason in the setuid emulator of suidperl.

Can't do seteuid!

(P) The setuid emulator of suidperl failed for some reason.

Can't do setuid

(F) This typically means that ordinary perl tried to exec suidperl to do setuid emulation, but couldn't exec it. It looks for a name of the form sperl5.000 in the same directory that the perl executable resides under the name perl5.000, typically /usr/local/bin on Unix machines. If the file is there, check the execute permissions. If it isn't, ask your sysadmin why he and/or she removed it.

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 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 (remember that the names of character properties consist only of alphanumeric characters), or maybe you forgot the `Is` or `In` prefix?

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 goto in *perlfunc*.

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 %s property definition %s

(F) You may have tried to use `\p` which means a Unicode property (for example `\p{Lu}` is all uppercase letters). If you did mean to use a Unicode property, see *perlunicode* for the list of known 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, until possible `\E`).

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. A good programmer’s editor will have a way to help you find these characters.

Can’t fork

(F) A fatal error occurred while trying to fork while opening a pipeline.

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 goto in *perlfunc*.

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 goto in *perlfunc*.

Can't goto subroutine from an eval-string

(F) The "goto subroutine" call can't be used to jump out of an eval "string". (You can use it to jump out of an eval {BLOCK}, but you probably don't want to.)

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 goto in *perlfunc*.

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 "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 last in *perlfunc*.

Can't localize lexical variable %s

(F) You used local on a variable name that was previously declared as a lexical variable using "my". 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 pseudo-hash element

(F) You said something like `local $ar->{'key'}`, where $ar is a reference to a pseudo-hash. That hasn't been implemented yet, but you can get a similar effect by localizing the corresponding array element directly – `local $ar->[$ar->[0]{'key'}]`.

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 require in *perlfunc* 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 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 *perlobj*.

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 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 Lvalue subroutines in *perlsub*.

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 next in *perlfunc*.

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.

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

(F) The script you specified can't be opened for the indicated reason.

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 *perlvms*) so that environ is not searched.

Can't redefine active sort subroutine %s

(F) Perl optimizes the internal handling of sort subroutines and keeps pointers into them. You tried to redefine one such sort subroutine when it was currently active, which is not allowed. If you really want to do this, you should write `sort { &func } @x` instead of `sort func @x`.

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 redo in *perlfunc*.

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 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 method name is `???`, this is an internal error.

Can’t reswap uid and euid

(P) The setreuid() call failed for some reason in the setuid emulator of suidperl.

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 *perlsub*.

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 swap uid and euid

(P) The setreuid() call failed for some reason in the setuid emulator of suidperl.

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 unshift

(F) You tried to unshift an "unreal" array that can’t be unshifted, such as the main Perl stack.

Can’t upgrade that kind of scalar

(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 upgrade to undef

(P) The undefined SV is the bottom of the totem pole, in the scheme of upgradability. Upgrading to undef indicates an error in the code calling sv_upgrade.

Can’t use anonymous symbol table for method lookup

(P) 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 *perlref*.

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 %s for loop variable

(F) Only a simple scalar variable may be used as a loop variable on a foreach.

Can't use global %s in "my"

(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 "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

(F) Only hard references are allowed by "strict refs". Symbolic references are disallowed. See *perlref*.

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 an 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 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 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.

close() on unopened filehandle %s

(W unopened) You tried to close a filehandle that was never opened.

Code missing after '/'

(F) You had a (sub-)template that ends with a '/'. There must be another template code following the slash. See pack in *perlfunc*.

%s: Command not found

(A) You've accidentally run your script through **csh** instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

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*.)

cond_broadcast() called on unlocked variable

(W threads) Within a thread-enabled program, you tried to call cond_broadcast() on a variable which wasn't locked. The cond_broadcast() function is used to wake up another thread that is waiting in a cond_wait(). To ensure that the signal isn't sent before the other thread has a chance to enter the wait, it is usual for the signaling thread to first wait for a lock on variable. This lock attempt will only succeed after the other thread has entered cond_wait() and thus relinquished the lock.

cond_signal() called on unlocked variable

(W threads) Within a thread-enabled program, you tried to call cond_signal() on a variable which wasn't locked. The cond_signal() function is used to wake up another thread that is waiting in a cond_wait(). To ensure that the signal isn't sent before the other thread has a chance to enter the wait, it is usual for the signaling thread to first wait for a lock on variable. This lock attempt will only succeed after the other thread has entered cond_wait() and thus relinquished the lock.

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 connect in *perlfunc*.

Constant(%s)%s: %s

(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` or `charnames` pragma? See *charnames* and *overload*.

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 Constant Functions in *perlsub* and *constant*.

Constant subroutine %s redefined

(S) You redefined a subroutine which had previously been eligible for inlining. See Constant Functions in *perlsub* for commentary and workarounds.

Constant subroutine %s undefined

(W misc) You undefined a subroutine which had previously been eligible for inlining. See Constant Functions in *perlsub* for commentary and workarounds.

Copy method did not return a reference

(F) The method which overloads "=" is buggy. See Copy Constructor in *overload*.

CORE::%s is not a keyword

(F) The CORE:: namespace is reserved for Perl keywords.

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%lx at 0x%lx

(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 pack in *perlfunc*.

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.

defined(@array) is deprecated

(D deprecated) defined() is not usually 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.

defined(%hash) is deprecated

(D deprecated) defined() is not usually useful on hashes because it checks for an undefined *scalar* value. If you want to see if the hash is empty, just use `if (%hash) { # not empty }` for example.

%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`.

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.

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 require in *perlfunc*.

(Did you mean &%s instead?)

(W) 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 both `$@` and `$_` were 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 pack in *perlfunc*.

Don't know how to handle magic of type '%s'

(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 dump in *perlfunc*.

Duplicate free() ignored

(S malloc) An internal routine called free() on something that had already been freed.

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 %s

(F) `\p` and `\P` are used to introduce a named Unicode property, as described in *perlunicode* and *perlre*. 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.

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.

%s: 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 (?{ code }) in *perlre*, and *perlsec*.

%s: Eval-group not allowed at run time

(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 explicitly building the pattern from an interpolated string at run time and using that in an eval(). See (?{ code }) in *perlre*.

%s: Eval-group not allowed, use re 'eval'

(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 (?{ code }) in *perlre*.

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 in MacPerl. See *perlport*.

Execution of %s aborted due to compilation errors

(F) The final summary message when a Perl compilation fails.

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 sort in *perlfunc*.

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.

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 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) 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 "-". Consider quoting the "-", "\-". The <– HERE shows in the regular expression about where the problem was discovered. See *perlre*.

Fatal VMS error 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?

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 open in *perlfunc*.

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 "<" or nothing. If you intended only to read from the file, use "<". See open in *perlfunc*. 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 occured 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 occured 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 getsockopt in *perlfunc*.

Global symbol "%s" requires explicit package name

(F) You've said "use strict vars", which indicates that all variables must either be lexically scoped (using "my"), declared beforehand using "our", or explicitly qualified to say which package the global variable is in (using "::").

glob failed (%s)

(W 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".

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 `goto` in *perlfunc*.

()-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 `pack` in *perlfunc*.

%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.

Hash %%s missing the % in argument %d of %s()

(D deprecated) Really old Perl let you omit the % on hash names in some spots. This is now heavily deprecated.

%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.

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 *perlport* 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.

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 %s (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 syntax) An illegal character was found in a prototype declaration. Legal characters in prototypes are $, @, %, *, ;, [,], &, and \.

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 *perlsub*.

Illegal declaration of subroutine %s

(F) A subroutine was not declared correctly. See *perlsub*.

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 switch in PERL5OPT: %s

(X) The PERL5OPT environment variable may only be used to set the following switches: **-[DIMUdmtw]**.

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 G_KEEPERR in *perlcall*.

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).

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 *perlsec* 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. See *perlsec*.

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 *perlsec*.

Integer overflow in %s number

(W 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.

Internal disaster in regex; marked by <– HERE in m/%s/

(P) Something went badly wrong in the regular expression parser. The <– HERE shows in the regular expression about where 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 exec LIST in *perlvms*). 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 urp in regex; marked by <– HERE in m/%s/

(P) Something went badly awry in the regular expression parser. The <– HERE shows in the regular expression about where 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 Terms and List Operators (Leftward) in *perlop*.

Invalid %s attribute: %s

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

The indicated attributes for a subroutine or variable were not recognized by Perl or by a user-supplied handler. See *attributes*.

Invalid conversion in %s: "%s"

(W printf) Perl does not understand the given format conversion. See sprintf in *perlfunc*.

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 in the regular expression about where the problem was discovered. See *perlre*.

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 *perlop*.

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 type '%s' in %s

(F) The given character is not a valid pack or unpack type. See pack in *perlfunc*. (W) The given character is not a valid pack or unpack type but used to be silently ignored.

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 you 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 or an ioctl call (SIOCATMARK).

'%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.

junk on end of regexp

(P) The regular expression parser is confused.

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 last in *perlfunc*.

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 last in *perlfunc*.

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 last in *perlfunc*.

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 alread used up when an unpack length/code combination tried to obtain more data. This results in an undefined value for the length. See pack in *perlfunc*.

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 listen in *perlfunc*.

Lookbehind longer than %d not implemented in regex; marked by <– HERE in 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. The <– HERE shows in the regular expression about where the problem was discovered.

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 subs returning %s not implemented yet

(F) Due to limitations in the current implementation, array and hash values cannot be returned in subroutines used in lvalue context. See Lvalue subroutines in *perlsub*.

Malformed integer in [in pack]

(F) Between the brackets enclosing a numeric repeat count only digits are permitted. See pack in *perlfunc*.

Malformed integer in [in unpack]

(F) Between the brackets enclosing a numeric repeat count only digits are permitted. See pack in *perlfunc*.

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.

Malformed UTF-8 character (%s)

Perl detected something that didn't comply with UTF-8 encoding rules.

One possible cause is that you read in data that you thought to be in UTF-8 but it wasn't (it was for example legacy 8-bit data). Another possibility is careless use of utf8::upgrade().

Malformed UTF-16 surrogate

Perl thought it was reading UTF-16 encoded character data but while doing it Perl met a malformed Unicode surrogate.

%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 in the regular expression about where the problem was discovered. See *perlre*.

"%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 unpack in *perlfunc*.

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 %sbrace%s on \N{}

(F) Wrong syntax of character name literal `\N{charname}` within 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 name in "my sub"

(F) The reserved 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 right brace on %s

(F) Missing right brace in `\p{...}` or `\P{...}`.

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 *perlrun* for full details about `-M` and `-m`.

More than one argument to 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 open in *perlfunc* for details.

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 be followed by 'a*', 'A*' or 'Z*'

(F) You had a pack template indicating a counted-length string, Currently the only things that can have their length counted are a*, A* or Z*. See pack in *perlfunc*.

'/' 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 pack in *perlfunc*.

"my sub" not yet implemented

(F) Lexically scoped subroutines are not yet implemented. Don't try that yet.

"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 provided for this purpose.

NOTE: This warning detects symbols that have been used only once so $c, @c, %c, *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.

Negative '/' count in unpack

(F) The length count obtained from a length/code unpack operation was negative. See pack in *perlfunc*.

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.

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 in the regular expression about where the problem was discovered.

Note that the minimal matching quantifiers, `*?`, `+?`, and `??` appear to be nested quantifiers, but aren't. See *perlre*.

%s never introduced

(S internal) The symbol in question was declared but somehow went out of scope before it could possibly have been used.

Newline in left-justified string for %s

(W printf) There is a newline in a string to be left justified by `printf` or `sprintf`.

The padding spaces will appear after the newline, which is probably not what you wanted. Usually you should remove the newline from the string and put formatting characters in the `sprintf` format.

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 *perlsec*.

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 use in *perlfunc* and import in *perlfunc*. 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 perl5db.pl file (or some facsimile thereof) didn't define a routine to be called at the beginning of each statement. Which is odd, because the file should have been required automatically, and should have blown up the require if it didn't parse right.

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 DBsub routine

(F) The currently executing code was compiled with the **-d** switch, but for some reason the perl5db.pl file (or some facsimile thereof) didn't define a DB::sub routine to be called at the beginning of each ordinary subroutine call.

No -e allowed in setuid scripts

(F) A setuid script can’t be specified by the user.

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 pack in *perlfunc*.

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 #! line

(F) The setuid emulator requires that scripts have a well-formed #! line even on machines that don’t support the #! construct.

"no" not allowed in expression

(F) The "no" keyword is recognized and executed at compile time, and returns no useful value. See *perlmod*.

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 semantics. 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 space allowed after -%c

(F) The argument to the indicated command line switch must follow immediately after the switch, without intervening spaces.

No %s specified for -%c

(F) The indicated command line switch needs a mandatory argument, but you haven’t specified one.

No such class %s

(F) You provided a class qualifier in a "my" or "our" declaration, but this class doesn’t exist at this point in your program.

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 pseudo-hash field "%s"

(F) You tried to access an array as a hash, but the field name used is not defined. The hash at index 0 should map all valid field names to array indices for that to work.

No such pseudo-hash field "%s" in variable %s of type %s

(F) You tried to access a field of a typed variable where the type does not know about the field name. The field names are looked up in the %FIELDS hash in the type package at compile time. The %FIELDS hash is %usually set up with the 'fields' pragma.

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 *perlref*.

Not a format reference

(F) I'm not sure how you managed to generate a reference to an anonymous format, but this indicates you did, and that it didn't exist.

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 *perlref*.

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 *perlref*.

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 *perlref*.

Not a perl script

(F) The setuid emulator requires that scripts have a well-formed #! line even on machines that don't support the #! construct. The line must mention perl.

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 *perlref*.

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 *perlref*.

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 *perlform*.

%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.

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 filename used

(F) You can't require the null filename, especially because on many machines that means the current directory! See require in *perlfunc*.

NULL OP IN RUN

(P 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 *perlform*.

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").

Octal number in vector unsupported

(F) Numbers with a leading `0` are not currently allowed in vectors. The octal number interpretation of such numbers may be supported in a future version.

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 *perlport* for more on portability concerns.

See also *perlport* for writing portable code.

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) You tried to do a read/write/send/recv operation with an offset pointing outside the buffer. This is difficult to imagine. The sole exception to this is that `sysread()`ing past the buffer will extend the buffer and zero pad the new area.

%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 -X in *perlfunc*.

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.

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 `fallback` overloading key is specified to be true. See *overload*.

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'".

"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 "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 %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 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 unpack

(F) You had a template that specified an absolute position outside the string being unpacked. See pack in *perlfunc*.

%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 pack in *perlfunc*.

page overflow

(W io) A single call to write() produced more lines than can fit on a page. See *perlform*.

panic: %s

(P) An internal error.

panic: ck_grep

(P) Failed an internal consistency check trying to compile a grep.

panic: ck_split

(P) Failed an internal consistency check trying to compile a split.

panic: corrupt saved stack index

(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: Devel::DProf inconsistent subroutine return

(P) Devel::DProf called a subroutine that exited using goto(LABEL), last(LABEL) or next(LABEL). Leaving that way a subroutine called from an XSUB will lead very probably to a crash of the interpreter. This is a bug that will hopefully one day get fixed.

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: frexp

(P) The library function frexp() failed, making printf("%f") impossible.

panic: goto

(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: INTERPCASEMOD

(P) The lexer got into a bad state at a case modifier.

panic: INTERPCONCAT

(P) The lexer got into a bad state parsing a string with brackets.

panic: kid popen errno read

(F) forked child returned an incomprehensible message about its errno.

panic: last

(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

(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

(P) Something requested a negative number of bytes of malloc.

panic: mapstart

(P) The compiler is screwed up with respect to the map() function.

panic: memory wrap

(P) Something tried to allocate more memory than possible.

panic: null array

(P) One of the internal array routines was passed a null AV pointer.

panic: pad_alloc

(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_free curpad

(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_free po

(P) An invalid scratch pad offset was detected internally.

panic: pad_reset curpad

(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_sv po

(P) An invalid scratch pad offset was detected internally.

panic: pad_swipe curpad

(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

(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

(P) Something terrible went wrong in setting up for the split.

panic: realloc

(P) Something requested a negative number of bytes of realloc.

panic: restartop

(P) Some internal routine requested a goto (or something like it), and didn't supply the destination.

panic: return

(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

(P) scan_num() got called on something that wasn't a number.

panic: sv_insert

(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: utf16_to_utf8: odd bytelen

(P) Something tried to call utf16_to_utf8 with an odd (as opposed to even) byte length.

panic: yylex

(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", and "local" bind tighter than comma.

-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().)

(perhaps you forgot to load "%s"?)

(F) This is an educated guess made in conjunction with the message "Can't locate object method \"%s\" via package \"%s\"". It often means that a method requires a package that has not been loaded.

Perl %s required–this is only version %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 require in *perlfunc*.

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"

See PERL_SIGNALS in *perlrun* for legal values.

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, 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 *perllocale* section **LOCALE PROBLEMS**.

Permission denied

(F) The setuid emulator in suidperl decided you were up to no good.

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 "*".

-P not allowed for setuid/setgid script

(F) The script would have to be opened by the C preprocessor by name, which provides a race condition that breaks security.

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 in the regular expression about where 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 *perlre*.

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 [%s 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 in the regular expression about where the problem was discovered. See *perlre*.

POSIX syntax [. . is reserved for future] **extensions in regex; marked by <– HERE in m/%s/**

(F regexp) 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 in the regular expression about where the problem was discovered. See *perlre*.

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 in the regular expression about where the problem was discovered. See *perlre*.

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 ioctl in *perlfunc*.

Possible precedence problem on bitwise %c 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 %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.

Possible Y2K bug: %s

(W y2k) You are concatenating the number 19 with another number, which could be a potential Year 2000 problem.

pragma "attrs" is deprecated, use "sub NAME : ATTRS" instead

(D deprecated) You have written something like this:

```
sub doit
{
    use attrs qw(locked);
}
```

You should use the new declaration syntax instead.

```
sub doit : locked
{
    ...
```

The `use attrs` pragma is now obsolete, and is only provided for backward-compatibility. See Subroutine Attributes in *perlsub*.

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 Signals in *perlipc*. See also "Process terminated by SIGTERM/SIGINT" in *perlos2*.

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.

Pseudo-hashes are deprecated

(D deprecated) Pseudo-hashes were deprecated in Perl 5.8.0 and they will be removed in Perl 5.10.0, see *perl58delta* for more details. You can continue to use the `fields` pragma.

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 in the regular expression about where the problem was discovered. See *perlre*.

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 in the regular expression about where the problem was discovered. See *perlre*.

Quantifier unexpected on zero-length expression; marked by <– HERE in 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}/`.

The <– HERE shows in the regular expression about where the problem was discovered.

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.

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: %lx

(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 -DDEBUGGING to use -D switch

(F 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 inheritance detected in package '%s'

(F) More than 100 levels of inheritance were used. Probably indicates an unintended loop in your inheritance hierarchy.

Recursive inheritance detected while looking for method %s

(F) More than 100 levels of inheritance were encountered while invoking a method. Probably indicates an unintended loop in your inheritance hierarchy.

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 miscount in sv_replace()

(W internal) The internal sv_replace() function was handed a new SV with a reference count of other than 1.

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 value 7 inserted into the regular expression, prepend a zero to make the number at least two digits: `\07`

The <– HERE shows in the regular expression about where the problem was discovered.

regexp memory corruption

(P) The regular expression engine got confused by what the regular expression compiler gave it.

Regexp out of space

(P) A "can't happen" error, because safemalloc() should have caught it earlier.

Repeated format line will never terminate (~~ and @# incompatible)

(F) Your format containes 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 *perlform*.

Reversed %s= operator

(W syntax) You wrote your assignment operator backwards. The = must always comes last, to avoid ambiguity with subsequent unary operators.

Runaway format

(F) Your format contained the ~~ repeat-until-blank sequence, but it produced 200 lines at once, and the 200th line looked exactly like the 199th line. Apparently you didn't arrange for the arguments to exhaust themselves, either by using ^ instead of @ (for scalar variables), or by shifting or popping (for array variables). See *perlform*.

Scalars leaked: %d

(P) 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 *perlref*.

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 *perlref*.

Script is not setuid/setgid in suidperl

(F) Oddly, the suidperl program was invoked on a script without a setuid or setgid bit set. This doesn't make much sense.

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 5.9.0 a // can also be the *defined-or* construct, not just the empty search pattern. Therefore code written in Perl 5.9.0 or later that uses the // as the *defined-or* can be misparsed by pre-5.9.0 Perls as a non-terminated search pattern.

%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 (? incomplete in regex; marked by <– HERE in m/%s/

(F) A regular expression ended with an incomplete extension (?. The <– HERE shows in the regular expression about where the problem was discovered. See *perlre*.

Sequence (?%s...) 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 in the regular expression about where the problem was discovered. See *perlre*.

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 in the regular expression about where the problem was discovered. See *perlre*.

Sequence (?#... not terminated in regex; marked by <– HERE in m/%s/

(F) A regular expression comment must be terminated by a closing parenthesis. Embedded parentheses aren't allowed. The <– HERE shows in the regular expression about where the problem was discovered. See *perlre*.

Sequence (?{...}) not terminated or not {}-balanced in regex; marked by <– HERE in m/%s/

(F) If the contents of a (?{...}) clause contains braces, they must balance for Perl to properly detect the end of the clause. The <– HERE shows in the regular expression about where the problem was discovered. See *perlre*.

500 Server error

See Server error.

Server error

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 setsockopt in *perlfunc*.

Setuid/gid script is writable by world

(F) The setuid emulator won't run a script that is writable by the world, because the world might have written on it already.

shm%s not implemented

(F) You don't have System V shared memory IPC on your system.

<> should be quotes

(F) You wrote `require <file>` when you should have written `require 'file'`.

/%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?

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 a numeric value

(F) A sort comparison routine must return a number. You probably blew it by not using <=> or `cmp`, or by not using them correctly. See sort in *perlfunc*.

Sort subroutine didn't return single value

(F) A sort comparison subroutine may not return a list value with more or less than one element. See sort in *perlfunc*.

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 splice in *perlfunc*.

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 split in *perlfunc*.

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.

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.

Stub found while resolving method '%s' overloading %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 redefined

(W redefine) You redefined a subroutine. To suppress this warning, say

```
        {
            no warnings 'redefine';
            eval "sub name { ... }";
        }
```

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 Quote and Quote-like Operators in *perlop*.

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 substr in *perlfunc*. 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).

suidperl is no longer needed since %s

(F) Your Perl was compiled with **-D**SETUID_SCRIPTS_ARE_SECURE_NOW, but a version of the setuid emulator somehow got run anyway.

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 in the regular expression about where the problem was discovered. See *perlre*.

Switch condition not recognized in regex; marked by <– HERE in m/%s/

(F) If the argument to the (?(...)if-clause|else-clause) construct is a number, it can be only a number. The <– HERE shows in the regular expression about where the problem was discovered. See *perlre*.

switching effective %s is not implemented

(F) While under the `use filetest` pragma, we cannot switch the real and effective uids or gids.

%s syntax

(F) The final summary message when a `perl -c` succeeds.

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".

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.

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 $[in *perlvar*.

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

The function indicated isn't implemented on this architecture, according to the probings of Configure.

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) Currently this attribute is not supported on `my` or `sub` declarations. See our in *perlfunc*.

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 *perlvms*) so that the environ array isn't the target of the change to %ENV which produced the warning.

thread failed to start: `%s`

(F) The entry point function of threads->create() failed for some reason.

5.005 threads are deprecated

(D deprecated) The 5.005-style threads (activated by `use Thread;`) are deprecated and one should use the new ithreads instead, see *perl58delta* for more details.

Tied variable freed while still in use

(F) An access method for a tied variable (e.g. FETCH) did something to free the variable. Since continuing the current operation is likely to result in a coredump, Perl is bailing out instead.

times not implemented

(F) Your version of the C library apparently doesn't do times(). I suspect you're not running on Unix.

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 User-Defined Character Properties in *perlunicode*.

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 late for "-`%s`" option

(X) The #! line (or local equivalent) in a Perl script contains the **-M** or **-m** option. This is an error because **-M** and **-m** options are not intended for use inside scripts. Use the `use` pragma instead.

Too late for "-T" option

(X) The #! line (or local equivalent) in a Perl script contains 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 **-T** option is a part of Perl's first argument: e.g. change `perl -n -T` to `perl -T -n`.

If the Perl script is being executed as `perl scriptname`, then the **-T** option must appear on the command line: `perl -T scriptname`.

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)'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 *perlre*.

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 %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 *perlref*.

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).

Unable to create sub named "%s"

(F) You attempted to create or access a subroutine with an illegal name.

Unbalanced context: %d more PUSHes than POPs

(W internal) The exit code detected an internal inconsistency in how many execution contexts were entered and left.

Unbalanced saves: %d more saves than restores

(W internal) The exit code detected an internal inconsistency in how many values were temporarily localized.

Unbalanced scopes: %d more ENTERs than LEAVEs

(W internal) The exit code detected an internal inconsistency in how many blocks were entered and left.

Unbalanced tmps: %d more allocs than frees

(W 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 *perlform*.

Undefined sort subroutine "%s" called

(F) The sort comparison routine specified doesn't seem to exist. Perhaps it's in a different package? See sort in *perlfunc*.

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 sort in *perlfunc*.

Undefined top format "%s" called

(F) The format indicated doesn't seem to exist. Perhaps it's really in another package? See *perlform*.

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.

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.

Unicode character %s is illegal

(W utf8) Certain Unicode characters have been designated off-limits by the Unicode standard and should not be generated. If you really know what you are doing you can turn off this warning by `no warnings 'utf8';`.

Unknown BYTEORDER

(F) There are no byte-swapping functions for a machine with this byte order.

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 "re" subpragma '%s' (known ones are: %s)

You tried to use an unknown subpragma of the "re" pragma.

Unknown switch condition (?(%.2s 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 may be lookahead or lookbehind (the condition is true if the lookahead or lookbehind is true), a (?{...}) construct (the condition is true if the code evaluates to a true value), or a number (the condition is true if the set of capturing parentheses named by the number matched).

The <– HERE shows in the regular expression about where the problem was discovered. See *perlre*.

Unknown Unicode option letter '%c'

You specified an unknown Unicode option. See *perlrun* documentation of the `-C` switch for the list of known options.

Unknown Unicode option value %x

You specified an unknown Unicode option. See *perlrun* documentation of the `-C` switch for the list of known options.

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 imported this module first.

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 in the regular expression about where the problem was discovered. See *perlre*.

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 in the regular expression about where the problem was discovered. See *perlre*.

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

(F) The Perl parser has no idea what to do with the specified character in your Perl script (or eval). Perhaps you tried to run a compressed script, a binary program, or a directory as a Perl program.

/%s/: Unrecognized escape \\%c in character class passed through

(W regexp) You used a backslash-character combination which is not recognized by Perl inside character classes. The character was understood literally.

Unrecognized escape \\%c passed through

(W misc) You used a backslash-character combination which is not recognized by Perl.

Unrecognized escape \\%c passed through in regex; marked by <– HERE in m/%s/

(W regexp) You used a backslash-character combination which is not recognized by Perl. This combination appears in an interpolated variable or a `'`-delimited regular expression. The character was understood literally. The <– HERE shows in the regular expression about where 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.)

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 chomp in *perlfunc*.

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 pack in *perlfunc*.

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".

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 FUNCTIONS in *POSIX* for more information.

Usage: Win32::%s(%s)

(F) You called a Win32 function with incorrect arguments. See *Win32* for more information.

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 in the regular expression about where the problem was discovered. See *perlre*.

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 in the regular expression about where the problem was discovered. See *perlre*.

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 *perlref* 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 "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 *perlmod*.

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 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 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 implicit split to @_ is deprecated

(D deprecated) It makes a lot of work for the compiler when you clobber a subroutine's argument list, so it's better if you assign the results of a split() explicitly to an array (or list).

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 $ * is deprecated

(D deprecated) This variable magically turned on multi-line pattern matching, both for you and for any luckless subroutine that you happen to call. You should use the new `//m` and `//s` modifiers now to do that without the dangerous action-at-a-distance effects of `$*`.

Use of $ # is deprecated

(D deprecated) This was an ill-advised attempt to emulate a poorly defined **awk** feature. Use an explicit printf() or sprintf() instead.

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 -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 "package" with no arguments is deprecated

(D deprecated) You used the `package` keyword without specifying a package name. So no namespace is current at all. Using this can cause many otherwise reasonable constructs to fail in baffling ways. `use strict;` instead.

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, either, because you can overload the numification and stringification operators and then you assumedly know what you are doing.

Use of reserved word "%s" is deprecated

(D deprecated) The indicated bareword is a reserved word. Future versions of perl may use it as a keyword, so you're better off either explicitly quoting the word in a manner appropriate for its context of use, or using a different name altogether. The warning can be suppressed for subroutine names by either adding a `&` prefix, or using a package qualifier, e.g. `&our()`, or `Foo::our()`.

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 *perlsec*.

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 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.

Using a hash as a reference is deprecated

(D deprecated) You tried to use a hash as a reference, as in `%foo->{"bar"}` or `%$ref->{"hello"}`. Versions of perl <= 5.6.1 used to allow this syntax, but shouldn't have. It is now deprecated, and will be removed in a future version.

Using an array as a reference is deprecated

(D deprecated) You tried to use an array as a reference, as in `@foo->[23]` or `@$ref->[99]`. Versions of perl <= 5.6.1 used to allow this syntax, but shouldn't have. It is now deprecated, and will be removed in a future version.

UTF-16 surrogate %s

(W utf8) You tried to generate half of an UTF-16 surrogate by requesting a Unicode character between the code points 0xD800 and 0xDFFF (inclusive). That range is reserved exclusively for the use of UTF-16 encoding (by having two 16-bit UCS-2 characters); but Perl encodes its characters in UTF-8, so what you got is a very illegal character. If you really know what you are doing you can turn off this warning by `no warnings 'utf8';`.

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.

Variable "%s" is not imported%s

(F) While "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; marked by <– HERE in m/%s/

(F) Lookbehind is allowed only for subexpressions whose length is fixed and known at compile time. The <– HERE shows in the regular expression about where the problem was discovered. See *perlre*.

"%s" variable %s masks earlier declaration in same %s

(W misc) A "my" or "our" 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 referents to it are destroyed.

Variable "%s" may be unavailable

(W closure) An inner (nested) *anonymous* subroutine is inside a *named* subroutine, and outside that is another subroutine; and the anonymous (innermost) subroutine is referencing a lexical variable defined in the outermost subroutine. For example:

```
sub outermost { my $a; sub middle { sub { $a } } }
```

If the anonymous subroutine is called or referenced (directly or indirectly) from the outermost subroutine, it will share the variable as you would expect. But if the anonymous subroutine is called or referenced when the outermost subroutine is not active, it will see the value of the shared variable as it was before and during the *first* call to the outermost subroutine, which is probably not what you want.

In these circumstances, it is usually best to make the middle subroutine anonymous, using the `sub {}` syntax. Perl has specific support for shared variables in nested anonymous subroutines; a named subroutine in between interferes with this feature.

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 subroutine.

When the inner subroutine is called, it will probably 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.

Furthermore, if the outer subroutine is anonymous and references a lexical variable outside itself, then the outer and inner subroutines will *never* share the given variable.

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 called or referenced, they are automatically rebound to the current values of such variables.

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.

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: 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.

Wide character in %s

(W 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 binmode in *perlfunc*.

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 an 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%s" does not map to Unicode

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 pack in *perlfunc*.

'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 pack in *perlfunc*.

Xsub "%s" called in sort

(F) The use of an external subroutine as a sort comparison is not yet supported.

Xsub called in sort

(F) The use of an external subroutine as a sort comparison is not yet supported.

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 initialise the random seed for hashes, Perl could not get any randomness out of your system. This usually indicates Something Very Wrong.

Chapter 37

perllexwarn

Perl Lexical Warnings

37.1 DESCRIPTION

The `use warnings` pragma is a replacement for both the command line flag **-w** and the equivalent Perl variable, `$^W`.

The pragma works just like the existing "strict" pragma. This means that the scope of the warning pragma is limited to the enclosing block. It also means that the pragma setting will not leak across files (via `use`, `require` or `do`). This allows authors to independently define the degree of warning checks that will be applied to their module.

By default, optional warnings are disabled, so any legacy code that doesn't attempt to control the warnings will work unchanged.

All warnings are enabled in a block by either of these:

```
    use warnings ;
    use warnings 'all' ;
```

Similarly all warnings are disabled in a block by either of these:

```
    no warnings ;
    no warnings 'all' ;
```

For example, consider the code below:

```
    use warnings ;
    my @a ;
    {
        no warnings ;
        my $b = @a[0] ;
    }
    my $c = @a[0];
```

The code in the enclosing block has warnings enabled, but the inner block has them disabled. In this case that means the assignment to the scalar `$c` will trip the `"Scalar value @a[0] better written as $a[0]"` warning, but the assignment to the scalar `$b` will not.

37.1.1 Default Warnings and Optional Warnings

Before the introduction of lexical warnings, Perl had two classes of warnings: mandatory and optional.

As its name suggests, if your code tripped a mandatory warning, you would get a warning whether you wanted it or not. For example, the code below would always produce an `"isn't numeric"` warning about the "2:".

```
my $a = "2:" + 3;
```

With the introduction of lexical warnings, mandatory warnings now become *default* warnings. The difference is that although the previously mandatory warnings are still enabled by default, they can then be subsequently enabled or disabled with the lexical warning pragma. For example, in the code below, an `"isn't numeric"` warning will only be reported for the `$a` variable.

```
my $a = "2:" + 3;
no warnings ;
my $b = "2:" + 3;
```

Note that neither the **-w** flag or the `$^W` can be used to disable/enable default warnings. They are still mandatory in this case.

37.1.2 What's wrong with -w and `$^W`

Although very useful, the big problem with using **-w** on the command line to enable warnings is that it is all or nothing. Take the typical scenario when you are writing a Perl program. Parts of the code you will write yourself, but it's very likely that you will make use of pre-written Perl modules. If you use the **-w** flag in this case, you end up enabling warnings in pieces of code that you haven't written.

Similarly, using `$^W` to either disable or enable blocks of code is fundamentally flawed. For a start, say you want to disable warnings in a block of code. You might expect this to be enough to do the trick:

```
 {
     local ($^W) = 0 ;
     my $a =+ 2 ;
     my $b ; chop $b ;
 }
```

When this code is run with the **-w** flag, a warning will be produced for the `$a` line – `"Reversed += operator"`.

The problem is that Perl has both compile-time and run-time warnings. To disable compile-time warnings you need to rewrite the code like this:

```
 {
     BEGIN { $^W = 0 }
     my $a =+ 2 ;
     my $b ; chop $b ;
 }
```

The other big problem with `$^W` is the way you can inadvertently change the warning setting in unexpected places in your code. For example, when the code below is run (without the **-w** flag), the second call to `doit` will trip a `"Use of uninitialized value"` warning, whereas the first will not.

```
sub doit
{
    my $b ; chop $b ;
}
```

```
    doit() ;

    {
        local ($^W) = 1 ;
        doit()
    }
```

This is a side-effect of `$^W` being dynamically scoped.

Lexical warnings get around these limitations by allowing finer control over where warnings can or can't be tripped.

37.1.3 Controlling Warnings from the Command Line

There are three Command Line flags that can be used to control when warnings are (or aren't) produced:

-w

This is the existing flag. If the lexical warnings pragma is **not** used in any of you code, or any of the modules that you use, this flag will enable warnings everywhere. See Backward Compatibility for details of how this flag interacts with lexical warnings.

-W

If the **-W** flag is used on the command line, it will enable all warnings throughout the program regardless of whether warnings were disabled locally using `no warnings` or `$^W =0`. This includes all files that get included via `use`, `require` or `do`. Think of it as the Perl equivalent of the "lint" command.

-X

Does the exact opposite to the **-W** flag, i.e. it disables all warnings.

37.1.4 Backward Compatibility

If you are used with working with a version of Perl prior to the introduction of lexically scoped warnings, or have code that uses both lexical warnings and `$^W`, this section will describe how they interact.

How Lexical Warnings interact with **-w**/`$^W`:

1. If none of the three command line flags (**-w**, **-W** or **-X**) that control warnings is used and neither `$^W` or the `warnings` pragma are used, then default warnings will be enabled and optional warnings disabled. This means that legacy code that doesn't attempt to control the warnings will work unchanged.

2. The **-w** flag just sets the global `$^W` variable as in 5.005 – this means that any legacy code that currently relies on manipulating `$^W` to control warning behavior will still work as is.

3. Apart from now being a boolean, the `$^W` variable operates in exactly the same horrible uncontrolled global way, except that it cannot disable/enable default warnings.

4. If a piece of code is under the control of the `warnings` pragma, both the `$^W` variable and the **-w** flag will be ignored for the scope of the lexical warning.

5. The only way to override a lexical warnings setting is with the **-W** or **-X** command line flags.

The combined effect of 3 & 4 is that it will allow code which uses the `warnings` pragma to control the warning behavior of $^W-type code (using a `local $^W=0`) if it really wants to, but not vice-versa.

37.1.5 Category Hierarchy

A hierarchy of "categories" have been defined to allow groups of warnings to be enabled/disabled in isolation.

The current hierarchy is:

```
all -+
     |
     +- closure
     |
     +- deprecated
     |
     +- exiting
     |
     +- glob
     |
     +- io -----------+
     |                |
     |                +- closed
     |                |
     |                +- exec
     |                |
     |                +- layer
     |                |
     |                +- newline
     |                |
     |                +- pipe
     |                |
     |                +- unopened
     |
     +- misc
     |
     +- numeric
     |
     +- once
     |
     +- overflow
     |
     +- pack
     |
     +- portable
     |
     +- recursion
     |
     +- redefine
     |
     +- regexp
     |
     +- severe -------+
     |                |
     |                +- debugging
     |                |
     |                +- inplace
     |                |
     |                +- internal
     |                |
     |                +- malloc
```

```
       |
       +- signal
       |
       +- substr
       |
       +- syntax -------+
       |                |
       |                +- ambiguous
       |                |
       |                +- bareword
       |                |
       |                +- digit
       |                |
       |                +- parenthesis
       |                |
       |                +- precedence
       |                |
       |                +- printf
       |                |
       |                +- prototype
       |                |
       |                +- qw
       |                |
       |                +- reserved
       |                |
       |                +- semicolon
       |
       +- taint
       |
       +- threads
       |
       +- uninitialized
       |
       +- unpack
       |
       +- untie
       |
       +- utf8
       |
       +- void
       |
       +- y2k
```

Just like the "strict" pragma any of these categories can be combined

```
    use warnings qw(void redefine) ;
    no warnings qw(io syntax untie) ;
```

Also like the "strict" pragma, if there is more than one instance of the `warnings` pragma in a given scope the cumulative effect is additive.

```
    use warnings qw(void) ; # only "void" warnings enabled
    ...
    use warnings qw(io) ;   # only "void" & "io" warnings enabled
    ...
    no warnings qw(void) ;  # only "io" warnings enabled
```

To determine which category a specific warning has been assigned to see *perldiag*.

Note: In Perl 5.6.1, the lexical warnings category "deprecated" was a sub-category of the "syntax" category. It is now a top-level category in its own right.

37.1.6 Fatal Warnings

The presence of the word "FATAL" in the category list will escalate any warnings detected from the categories specified in the lexical scope into fatal errors. In the code below, the use of `time`, `length` and `join` can all produce a `"Useless use of xxx in void context"` warning.

```
    use warnings ;

    time ;

    {
        use warnings FATAL => qw(void) ;
        length "abc" ;
    }

    join "", 1,2,3 ;

    print "done\n" ;
```

When run it produces this output

```
    Useless use of time in void context at fatal line 3.
    Useless use of length in void context at fatal line 7.
```

The scope where `length` is used has escalated the `void` warnings category into a fatal error, so the program terminates immediately it encounters the warning.

To explicitly turn off a "FATAL" warning you just disable the warning it is associated with. So, for example, to disable the "void" warning in the example above, either of these will do the trick:

```
    no warnings qw(void);
    no warnings FATAL => qw(void);
```

If you want to downgrade a warning that has been escalated into a fatal error back to a normal warning, you can use the "NONFATAL" keyword. For example, the code below will promote all warnings into fatal errors, except for those in the "syntax" category.

```
    use warnings FATAL => 'all', NONFATAL => 'syntax';
```

37.1.7 Reporting Warnings from a Module

The `warnings` pragma provides a number of functions that are useful for module authors. These are used when you want to report a module-specific warning to a calling module has enabled warnings via the `warnings` pragma.

Consider the module `MyMod::Abc` below.

```
    package MyMod::Abc;

    use warnings::register;
    sub open {
        my $path = shift ;
        if ($path !~ m#^/#) {
            warnings::warn("changing relative path to /var/abc")
                if warnings::enabled();
            $path = "/var/abc/$path";
        }
    }

    1 ;
```

The call to `warnings::register` will create a new warnings category called "MyMod::abc", i.e. the new category name matches the current package name. The `open` function in the module will display a warning message if it gets given a relative path as a parameter. This warnings will only be displayed if the code that uses `MyMod::Abc` has actually enabled them with the `warnings` pragma like below.

```
    use MyMod::Abc;
    use warnings 'MyMod::Abc';
    ...
    abc::open("../fred.txt");
```

It is also possible to test whether the pre-defined warnings categories are set in the calling module with the `warnings::enabled` function. Consider this snippet of code:

```
    package MyMod::Abc;

    sub open {
        warnings::warnif("deprecated",
                         "open is deprecated, use new instead") ;
        new(@_) ;
    }

    sub new
    ...
    1 ;
```

The function `open` has been deprecated, so code has been included to display a warning message whenever the calling module has (at least) the "deprecated" warnings category enabled. Something like this, say.

```
    use warnings 'deprecated';
    use MyMod::Abc;
    ...
    MyMod::Abc::open($filename) ;
```

Either the `warnings::warn` or `warnings::warnif` function should be used to actually display the warnings message. This is because they can make use of the feature that allows warnings to be escalated into fatal errors. So in this case

```
    use MyMod::Abc;
    use warnings FATAL => 'MyMod::Abc';
    ...
    MyMod::Abc::open('../fred.txt');
```

the `warnings::warnif` function will detect this and die after displaying the warning message.

The three warnings functions, `warnings::warn`, `warnings::warnif` and `warnings::enabled` can optionally take an object reference in place of a category name. In this case the functions will use the class name of the object as the warnings category.

Consider this example:

```
    package Original ;

    no warnings ;
    use warnings::register ;

    sub new
    {
        my $class = shift ;
        bless [], $class ;
    }

    sub check
    {
        my $self = shift ;
        my $value = shift ;

        if ($value % 2 && warnings::enabled($self))
          { warnings::warn($self, "Odd numbers are unsafe") }
    }

    sub doit
    {
        my $self = shift ;
        my $value = shift ;
        $self->check($value) ;
        # ...
    }

    1 ;

    package Derived ;

    use warnings::register ;
    use Original ;
    our @ISA = qw( Original ) ;
    sub new
    {
        my $class = shift ;
        bless [], $class ;
    }

    1 ;
```

The code below makes use of both modules, but it only enables warnings from `Derived`.

```
    use Original ;
    use Derived ;
    use warnings 'Derived';
    my $a = new Original ;
    $a->doit(1) ;
    my $b = new Derived ;
    $a->doit(1) ;
```

When this code is run only the `Derived` object, `$b`, will generate a warning.

```
    Odd numbers are unsafe at main.pl line 7
```

Notice also that the warning is reported at the line where the object is first used.

37.2 TODO

```
  perl5db.pl
    The debugger saves and restores C<$^W> at runtime. I haven't checked
    whether the debugger will still work with the lexical warnings
    patch applied.

  diagnostics.pm
    I *think* I've got diagnostics to work with the lexical warnings
    patch, but there were design decisions made in diagnostics to work
    around the limitations of C<$^W>. Now that those limitations are gone,
    the module should be revisited.

  document calling the warnings::* functions from XS
```

37.3 SEE ALSO

warnings, *perldiag*.

37.4 AUTHOR

Paul Marquess

Chapter 38

perldebug

Perl debugging

38.1 DESCRIPTION

First of all, have you tried using the **-w** switch?

If you're new to the Perl debugger, you may prefer to read *perldebtut*, which is a tutorial introduction to the debugger .

38.2 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.

38.2.1 Debugger Commands

The 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 §38.2.2.

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

Toggle trace mode (see also the `AutoTrace` option).

t expr

Trace through execution of `expr`. See Frame Listing Output Examples in *perldebguts* 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
```

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.

**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. We hope you know what one of these is, because they're supposed to be obvious.

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 §38.2.2 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've 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 ˆD

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

Displays 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/';
```

38.2.2 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 command or spawn 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 *BUGS* 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'd` 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.

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 `".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 Debugger Internals in *perldebguts* for details.

38.2.3 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 *perldebguts* for incredibly long examples of these.

38.2.4 Debugging compile-time statements

If you have compile-time executable statements (such as code within BEGIN 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 `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.

38.2.5 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 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 initilization. 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 *perldebguts*) are considered for internal use only, and as such are subject to change without notice.

38.2.6 Readline Support

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, you will have full editing capabilities much like 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. Unfortunately, the names of lexical variables are not available for completion.

38.2.7 Editor Support for Debugging

If you have the FSF'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.

Perl comes with a start file for making **emacs** act like a syntax-directed editor that understands (some of) Perl's syntax. Look in the *emacs* directory of the Perl source distribution.

A similar setup by Tom Christiansen for interacting with any vendor-shipped **vi** and the X11 window system is also available. This works similarly to the integrated multiwindow support that **emacs** provides, where the debugger drives the editor. At the time of this writing, however, that tool's eventual location in the Perl distribution was uncertain.

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.

38.2.8 The Perl Profiler

If you wish to supply an alternative debugger for Perl to run, just invoke your script with a colon and a package argument given to the **-d** flag. The most popular alternative debuggers for Perl is the Perl profiler. Devel::DProf is now included with the standard Perl distribution. To profile your Perl program in the file *mycode.pl*, just type:

```
    $ perl -d:DProf mycode.pl
```

When the script terminates the profiler will dump the profile information to a file called *tmon.out*. A tool like **dprofpp**, also supplied with the standard Perl distribution, can be used to interpret the information in that profile.

38.3 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 Debugging regular expressions in *perldebguts*.

38.4 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 Debugging Perl memory usage in *perldebguts* for the details.

38.5 SEE ALSO

You did try the **-w** switch, didn't you?

perldebtut, *perldebguts*, *re*, *DB*, *Devel::DProf*, *dprofpp*, *Dumpvalue*, and *perlrun*.

38.6 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.

Chapter 39

perlvar

Perl predefined variables

39.1 DESCRIPTION

39.1.1 Predefined Names

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**. In general, it's best to use the

```
    use English '-no_match_vars';
```

invocation if you don't need $PREMATCH, $MATCH, or $POSTMATCH, as it avoids a certain performance hit with the use of regular expressions. See *English*.

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..5){
        nasty_break();
        print "$_ ";
    }
    sub nasty_break {
        $_ = 5;
        # do something with $_
    }
```

You probably expect this code to print:

```
    1 2 3 4 5
```

but instead you get:

```
    5 5 5 5 5
```

Why? Because nasty_break() modifies `$_` without localizing it first. The fix is to add local():

```
        local $_ = 5;
```

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.

The following list is ordered by scalar variables first, then the arrays, then the hashes.

$ 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:

- Various unary functions, including functions like ord() and int(), as well as the all file tests (`-f`, `-d`) except for `-t`, which defaults to STDIN.
- Various list functions like print() and unlink().
- The pattern matching operations `m//`, `s///`, and `tr///` 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 default place to put an input record when a `<FH>` operation's result is tested by itself as the sole criterion of a `while` test. Outside a `while` test, this will not happen.

(Mnemonic: underline is understood in certain operations.)

$ a

$ b

Special package variables when using sort(), see sort in *perlfunc*. 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.

$ <*digits*>

Contains the subpattern from the corresponding set of capturing parentheses from the last pattern match, not counting patterns matched in nested blocks that have been exited already. (Mnemonic: like \digits.) These variables are all read-only and dynamically scoped to the current BLOCK.

$ 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). (Mnemonic: like & in some editors.) This variable is read-only and dynamically scoped to the current BLOCK.

The use of this variable anywhere in a program imposes a considerable performance penalty on all regular expression matches. See BUGS.

$ 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). (Mnemonic: ‘ often precedes a quoted string.) This variable is read-only.

The use of this variable anywhere in a program imposes a considerable performance penalty on all regular expression matches. See BUGS.

$ 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). (Mnemonic: ’ often follows a quoted string.) Example:

```
local $_ = 'abcdefghi';
/def/;
print "$`:$&:$'\n";         # prints abc:def:ghi
```

This variable is read-only and dynamically scoped to the current BLOCK.

The use of this variable anywhere in a program imposes a considerable performance penalty on all regular expression matches. See BUGS.

$ 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 = $+);
```

(Mnemonic: be positive and forward looking.) This variable is read-only and dynamically scoped to the current BLOCK.

$ ˆN

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. (Mnemonic: the (possibly) Nested parenthesis that most recently closed.)

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 is dynamically scoped to the current BLOCK.

@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.

$ *

Set to a non-zero integer value to do multi-line matching within a string, 0 (or undefined) to tell Perl that it can assume that strings contain a single line, for the purpose of optimizing pattern matches. Pattern matches on strings containing multiple newlines can produce confusing results when `$*` is 0 or undefined. Default is undefined. (Mnemonic: * matches multiple things.) This variable influences the interpretation of only ˆ and `$`. A literal newline can be searched for even when `$* == 0`.

Use of `$*` is deprecated in modern Perl, supplanted by the `/s` and `/m` modifiers on pattern matching.

Assigning a non-numerical value to `$*` triggers a warning (and makes `$*` act if `$* == 0`), while assigning a numerical value to `$*` makes that an implicit `int` is applied on the value.

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 I/O Operators in *perlop*. Because <> never does an explicit close, line numbers increase across ARGV files (but see examples in eof in *perlfunc*).

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. (Mnemonic: / delimits line boundaries when quoting poetry.)

```
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. 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 bytes from FILE. 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.

On VMS, record reads are done with the equivalent of `sysread`, so it's best not to mix record and non-record reads on the same file. (This is unlikely to be a problem, because any file you'd want to read in record mode is probably unusable in line mode.) Non-VMS systems do normal I/O, so it's safe to mix record and non-record reads of a file.

See also Newlines in *perlport*. Also see `$.`.

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 getc in *perlfunc* for that. (Mnemonic: when you want your pipes to be piping hot.)

IO::Handle->output_field_separator EXPR

$ OUTPUT_FIELD_SEPARATOR

$ OFS

$,

The output field separator for the print operator. Ordinarily the print operator simply prints out its arguments without further adornment. To get behavior more like **awk**, set this variable as you would set **awk**'s OFS variable to specify what is printed between fields. (Mnemonic: what is printed when there is a "," in your print statement.)

IO::Handle->output_record_separator EXPR

$ OUTPUT_RECORD_SEPARATOR

$ ORS

**$ **

The output record separator for the print operator. Ordinarily the print operator simply prints out its arguments as is, with no trailing newline or other end-of-record string added. To get behavior more like **awk**, set this variable as you would set **awk**'s ORS variable to specify what is printed at the end of the print. (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.)

$ LIST_SEPARATOR

$ "

This is like `$,` except that it applies to array and slice values interpolated into a double-quoted string (or similar interpreted string). Default is a space. (Mnemonic: obvious, I think.)

$ SUBSCRIPT_SEPARATOR

$ SUBSEP

$;

The subscript separator for multidimensional array emulation. If you refer to a hash element as

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

it really means

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

But don't put

```
    @foo{$a,$b,$c}      # a slice--note the @
```

which means

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

Default is "\034", the same as SUBSEP in **awk**. If your keys contain binary data there might not be any safe value for `$;`. (Mnemonic: comma (the syntactic subscript separator) is a semi-semicolon. Yeah, I know, it's pretty lame, but `$,` is already taken for something more important.)

Consider using "real" multidimensional arrays as described in *perllol*.

$ #

The output format for printed numbers. This variable is a half-hearted attempt to emulate **awk**'s OFMT variable. There are times, however, when **awk** and Perl have differing notions of what counts as numeric. The initial value is "%.*n*g", where *n* is the value of the macro DBL_DIG from your system's *float.h*. This is different from **awk**'s default OFMT setting of "%.6g", so you need to set `$#` explicitly to get **awk**'s value. (Mnemonic: # is the number sign.)

Use of `$#` is deprecated.

HANDLE->format_page_number(EXPR)

$ FORMAT_PAGE_NUMBER

$ %

The current page number of the currently selected output channel. Used with formats. (Mnemonic: % is page number in **nroff**.)

HANDLE->format_lines_per_page(EXPR)

$ FORMAT_LINES_PER_PAGE

$ =

The current page length (printable lines) of the currently selected output channel. Default is 60. Used with formats. (Mnemonic: = has horizontal lines.)

HANDLE->format_lines_left(EXPR)

$ FORMAT_LINES_LEFT

$ -

The number of lines left on the page of the currently selected output channel. Used with formats. (Mnemonic: lines_on_page - lines_printed.)

@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])`

HANDLE->format_name(EXPR)

$ FORMAT_NAME

$ ˜

The name of the current report format for the currently selected output channel. Default is the name of the filehandle. (Mnemonic: brother to `$^`.)

HANDLE->format_top_name(EXPR)

$ FORMAT_TOP_NAME

$ ˆ

The name of the current top-of-page format for the currently selected output channel. Default is the name of the filehandle with _TOP appended. (Mnemonic: points to top of page.)

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 ˆ) in a format. Default is " \n-", to break on whitespace or hyphens. (Mnemonic: a "colon" in poetry is a part of a line.)

IO::Handle->format_formfeed EXPR

$ FORMAT_FORMFEED

$ ˆL

What formats output as a form feed. Default is \f.

$ ACCUMULATOR

$ ˆA

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 *perlform* and formline() in *perlfunc*.

$ 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 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. (Mnemonic: similar to **sh** and **ksh**.)

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 `$?` reflect the actual VMS exit status, instead of the default emulation of POSIX status; see `$?` in *perlvms* for details.

Also see Error Indicators.

$ {^ENCODING}

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*. The direct manipulation of this variable is highly discouraged. See *encoding* for more details.

$ OS_ERROR

$ ERRNO

$!

If used numerically, yields the current value of the C `errno` variable, or in other words, if a system or library call fails, it sets this variable. This means that the value of `$!` is meaningful only *immediately* after a **failure**:

```
if (open(FH, $filename)) {
    # Here $! is meaningless.
    ...
} else {
    # ONLY here is $! meaningful.
    ...
    # Already here $! might be meaningless.
}
# Since here we might have either success or failure,
# here $! is meaningless.
```

In the above *meaningless* stands for anything: zero, non-zero, `undef`. A successful system or library call does **not** set the variable to zero.

If used as a string, yields the corresponding system error string. You can assign a number to `$!` to set *errno* if, for instance, you want `"$!"` to return the string for error *n*, or you want to set the exit value for the die() operator. (Mnemonic: What just went bang?)

Also see Error Indicators.

%!

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 above for the validity of $!.

$ EXTENDED_OS_ERROR

$ ˆE

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, $ˆE is always just the same as $!.

Under VMS, $ˆE 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, $ˆE is set to the error code of the last call to OS/2 API either via CRT, or directly from perl.

Under Win32, $ˆE 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 $ˆE. 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 $ˆE, also. (Mnemonic: Extra error explanation.)

Also see Error Indicators.

$ 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). (Mnemonic: Where was the syntax error "at"?)

Warning messages are not collected in this variable. You can, however, set up a routine to process warnings by setting $SIG{__WARN__} as described below.

Also see Error Indicators.

$ PROCESS_ID

$ PID

$ $

The process number of the Perl running this script. You should consider this variable read-only, although it will be altered across fork() calls. (Mnemonic: same as shells.)

Note for Linux users: on Linux, the C functions `getpid()` and `getppid()` return different values from different threads. In order to be portable, this behavior is not reflected by $$, whose value remains consistent across threads. If you want to call the underlying `getpid()`, you may use the CPAN module `Linux::Pid`.

$ REAL_USER_ID

$ UID

$ <

The real uid of this process. (Mnemonic: it's the uid you came *from*, if you're running setuid.) You can change both the real uid and the effective uid at the same time by using POSIX::setuid().

$ EFFECTIVE_USER_ID

$ EUID

$ >

The effective uid of this process. Example:

```
    $< = $>;            # set real to effective uid
    ($<,$>) = ($>,$<);  # swap real and effective uid
```

You can change both the effective uid and the real uid at the same time by using POSIX::setuid().

(Mnemonic: it's the uid you went *to*, if you're running setuid.) `$<` and `$>` can be swapped only on machines supporting setreuid().

$ 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.

You can change both the real gid and the effective gid at the same time by using POSIX::setgid().

(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).

(Mnemonic: parentheses are used to *group* things. The effective gid is the group that's *right* for you, if you're running setgid.)

`$<`, `$>`, `$(` and `$)` can be set only on machines that support the corresponding *set[re][ug]id()* routine. `$(` and `$)` can be swapped only on machines supporting setregid().

$ PROGRAM_NAME

$ 0

Contains the name of the program being executed.

On some (read: 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. (Mnemonic: same as **sh** and **ksh**.)

Note that there are platform specific limitations on the 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 the view of `$0` the other threads have will not change since they have their own copies of it.

$ [

The index of the first element in an array, and of the first character in a substring. 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. (Mnemonic: [begins subscripts.)

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.

Note that, unlike other compile-time directives (such as *strict*), assignment to $[can be seen from outer lexical scopes in the same file. However, you can use local() on it to strictly bound its value to a lexical block.

$]

The version + patchlevel / 1000 of the Perl interpreter. This variable can be used to determine whether the Perl interpreter executing a script is in the right range of versions. (Mnemonic: Is this version of perl in the right bracket?) Example:

```
warn "No checksumming!\n" if $] < 3.019;
```

See also the documentation of `use VERSION` and `require VERSION` for a convenient way to fail if the running Perl interpreter is too old.

When testing the variable, to steer clear of floating point inaccuracies you might want to prefer the inequality tests < and > to the tests containing equivalence: <=, ==, and >=.

The floating point representation can sometimes lead to inaccurate numeric comparisons. See `$^V` for a more modern representation of the Perl version that allows accurate string comparisons.

$ 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. See *perlcc*. Setting `$^C = 1` is similar to calling `B::minus_c`.

$ DEBUGGING

$ ^D

The current value of the debugging flags. (Mnemonic: value of **-D** switch.) May be read or set. Like its command-line equivalent, you can use numeric or symbolic values, eg `$^D = 10` or `$^D = "st"`.

$ 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().

$ ^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 $ˆH 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 $ˆH.

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 $ˆH will therefore be visible only while the body of foo() is being compiled.

Substitution of the above BEGIN 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 }
```

%ˆH

WARNING: This variable is strictly for internal use only. Its availability, behavior, and contents are subject to change without notice.

The %ˆH hash provides the same scoping semantic as $ˆH. This makes it useful for implementation of lexically scoped pragmas.

$ INPLACE_EDIT

$ ˆI

The current value of the inplace-edit extension. Use `undef` to disable inplace editing. (Mnemonic: value of **-i** switch.)

$ ˆM

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 enable this option. To discourage casual use of this advanced feature, there is no English long name for this variable.

$ OSNAME

$ ˆO

The name of the operating system under which this copy of Perl was built, as determined during the configuration process. The value is identical to `$Config{'osname'}`. See also *Config* and the **-V** command-line switch documented in *perlrun*.

In Windows platforms, $ˆO 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 *perlport*) to distinguish between the variants.

$ {^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.

$ PERLDB

$ ^P

The internal variable for debugging support. The meanings of the various bits are subject to change, but currently indicate:

1. x01
   Debug subroutine enter/exit.
2. x02
   Line-by-line debugging.
3. x04
   Switch off optimizations.
4. x08
   Preserve more data for future interactive inspections.
5. x10
   Keep info about source lines on which a subroutine is defined.
6. x20
   Start with single-step on.
7. x40
   Use subroutine address instead of name when reporting.
8. x80
   Report `goto &subroutine` as well.
9. x100
   Provide informative "file" names for evals based on the place they were compiled.
10. x200
    Provide informative names to anonymous subroutines based on the place they were compiled.
11. x400
    Debug assertion subroutines enter/exit.

Some bits may be relevant at compile-time only, some at run-time only. This is a new mechanism and the details may change.

$ LAST_REGEXP_CODE_RESULT

$ ^R

The result of evaluation of the last successful `(?{ code })` regular expression assertion (see *perlre*). May be written to.

$ EXCEPTIONS_BEING_CAUGHT

$ ^S

Current state of the interpreter.

```
$^S         State
---------   -------------------
undef       Parsing module/eval
true (1)    Executing an eval
false (0)   Otherwise
```

The first state may happen in $SIG{__DIE__} and $SIG{__WARN__} handlers.

$ BASETIME

$ ˆT

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.

$ {ˆTAINT}

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**).

$ {ˆUNICODE}

Reflects certain Unicode settings of Perl. See *perlrun* documentation for the `-C` switch for more information about the possible values. This variable is set during Perl startup and is thereafter read-only.

$ PERL_VERSION

$ ˆV

The revision, version, and subversion of the Perl interpreter, represented as a string composed of characters with those ordinals. Thus in Perl v5.6.0 it equals `chr(5) . chr(6) . chr(0)` and will return true for `$ˆV eq v5.6.0`. Note that the characters in this string value can potentially be in Unicode range.

This can be used to determine whether the Perl interpreter executing a script is in the right range of versions. (Mnemonic: use ˆV for Version Control.) Example:

```
    warn "No \"our\" declarations!\n" if $^V and $^V lt v5.6.0;
```

To convert `$ˆV` into its string representation use sprintf()'s `"%vd"` conversion:

```
    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 an older representation of the Perl version.

$ WARNING

$ ˆW

The current value of the warning switch, initially true if **-w** was used, false otherwise, but directly modifiable. (Mnemonic: related to the **-w** switch.) See also *warnings*.

$ {ˆWARNING_BITS}

The current set of warning checks enabled by the `use warnings` pragma. See the documentation of `warnings` for more details.

$ EXECUTABLE_NAME

$ ˆX

The name used to execute the current copy of Perl, from C's `argv[0]`.

Depending on the host operating system, the value of $ˆX 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 $ˆX is in PATH. For VMS, the value may or may not include a version number.

You usually can use the value of $ˆX 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 $ˆX 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 $ˆX to a path name, use the following statements:

# Build up a set of file names (not command names). use Config; $this_perl = $ˆX; if ($ˆO 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 $ˆX. The following statements accomplish this goal, and produce a pathname that can be invoked as a command or referenced as a file.

```
use Config;
$secure_perl_path = $Config{perlpath};
if ($^O ne 'VMS')
   {$secure_perl_path .= $Config{_exe}
        unless $secure_perl_path =~ m/$Config{_exe}$/i;}
```

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`.

$ 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` for the command name.

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 *perlrun* for the **-i** switch.

@F

The array @F contains the fields of each line read in when autosplit mode is turned on. See *perlrun* 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 require in *perlfunc* for details.

@_

Within a subroutine the array @_ contains the parameters passed to that subroutine. See *perlsub*.

%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 require in *perlfunc* 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.

%ENV

$ ENV{expr}

The hash %ENV contains your current environment. Setting a value in `ENV` changes the environment for any child processes you subsequently fork() off.

%SIG

$ SIG{expr}

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 *perlipc* 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 5.8.0 from immediate (also known as "unsafe") to deferred, also known as "safe signals". See *perlipc* 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;
```

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`, 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...
         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.

See die in *perlfunc*, warn in *perlfunc*, eval in *perlfunc*, and *warnings* for additional information.

39.1.2 Error Indicators

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:

```
    eval q{
        open my $pipe, "/cdrom/install |" or die $!;
        my @res = <$pipe>;
        close $pipe or die "bad pipe: $?, $!";
    };
```

After execution of this statement all 4 variables may have been set.

`$@` 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.)

When the eval() expression above is executed, open(), `<PIPE>`, and `close` are translated to calls in the C run-time library and thence to the operating system kernel. `$!` is set to the C library's `errno` if one of these calls fails.

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 `$@`, `$!`, `$^E`, and `$?`.

39.1.3 Technical Note on 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 *perlmod*.

Perl variable names may also be a sequence of digits or a single punctuation or control character. 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.

Finally, new in Perl 5.6, Perl variable names may be alphanumeric strings that begin with control characters (or better yet, a caret). 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 new special `${^_XYZ}` variables are always taken to be in package `main`, regardless of any `package` declarations presently in scope.

39.2 BUGS

Due to an unfortunate accident of Perl's implementation, `use English` imposes a considerable performance penalty on all regular expression matches in a program, regardless of whether they occur in the scope of `use English`. For that reason, saying `use English` in libraries is strongly discouraged. See the Devel::SawAmpersand module documentation from CPAN (http://www.cpan.org/modules/by-module/Devel/) for more information.

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.

Chapter 40

perlre

Perl regular expressions

40.1 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 *perlrequick*, and a longer tutorial introduction is available in *perlretut*.

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 Regexp Quote-Like Operators in *perlop*.

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 Regexp Quote-Like Operators in *perlop* and Gory details of parsing quoted constructs in *perlop*.

i

Do case-insensitive pattern matching.

If `use locale` is in effect, the case map is taken from the current locale. See *perllocale*.

m

Treat string as multiple lines. That is, change "^" and "$" from matching the start or end of the string to matching the start or end of any line anywhere 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.

The `/s` and `/m` modifiers both override the `$*` setting. That is, no matter what `$*` contains, `/s` without `/m` will force "^" to match only at the beginning of the string and "$" to match only at the end (or just before a newline at the end) of the string. Together, as /ms, they let the "." match any character whatsoever, while still allowing "^" and "$" to match, respectively, just after and just before newlines within the string.

x

Extend your pattern's legibility by permitting whitespace and comments.

These are usually written as "the `/x` modifier", even though the delimiter in question might not really be a slash. Any of these modifiers may also be embedded within the regular expression itself using the `(?...)` construct. See below.

The `/x` modifier itself needs a little more explanation. It tells the regular expression parser to ignore whitespace that is neither backslashed nor within a character class. You can use this to break up your regular expression into (slightly) more readable parts. The `#` character is also treated as a metacharacter introducing a comment, just as in ordinary Perl code. This also means that if you want real whitespace or `#` characters in the pattern (outside a character class, where they are unaffected by `/x`), that you'll either have to escape them or encode them using octal or hex escapes. Taken together, these features go a long way towards making Perl's regular expressions more readable. Note that you have to be careful not to include the pattern delimiter in the comment–perl has no way of knowing you did not intend to close the pattern early. See the C-comment deletion code in *perlop*.

40.1.1 Regular Expressions

The patterns used in Perl pattern matching derive from supplied in the Version 8 regex routines. (The routines are derived (distantly) from Henry Spencer's freely redistributable reimplementation of the V8 routines.) See Version 8 Regular Expressions 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 line (or before newline at the end)
|   Alternation
()  Grouping
[]  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, 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 practice is now deprecated.)

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. The `/s` modifier also overrides the setting of `$*`, in case you have some (badly behaved) older code that sets it in another module.

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, it is treated as a regular character. In particular, the lower bound is not optional.) The "*" modifier is equivalent to `{0,}`, the "+" modifier to `{1,}`, and the "?" modifier to `{0,1}`. n and m are limited to 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
+?     Match 1 or more times
??     Match 0 or 1 time
{n}?   Match exactly n times
{n,}?  Match at least n times
{n,m}? Match at least n but not more than m times
```

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)
    \033        octal char (think of a PDP-11)
    \x1B        hex char
    \x{263a}    wide hex char         (Unicode SMILEY)
    \c[         control char
    \N{name}    named char
    \l          lowercase next char (think vi)
    \u          uppercase next char (think vi)
    \L          lowercase till \E (think vi)
    \U          uppercase till \E (think vi)
    \E          end case modification (think vi)
    \Q          quote (disable) pattern metacharacters till \E
```

If `use locale` is in effect, the case map used by `\l`, `\L`, `\u` and `\U` is taken from the current locale. See *perllocale*. For documentation of `\N{name}`, see *charnames*.

You cannot include a literal `$` or `@` within a `\Q` sequence. An unescaped `$` or `@` interpolates the corresponding variable, while escaping will cause the literal string `\$` to be matched. You'll need to write something like `m/\Quser\E\@\Qhost/`.

In addition, Perl defines the following:

```
    \w  Match a "word" character (alphanumeric plus "_")
    \W  Match a non-"word" character
    \s  Match a whitespace character
    \S  Match a non-whitespace character
    \d  Match a digit character
    \D  Match a non-digit character
    \pP Match P, named property.  Use \p{Prop} for longer names.
    \PP Match non-P
    \X  Match eXtended Unicode "combining character sequence",
        equivalent to (?:\PM\pM*)
    \C  Match a single C char (octet) even under Unicode.
        NOTE: breaks up characters into their UTF-8 bytes,
        so you may end up with malformed pieces of UTF-8.
        Unsupported in lookbehind.
```

A `\w` matches a single alphanumeric character (an alphabetic character, or a decimal digit) or `_`, not a whole word. Use `\w+` to match a string of Perl-identifier characters (which isn't the same as matching an English word). If `use locale` is in effect, the list of alphabetic characters generated by `\w` is taken from the current locale. See *perllocale*. You may use `\w`, `\W`, `\s`, `\S`, `\d`, and `\D` within character classes, but if you try to use them as endpoints of a range, that's not a range, the "-" is understood literally. If Unicode is in effect, `\s` matches also "\x{85}", "\x{2028}, and "\x{2029}", see *perlunicode* for more details about `\pP`, `\PP`, and `\X`, and *perluniintro* about Unicode in general. You can define your own `\p` and `\P` propreties, see *perlunicode*.

The POSIX character class syntax

```
    [:class:]
```

is also available. The available classes and their backslash equivalents (if available) are as follows:

```
    alpha
    alnum
    ascii
    blank              [1]
    cntrl
    digit       \d
    graph
    lower
    print
    punct
    space       \s     [2]
    upper
    word        \w     [3]
    xdigit
```

[1]

A GNU extension equivalent to `[ \t]`, 'all horizontal whitespace'.

[2]

Not exactly equivalent to `\s` since the `[[:space:]]` includes also the (very rare) 'vertical tabulator', "\ck", chr(11).

[3]

A Perl extension, see above.

For example use `[:upper:]` to match all the uppercase characters. Note that the `[]` are part of the `[::]` construct, not part of the whole character class. For example:

```
    [01[:alpha:]%]
```

matches zero, one, any alphabetic character, and the percentage sign.

The following equivalences to Unicode \p{} constructs and equivalent backslash character classes (if available), will hold:

```
    [:...:]     \p{...}         backslash

    alpha       IsAlpha
    alnum       IsAlnum
    ascii       IsASCII
    blank       IsSpace
    cntrl       IsCntrl
    digit       IsDigit        \d
    graph       IsGraph
    lower       IsLower
    print       IsPrint
    punct       IsPunct
    space       IsSpace
                IsSpacePerl    \s
    upper       IsUpper
    word        IsWord
    xdigit      IsXDigit
```

For example `[:lower:]` and `\p{IsLower}` are equivalent.

If the `utf8` pragma is not used but the `locale` pragma is, the classes correlate with the usual isalpha(3) interface (except for 'word' and 'blank').

The assumedly non-obviously named classes are:

cntrl

Any control character. Usually characters that don't produce output as such but instead control the terminal somehow: for example newline and backspace are control characters. All characters with ord() less than 32 are most often classified as control characters (assuming ASCII, the ISO Latin character sets, and Unicode), as is the character with the ord() value of 127 (`DEL`).

graph

Any alphanumeric or punctuation (special) character.

print

Any alphanumeric or punctuation (special) character or the space character.

punct

Any punctuation (special) character.

xdigit

Any hexadecimal digit. Though this may feel silly ([0-9A-Fa-f] would work just fine) it is included for completeness.

You can negate the [::] character classes by prefixing the class name with a '^'. This is a Perl extension. For example:

```
    POSIX       traditional Unicode

    [:^digit:]      \D      \P{IsDigit}
    [:^space:]      \S      \P{IsSpace}
    [:^word:]       \W      \P{IsWord}
```

Perl respects the POSIX standard in that POSIX character classes are only supported within a character class. The POSIX character classes [.cc.] and [=cc=] are recognized but **not** supported and trying to use them will cause an error.

Perl defines the following zero-width assertions:

```
    \b  Match a word boundary
    \B  Match a non-(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 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 Regexp Quote-Like Operators in *perlop*. 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 pos in *perlfunc*. Currently `\G` is only fully supported when anchored to the start of the pattern; while it is permitted to use it elsewhere, as in `/(?<=\G..)./g`, some such uses (`/.\G/g`, for example) currently cause problems, and it is recommended that you avoid such usage for now.

The bracketing construct `( ... )` creates capture buffers. To refer to the digit'th buffer use \<digit> within the match. Outside the match use "$" instead of "\". (The \<digit> notation works in certain circumstances outside the match. See the warning below about \1 vs $1 for details.) Referring back to another part of the match is called a *backreference*.

There is no limit to the number of captured substrings that you may use. However Perl also uses \10, \11, etc. as aliases for \010, \011, etc. (Recall that 0 means octal, so \011 is the character at number 9 in your coded character set; which would be the 10th character, a horizontal tab under 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.

Examples:

```
s/^([^ ]*) *([^ ]*)/$2 $1/;     # swap first two words

 if (/(.)\1/) {                 # find first doubled char
     print "'$1' is the first doubled character\n";
 }

if (/Time: (..):(..):(..)/) {   # parse out values
    $hours = $1;
    $minutes = $2;
    $seconds = $3;
}
```

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 `$^N` contains whatever was matched by the most-recently closed group (submatch). `$^N` can be used in extended patterns (see below), for example to assign a submatch to a variable.

The numbered match variables ($1, $2, $3, etc.) and the related punctuation set (`$+`, `$&`, `` $` ``, `$'`, and `$^N`) are all dynamically scoped until the end of the enclosing block or until the next successful match, whichever comes first. (See Compound Statements in *perlsyn*.)

NOTE: failed matches in Perl do not reset the match variables, which makes easier to write code that tests for a series of more specific cases and remembers the best match.

WARNING: 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 `` $` `` if you can, but if you can't (and some algorithms really appreciate them), once you've used them once, use them at will, because you've already paid the price. As of 5.005, `$&` is not so costly as the other two.

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 Gory details of parsing quoted constructs in *perlop*.

40.1.2 Extended Patterns

Perl also defines a consistent extension syntax for features not found in standard tools like **awk** and **lex**. The syntax 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. If the `/x` modifier enables whitespace formatting, a simple `#` will suffice. Note that Perl closes the comment as soon as it sees a `)`, so there is no way to put a literal `)` in the comment.

`(?imsx-imsx)`

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, read in as an argument, are specified in a table somewhere, etc. Consider the case that some of which want to be case sensitive and some do not. The case insensitive ones need to include merely `(?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+ \1
```

will match a repeated (*including the case*!) word `blah` in any case, assuming `x` modifier, and no `i` modifier outside this group.

`(?:pattern)`

`(?imsx-imsx: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 `(?imsx-imsx)`. For example,

```
/(?s-i:more.*than).*million/i
```

is equivalent to the more verbose

```
/(?:(?s-i)more.*than).*million/i
```

(?=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. You would have to do something like `/(?!foo)...bar/` for that. We say "like" because there's the case of your "bar" not having three characters before it. You could cover that this way: `/(?:(?!foo)...|^.{0,2})bar/`. Sometimes it's still easier just to say:

```
if (/bar/ && $` !~ /foo$/)
```

For look-behind see below.

(?<=pattern)

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.

(?<!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.

(?{ code })

WARNING: This extended regular expression feature is considered highly experimental, and may be changed or deleted without notice.

This zero-width assertion evaluates any embedded Perl code. It always succeeds, and its `code` is not interpolated. Currently, the rules to determine where the `code` ends are somewhat convoluted.

This feature can be used together with the special variable `$^N` 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";
```

Inside the `(?{...})` 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 withing this string.

The `code` is properly scoped in the following sense: If the assertion is backtracked (compare §40.1.3), all changes introduced after `local`ization are undone, so that

```
$_ = '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 set `$res = 4`. Note that after the match, $cnt returns to the globally introduced value, because the scopes that restrict `local` operators are unwound.

This assertion may be used as 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 §40.1.3.

For reasons of security, this construct is forbidden if the regular expression involves run-time interpolation of variables, unless the perilous `use re 'eval'` pragma has been used (see *re*), or the variables contain results of `qr//` operator (see qr/STRING/imosx in *perlop*).

This restriction is because of the wide-spread and remarkably convenient custom of using run-time determined strings as patterns. For example:

```
    $re = <>;
    chomp $re;
    $string =~ /$re/;
```

Before Perl knew how to execute interpolated code within a pattern, this operation was completely safe from a security point of view, although it could raise an exception from an illegal pattern. If you turn on the `use re 'eval'`, though, it is no longer secure, so you should only do so if you are also using taint checking. Better yet, use the carefully constrained evaluation within a Safe compartment. See *perlsec* for details about both these mechanisms.

`(??{ code })`

WARNING: This extended regular expression feature is considered highly experimental, and may be changed or deleted without notice. A simplified version of the syntax may be introduced for commonly used idioms.

This is a "postponed" regular subexpression. The `code` is evaluated at run time, at the moment this subexpression may match. The result of evaluation is considered as a regular expression and matched as if it were inserted instead of this construct.

The `code` is not interpolated. As before, the rules to determine where the `code` ends are currently somewhat convoluted.

The following pattern matches a parenthesized group:

```
  $re = qr{
             \(
             (?:
                (?> [^()]+ )    # Non-parens without backtracking
              |
                (??{ $re })     # Group with matching parens
             )*
             \)
          }x;
```

`(?>pattern)`

WARNING: This extended regular expression feature is considered highly experimental, and may be changed or deleted without notice.

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 §40.1.3). 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 §40.1.3). In particular, `a*` inside `a*ab` will match fewer characters than a standalone `a*`, since this makes the tail match.

An effect similar to `(?>pattern)` may be achieved by writing `(?=(pattern))\1`. This matches the same substring as a standalone `a+`, and the following `\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 this pattern 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.

(?(condition)yes-pattern|no-pattern)

(?(condition)yes-pattern)

WARNING: This extended regular expression feature is considered highly experimental, and may be changed or deleted without notice.

Conditional expression. `(condition)` should be either an integer in parentheses (which is valid if the corresponding pair of parentheses matched), or look-ahead/look-behind/evaluate zero-width assertion.

For example:

```
        m{ ( \( )?
           [^()]+
           (?(1) \) )
         }x
```

matches a chunk of non-parentheses, possibly included in parentheses themselves.

40.1.3 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 Combining pieces together.

A fundamental feature of regular expression matching involves the notion called *backtracking*, which is currently used (when needed) by all regular 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 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)>). 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)>.

40.1.4 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 "\"). 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 one 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 coded character sets.) Also, if you try to use the character classes `\w`, `\W`, `\s`, `\S`, `\d`, or `\D` as endpoints of a range, that's not a range, the "-" is understood literally.

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 ([0-9]). Anything else is unsafe. If in doubt, spell out the character sets 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 octal digits, matches the character whose coded character set value is *nnn*. Similarly, \x*nn*, where *nn* are hexadecimal digits, matches the character whose numeric value 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 ("(", "[", or the beginning of the pattern) up to the first "|", and the last alternative contains everything from the last "|" to the next 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*. 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\1\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.

40.1.5 Warning on \1 vs $ 1

Some people get too used to writing things like:

```
    $pattern =~ s/(\W)/\\\1/g;
```

This is grandfathered 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///`.

40.1.6 Repeated patterns matching 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?` can match 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 * modifier. 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 modifiers `*+{}`, 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;
```

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 §40.1.3), 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.

40.1.7 Combining pieces together

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 §40.1.3). 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 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 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 })

The ordering is the same as for the regular expression which is the result of EXPR.

(?(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.

40.1.8 Creating custom RE engines

Overloaded constants (see *overload*) provide a simple way to extend the functionality of the RE engine.

Suppose that we want to enable a new RE escape-sequence \Y| which matches at boundary between white-space 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]'"}
```

```
    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|/;
```

40.2 BUGS

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.

40.3 SEE ALSO

perlrequick.

perlretut.

Regexp Quote-Like Operators in *perlop*.

Gory details of parsing quoted constructs in *perlop*.

perlfaq6.

pos in *perlfunc*.

perllocale.

perlebcdic.

Mastering Regular Expressions by Jeffrey Friedl, published by O'Reilly and Associates.

Chapter 41

perlreref

Perl Regular Expressions Reference

41.1 DESCRIPTION

This is a quick reference to Perl's regular expressions. For full information see *perlre* and *perlop*, as well as the §41.3 section in this document.

41.1.1 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/igmsoxc searches a string for a pattern match,
   applying the given options.

      i  case-Insensitive
      g  Global - all occurrences
      m  Multiline mode - ^ and $ match internal lines
      s  match as a Single line - . matches \n
      o  compile pattern Once
      x  eXtended legibility - free whitespace and comments
      c  don't reset pos on failed matches when using /g

   If 'pattern' is an empty string, the last I<successfully> matched
   regex is used. Delimiters other than '/' may be used for both this
   operator and the following ones.

qr/pattern/imsox 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/igmsoxe substitutes matches of
     'pattern' with 'replacement'. Modifiers as for m//
     with one addition:

        e  Evaluate replacement as an expression

     '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 L<reset|perlfunc/reset>.
```

41.1.2 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
   \1, \2 ...  The text from the Nth group
```

41.1.3 ESCAPE SEQUENCES

These work as in normal strings.

```
   \a       Alarm (beep)
   \e       Escape
   \f       Formfeed
   \n       Newline
   \r       Carriage return
   \t       Tab
   \038     Any octal ASCII value
   \x7f     Any hexadecimal ASCII value
   \x{263a} A wide hexadecimal value
   \cx      Control-x
   \N{name} A named character

   \l  Lowercase next character
   \u  Titlecase next character
   \L  Lowercase until \E
   \U  Uppercase until \E
   \Q  Disable pattern metacharacters until \E
   \E  End case modification
```

For Titlecase, see Titlecase.

This one works differently from normal strings:

```
   \b  An assertion, not backspace, except in a character class
```

41.1.4 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 work within or without a character class. The first six are locale aware, all are Unicode aware. The default character class equivalent are given. See *perllocale* and *perlunicode* for details.

```
\d      A digit                     [0-9]
\D      A nondigit                  [^0-9]
\w      A word character            [a-zA-Z0-9_]
\W      A non-word character        [^a-zA-Z0-9_]
\s      A whitespace character      [ \t\n\r\f]
\S      A non-whitespace character  [^ \t\n\r\f]

\C      Match a byte (with Unicode, '.' matches a character)
\pP     Match P-named (Unicode) property
\p{...} Match Unicode property with long name
\PP     Match non-P
\P{...} Match lack of Unicode property with long name
\X      Match extended unicode sequence
```

POSIX character classes and their Unicode and Perl equivalents:

```
alnum   IsAlnum              Alphanumeric
alpha   IsAlpha              Alphabetic
ascii   IsASCII              Any ASCII char
blank   IsSpace  [ \t]       Horizontal whitespace (GNU extension)
cntrl   IsCntrl              Control characters
digit   IsDigit  \d          Digits
graph   IsGraph              Alphanumeric and punctuation
lower   IsLower              Lowercase chars (locale and Unicode aware)
print   IsPrint              Alphanumeric, punct, and space
punct   IsPunct              Punctuation
space   IsSpace  [\s\ck]     Whitespace
        IsSpacePerl   \s     Perl's whitespace definition
upper   IsUpper              Uppercase chars (locale and Unicode aware)
word    IsWord   \w          Alphanumeric plus _ (Perl extension)
xdigit  IsXDigit [0-9A-Fa-f] Hexadecimal digit
```

Within a character class:

```
 POSIX       traditional   Unicode
 [:digit:]       \d        \p{IsDigit}
 [:^digit:]      \D        \P{IsDigit}
```

41.1.5 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 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
```

41.1.6 QUANTIFIERS

Quantifiers are greedy by default – match the **longest** leftmost.

```
   Maximal Minimal Allowed range
   ------- ------- -------------
   {n,m}   {n,m}?  Must occur at least n times but no more than m times
   {n,}    {n,}?   Must occur at least n times
   {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})
```

There is no quantifier {,n} – that gets understood as a literal string.

41.1.7 EXTENDED CONSTRUCTS

```
   (?#text)          A comment
   (?imxs-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
   (?{ code })       Embedded code, return value becomes $^R
   (??{ code })      Dynamic regex, return value used as regex
   (?(cond)yes|no)   cond being integer corresponding to capturing parens
   (?(cond)yes)         or a lookaround/eval zero-width assertion
```

41.1.8 VARIABLES

```
   $_    Default variable for operators to use
   $*    Enable multiline matching (deprecated; not in 5.9.0 or later)

   $&    Entire matched string
   $`    Everything prior to matched string
   $'    Everything after to matched string
```

The use of those last three will slow down **all** regex use within your program. Consult *perlvar* for `@LAST_MATCH_START` to see equivalent expressions that won't cause slow down. See also *Devel::SawAmpersand*.

```
   $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
```

Captured groups are numbered according to their *opening* paren.

41.1.9 FUNCTIONS

```
lc          Lowercase a string
lcfirst     Lowercase first char of a string
uc          Uppercase a string
ucfirst     Titlecase first char of a string

pos         Return or set current match position
quotemeta   Quote metacharacters
reset       Reset ?pattern? status
study       Analyze string for optimizing matching

split       Use regex to split a string into parts
```

The first four of these are like the escape sequences `\L`, `\l`, `\U`, and `\u`. For Titlecase, see Titlecase.

41.1.10 TERMINOLOGY

Titlecase

Unicode concept which most often is equal to uppercase, but for certain characters like the German "sharp s" there is a difference.

41.2 AUTHOR

Iain Truskett.

This document may be distributed under the same terms as Perl itself.

41.3 SEE ALSO

- *perlretut* for a tutorial on regular expressions.
- *perlrequick* for a rapid tutorial.
- *perlre* for more details.
- *perlvar* for details on the variables.
- *perlop* for details on the operators.
- *perlfunc* for details on the functions.
- *perlfaq6* for FAQs on regular expressions.
- The *re* module to alter behaviour and aid debugging.
- Debugging regular expressions in *perldebug*
- *perluniintro*, *perlunicode*, *charnames* and *locale* for details on regexes and internationalisation.
- *Mastering Regular Expressions* by Jeffrey Friedl (*http://regex.info/*) for a thorough grounding and reference on the topic.

41.4 THANKS

David P.C. Wollmann, Richard Soderberg, Sean M. Burke, Tom Christiansen, Jim Cromie, and Jeffrey Goff for useful advice.

Chapter 42

perlref

Perl references and nested data structures

42.1 NOTE

This is complete documentation about all aspects of references. For a shorter, tutorial introduction to just the essential features, see *perlreftut*.

42.2 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 Two-Phased Garbage Collection in *perlobj* for a detailed explanation.) If that thing happens to be an object, the object is destructed. See *perlobj* 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: 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.

42.2.1 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 *perlobj*.

   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() and called indirectly:

   ```
   $objref = new Doggie (Tail => 'short', Ears => 'long');
   ```

   But don't have to be:

   ```
   $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};
   ```

   All of these are self-explanatory except for `*foo{IO}`. It returns the IO handle, used for file handles (open in *perlfunc*), sockets (socket in *perlfunc* and socketpair in *perlfunc*), and directory handles (opendir in *perlfunc*). 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{IO}` is an alternative to the `*HANDLE` mechanism given in Typeglobs and Filehandles in *perldata* 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>;
   }
   ```

42.2.2 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 *perlobj*, 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 ref in *perlfunc* 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 *perlobj*.

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";
```

42.2.3 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() (as in Perl 4)
    $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.

42.2.4 Not-so-symbolic references

A new feature contributing to readability in perl version 5.001 is that the brackets around a symbolic reference behave more like quotes, just as they always have within a string. That is,

```
    $push = "pop on ";
    print "${push}over";
```

has always meant to print "pop on over", even though push is a reserved word. This has been generalized to work the same outside of quotes, so that

```
    print ${push} . "over";
```

and even

```
    print ${ push } . "over";
```

will have the same effect. (This would have been a syntax error in Perl 5.000, though Perl 4 allowed it in the spaceless form.) 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, we've applied the same rule 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.

42.2.5 Pseudo-hashes: Using an array as a hash

WARNING: This section describes an experimental feature. Details may change without notice in future versions.

NOTE: The current user-visible implementation of pseudo-hashes (the weird use of the first array element) is deprecated starting from Perl 5.8.0 and will be removed in Perl 5.10.0, and the feature will be implemented differently. Not only is the current interface rather ugly, but the current implementation slows down normal array and hash use quite noticeably. The 'fields' pragma interface will remain available.

Beginning with release 5.005 of Perl, you may use an array reference in some contexts that would normally require a hash reference. This allows you to access array elements using symbolic names, as if they were fields in a structure.

For this to work, the array must contain extra information. The first element of the array has to be a hash reference that maps field names to array indices. Here is an example:

```
    $struct = [{foo => 1, bar => 2}, "FOO", "BAR"];

    $struct->{foo};  # same as $struct->[1], i.e. "FOO"
    $struct->{bar};  # same as $struct->[2], i.e. "BAR"

    keys %$struct;   # will return ("foo", "bar") in some order
    values %$struct; # will return ("FOO", "BAR") in same some order

    while (my($k,$v) = each %$struct) {
       print "$k => $v\n";
    }
```

Perl will raise an exception if you try to access nonexistent fields. To avoid inconsistencies, always use the fields::phash() function provided by the `fields` pragma.

```
    use fields;
    $pseudohash = fields::phash(foo => "FOO", bar => "BAR");
```

For better performance, Perl can also do the translation from field names to array indices at compile time for typed object references. See *fields*.

There are two ways to check for the existence of a key in a pseudo-hash. The first is to use exists(). This checks to see if the given field has ever been set. It acts this way to match the behavior of a regular hash. For instance:

```
    use fields;
    $phash = fields::phash([qw(foo bar pants)], ['FOO']);
    $phash->{pants} = undef;

    print exists $phash->{foo};    # true, 'foo' was set in the declaration
    print exists $phash->{bar};    # false, 'bar' has not been used.
    print exists $phash->{pants};  # true, your 'pants' have been touched
```

The second is to use exists() on the hash reference sitting in the first array element. This checks to see if the given key is a valid field in the pseudo-hash.

```
    print exists $phash->[0]{bar};      # true, 'bar' is a valid field
    print exists $phash->[0]{shoes};# false, 'shoes' can't be used
```

delete() on a pseudo-hash element only deletes the value corresponding to the key, not the key itself. To delete the key, you'll have to explicitly delete it from the first hash element.

```
    print delete $phash->{foo};     # prints $phash->[1], "FOO"
    print exists $phash->{foo};     # false
    print exists $phash->[0]{foo};  # true, key still exists
    print delete $phash->[0]{foo};  # now key is gone
    print $phash->{foo};            # runtime exception
```

42.2.6 Function Templates

As explained above, a closure is an anonymous function with access to the lexical variables visible when that function was compiled. 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 type–like those in the `for` loop above–only works with closures, not general subroutines. In the general case, then, named subroutines do not nest properly, although anonymous ones do. 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". 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 closure (anonymous subroutine). But when it does, it has normal access to the lexical variable $x from the scope of outer().

This has the interesting effect of creating a function local to another function, something not normally supported in Perl.

42.3 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.

42.4 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 *perldsc* and *perllol* for how to use references to create complex data structures, and *perltoot*, *perlobj*, and *perlbot* for how to use them to create objects.

Chapter 43

perlform

Perl formats

43.1 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 *perlfunc*. 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 adjustification
   |    pad character for centering
   >    pad character for right adjustificat
   #    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.

43.1.1 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 Using Fill Mode for details.

```
   Example:
      format STDOUT =
      @<<<<<<   @||||||   @>>>>>>
      "left",   "middle", "right"
      .
   Output:
      left      middle    right
```

43.1.2 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   ####
```

43.1.3 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.

43.1.4 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
```

43.1.5 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 **always** determined by the current LC_NUMERIC locale. This means that, if, for example, the run-time environment happens to specify a German locale, "," will be used instead of the default ".". See *perllocale* and §43.3 for more information.

43.1.6 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.

43.1.7 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.

43.1.8 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.

43.1.9 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 write in *perlfunc*.

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.

43.1.10 Format Variables

The current format name is stored in the variable `$˜` (`$FORMAT_NAME`), and the current top of form format name is in `$ˆ` (`$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 `$ˆL` (`$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 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 '-no_match_vars';
    $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!

43.2 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;
 }
```

43.2.1 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 open() in *perlfunc*) 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.

43.2.2 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;
```

43.3 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. (They weren’t visible at all before version 5.001.)

Formats are the only part of Perl that unconditionally use information from a program’s locale; if a program’s environment specifies an LC_NUMERIC locale, it is always used to specify the decimal point character in formatted output. Perl ignores all other aspects of locale handling unless the `use locale` pragma is in effect. Formatted output cannot be controlled by `use locale` because the pragma is tied to the block structure of the program, and, for historical reasons, formats exist outside that block structure. See *perllocale* 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.

Chapter 44

perlobj

Perl objects

44.1 DESCRIPTION

First you need to understand what references are in Perl. See *perlref* for that. Second, if you still find the following reference work too complicated, a tutorial on object-oriented programming in Perl can be found in *perltoot* and *perltooc*.

If you're still with us, then here are three very simple definitions that you should find reassuring.

1. An object is simply a reference that happens to know which class it belongs to.
2. A class is simply a package that happens to provide methods to deal with object references.
3. A method is simply a subroutine that expects an object reference (or a package name, for class methods) as the first argument.

We'll cover these points now in more depth.

44.1.1 An Object is Simply a Reference

Unlike say C++, Perl doesn't provide any special syntax for constructors. A constructor is merely a subroutine that returns a reference to something "blessed" into a class, generally the class that the subroutine is defined in. Here is a typical constructor:

```
    package Critter;
    sub new { bless {} }
```

That word `new` isn't special. You could have written a construct this way, too:

```
    package Critter;
    sub spawn { bless {} }
```

This might even be preferable, because the C++ programmers won't be tricked into thinking that `new` works in Perl as it does in C++. It doesn't. We recommend that you name your constructors whatever makes sense in the context of the problem you're solving. For example, constructors in the Tk extension to Perl are named after the widgets they create.

One thing that's different about Perl constructors compared with those in C++ is that in Perl, they have to allocate their own memory. (The other things is that they don't automatically call overridden base-class constructors.) The `{}` allocates an anonymous hash containing no key/value pairs, and returns it The bless() takes that reference and tells the object it references that it's now a Critter, and returns the reference. This is for convenience, because the referenced object itself knows that it has been blessed, and the reference to it could have been returned directly, like this:

```
    sub new {
        my $self = {};
        bless $self;
        return $self;
    }
```

You often see such a thing in more complicated constructors that wish to call methods in the class as part of the construction:

```
    sub new {
        my $self = {};
        bless $self;
        $self->initialize();
        return $self;
    }
```

If you care about inheritance (and you should; see Modules: Creation, Use, and Abuse in *perlmodlib*), then you want to use the two-arg form of bless so that your constructors may be inherited:

```
    sub new {
        my $class = shift;
        my $self = {};
        bless $self, $class;
        $self->initialize();
        return $self;
    }
```

Or if you expect people to call not just `CLASS->new()` but also `$obj->new()`, then use something like the following. (Note that using this to call new() on an instance does not automatically perform any copying. If you want a shallow or deep copy of an object, you'll have to specifically allow for that.) The initialize() method used will be of whatever $class we blessed the object into:

```
    sub new {
        my $this = shift;
        my $class = ref($this) || $this;
        my $self = {};
        bless $self, $class;
        $self->initialize();
        return $self;
    }
```

Within the class package, the methods will typically deal with the reference as an ordinary reference. Outside the class package, the reference is generally treated as an opaque value that may be accessed only through the class's methods.

Although a constructor can in theory re-bless a referenced object currently belonging to another class, this is almost certainly going to get you into trouble. The new class is responsible for all cleanup later. The previous blessing is forgotten, as an object may belong to only one class at a time. (Although of course it's free to inherit methods from many classes.) If you find yourself having to do this, the parent class is probably misbehaving, though.

A clarification: Perl objects are blessed. References are not. Objects know which package they belong to. References do not. The bless() function uses the reference to find the object. Consider the following example:

```
    $a = {};
    $b = $a;
    bless $a, BLAH;
    print "\$b is a ", ref($b), "\n";
```

This reports $b as being a BLAH, so obviously bless() operated on the object and not on the reference.

44.1.2 A Class is Simply a Package

Unlike say C++, Perl doesn't provide any special syntax for class definitions. You use a package as a class by putting method definitions into the class.

There is a special array within each package called @ISA, which says where else to look for a method if you can't find it in the current package. This is how Perl implements inheritance. Each element of the @ISA array is just the name of another package that happens to be a class package. The classes are searched (depth first) for missing methods in the order that they occur in @ISA. The classes accessible through @ISA are known as base classes of the current class.

All classes implicitly inherit from class `UNIVERSAL` as their last base class. Several commonly used methods are automatically supplied in the UNIVERSAL class; see §44.1.6 for more details.

If a missing method is found in a base class, it is cached in the current class for efficiency. Changing @ISA or defining new subroutines invalidates the cache and causes Perl to do the lookup again.

If neither the current class, its named base classes, nor the UNIVERSAL class contains the requested method, these three places are searched all over again, this time looking for a method named AUTOLOAD(). If an AUTOLOAD is found, this method is called on behalf of the missing method, setting the package global $AUTOLOAD to be the fully qualified name of the method that was intended to be called.

If none of that works, Perl finally gives up and complains.

If you want to stop the AUTOLOAD inheritance say simply

```
        sub AUTOLOAD;
```

and the call will die using the name of the sub being called.

Perl classes do method inheritance only. Data inheritance is left up to the class itself. By and large, this is not a problem in Perl, because most classes model the attributes of their object using an anonymous hash, which serves as its own little namespace to be carved up by the various classes that might want to do something with the object. The only problem with this is that you can't sure that you aren't using a piece of the hash that isn't already used. A reasonable workaround is to prepend your fieldname in the hash with the package name.

```
    sub bump {
        my $self = shift;
        $self->{ __PACKAGE__ . ".count"}++;
    }
```

44.1.3 A Method is Simply a Subroutine

Unlike say C++, Perl doesn't provide any special syntax for method definition. (It does provide a little syntax for method invocation though. More on that later.) A method expects its first argument to be the object (reference) or package (string) it is being invoked on. There are two ways of calling methods, which we'll call class methods and instance methods.

A class method expects a class name as the first argument. It provides functionality for the class as a whole, not for any individual object belonging to the class. Constructors are often class methods, but see *perltoot* and *perltooc* for alternatives. Many class methods simply ignore their first argument, because they already know what package they're in and don't care what package they were invoked via. (These aren't necessarily the same, because class methods follow the inheritance tree just like ordinary instance methods.) Another typical use for class methods is to look up an object by name:

```
    sub find {
        my ($class, $name) = @_;
        $objtable{$name};
    }
```

An instance method expects an object reference as its first argument. Typically it shifts the first argument into a "self" or "this" variable, and then uses that as an ordinary reference.

```
    sub display {
        my $self = shift;
        my @keys = @_ ? @_ : sort keys %$self;
        foreach $key (@keys) {
            print "\t$key => $self->{$key}\n";
        }
    }
```

44.1.4 Method Invocation

For various historical and other reasons, Perl offers two equivalent ways to write a method call. The simpler and more common way is to use the arrow notation:

```
    my $fred = Critter->find("Fred");
    $fred->display("Height", "Weight");
```

You should already be familiar with the use of the `->` operator with references. In fact, since `$fred` above is a reference to an object, you could think of the method call as just another form of dereferencing.
Whatever is on the left side of the arrow, whether a reference or a class name, is passed to the method subroutine as its first argument. So the above code is mostly equivalent to:

```
    my $fred = Critter::find("Critter", "Fred");
    Critter::display($fred, "Height", "Weight");
```

How does Perl know which package the subroutine is in? By looking at the left side of the arrow, which must be either a package name or a reference to an object, i.e. something that has been blessed to a package. Either way, that's the package where Perl starts looking. If that package has no subroutine with that name, Perl starts looking for it in any base classes of that package, and so on.
If you need to, you *can* force Perl to start looking in some other package:

```
    my $barney = MyCritter->Critter::find("Barney");
    $barney->Critter::display("Height", "Weight");
```

Here `MyCritter` is presumably a subclass of `Critter` that defines its own versions of find() and display(). We haven't specified what those methods do, but that doesn't matter above since we've forced Perl to start looking for the subroutines in `Critter`.
As a special case of the above, you may use the `SUPER` pseudo-class to tell Perl to start looking for the method in the packages named in the current class's `@ISA` list.

```
    package MyCritter;
    use base 'Critter';    # sets @MyCritter::ISA = ('Critter');

    sub display {
        my ($self, @args) = @_;
        $self->SUPER::display("Name", @args);
    }
```

It is important to note that `SUPER` refers to the superclass(es) of the *current package* and not to the superclass(es) of the object. Also, the `SUPER` pseudo-class can only currently be used as a modifier to a method name, but not in any of the other ways that class names are normally used, eg:

```
    something->SUPER::method(...);      # OK
    SUPER::method(...);                 # WRONG
    SUPER->method(...);                 # WRONG
```

Instead of a class name or an object reference, you can also use any expression that returns either of those on the left side of the arrow. So the following statement is valid:

```
    Critter->find("Fred")->display("Height", "Weight");
```

and so is the following:

```
    my $fred = (reverse "rettirC")->find(reverse "derF");
```

44.1.5 Indirect Object Syntax

The other way to invoke a method is by using the so-called "indirect object" notation. This syntax was available in Perl 4 long before objects were introduced, and is still used with filehandles like this:

```
print STDERR "help!!!\n";
```

The same syntax can be used to call either object or class methods.

```
my $fred = find Critter "Fred";
display $fred "Height", "Weight";
```

Notice that there is no comma between the object or class name and the parameters. This is how Perl can tell you want an indirect method call instead of an ordinary subroutine call.

But what if there are no arguments? In that case, Perl must guess what you want. Even worse, it must make that guess *at compile time*. Usually Perl gets it right, but when it doesn't you get a function call compiled as a method, or vice versa. This can introduce subtle bugs that are hard to detect.

For example, a call to a method `new` in indirect notation – as C++ programmers are wont to make – can be miscompiled into a subroutine call if there's already a `new` function in scope. You'd end up calling the current package's `new` as a subroutine, rather than the desired class's method. The compiler tries to cheat by remembering bareword `require`s, but the grief when it messes up just isn't worth the years of debugging it will take you to track down such subtle bugs.

There is another problem with this syntax: the indirect object is limited to a name, a scalar variable, or a block, because it would have to do too much lookahead otherwise, just like any other postfix dereference in the language. (These are the same quirky rules as are used for the filehandle slot in functions like `print` and `printf`.) This can lead to horribly confusing precedence problems, as in these next two lines:

```
move $obj->{FIELD};                 # probably wrong!
move $ary[$i];                      # probably wrong!
```

Those actually parse as the very surprising:

```
$obj->move->{FIELD};                # Well, lookee here
$ary->move([$i]);                   # Didn't expect this one, eh?
```

Rather than what you might have expected:

```
$obj->{FIELD}->move();              # You should be so lucky.
$ary[$i]->move;                     # Yeah, sure.
```

To get the correct behavior with indirect object syntax, you would have to use a block around the indirect object:

```
move {$obj->{FIELD}};
move {$ary[$i]};
```

Even then, you still have the same potential problem if there happens to be a function named `move` in the current package. **The `->` notation suffers from neither of these disturbing ambiguities, so we recommend you use it exclusively.** However, you may still end up having to read code using the indirect object notation, so it's important to be familiar with it.

44.1.6 Default UNIVERSAL methods

The `UNIVERSAL` package automatically contains the following methods that are inherited by all other classes:

isa(CLASS)

`isa` returns *true* if its object is blessed into a subclass of `CLASS`

You can also call `UNIVERSAL::isa` as a subroutine with two arguments. The first does not need to be an object or even a reference. This allows you to check what a reference points to, or whether something is a reference of a given type. Example

```
if(UNIVERSAL::isa($ref, 'ARRAY')) {
    #...
}
```

To determine if a reference is a blessed object, you can write

```
print "It's an object\n" if UNIVERSAL::isa($val, 'UNIVERSAL');
```

can(METHOD)

`can` checks to see if its object has a method called `METHOD`, if it does then a reference to the sub is returned, if it does not then *undef* is returned.

`UNIVERSAL::can` can also be called as a subroutine with two arguments. It'll always return *undef* if its first argument isn't an object or a class name. So here's another way to check if a reference is a blessed object

```
print "It's still an object\n" if UNIVERSAL::can($val, 'can');
```

You can also use the `blessed` function of Scalar::Util:

```
use Scalar::Util 'blessed';

my $blessing = blessed $suspected_object;
```

`blessed` returns the name of the package the argument has been blessed into, or `undef`.

VERSION([NEED)]

`VERSION` 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 given package) not less than NEED; it will die if this is not the case. This method is normally called as a class method. This method is called automatically by the `VERSION` form of `use`.

```
use A 1.2 qw(some imported subs);
# implies:
A->VERSION(1.2);
```

NOTE: `can` directly uses Perl's internal code for method lookup, and `isa` uses a very similar method and cache-ing strategy. This may cause strange effects if the Perl code dynamically changes @ISA in any package.

You may add other methods to the UNIVERSAL class via Perl or XS code. You do not need to `use UNIVERSAL` to make these methods available to your program (and you should not do so).

44.1.7 Destructors

When the last reference to an object goes away, the object is automatically destroyed. (This may even be after you exit, if you've stored references in global variables.) If you want to capture control just before the object is freed, you may define a DESTROY method in your class. It will automatically be called at the appropriate moment, and you can do any extra cleanup you need to do. Perl passes a reference to the object under destruction as the first (and only) argument. Beware that the reference is a read-only value, and cannot be modified by manipulating `$_[0]` within the destructor. The object itself (i.e. the thingy the reference points to, namely `${$_[0]}`, `@{$_[0]}`, `%{$_[0]}` etc.) is not similarly constrained.

If you arrange to re-bless the reference before the destructor returns, perl will again call the DESTROY method for the re-blessed object after the current one returns. This can be used for clean delegation of object destruction, or for ensuring that destructors in the base classes of your choosing get called. Explicitly calling DESTROY is also possible, but is usually never needed.

Do not confuse the previous discussion with how objects *CONTAINED* in the current one are destroyed. Such objects will be freed and destroyed automatically when the current object is freed, provided no other references to them exist elsewhere.

44.1.8 Summary

That's about all there is to it. Now you need just to go off and buy a book about object-oriented design methodology, and bang your forehead with it for the next six months or so.

44.1.9 Two-Phased Garbage Collection

For most purposes, Perl uses a fast and simple, reference-based garbage collection system. That means there's an extra dereference going on at some level, so if you haven't built your Perl executable using your C compiler's `-O` flag, performance will suffer. If you *have* built Perl with `cc -O`, then this probably won't matter.

A more serious concern is that unreachable memory with a non-zero reference count will not normally get freed. Therefore, this is a bad idea:

```
    {
        my $a;
        $a = \$a;
    }
```

Even thought $a *should* go away, it can't. When building recursive data structures, you'll have to break the self-reference yourself explicitly if you don't care to leak. For example, here's a self-referential node such as one might use in a sophisticated tree structure:

```
    sub new_node {
        my $class = shift;
        my $node  = {};
        $node->{LEFT} = $node->{RIGHT} = $node;
        $node->{DATA} = [ @_ ];
        return bless $node => $class;
    }
```

If you create nodes like that, they (currently) won't go away unless you break their self reference yourself. (In other words, this is not to be construed as a feature, and you shouldn't depend on it.)

Almost.

When an interpreter thread finally shuts down (usually when your program exits), then a rather costly but complete mark-and-sweep style of garbage collection is performed, and everything allocated by that thread gets destroyed. This is essential to support Perl as an embedded or a multithreadable language. For example, this program demonstrates Perl's two-phased garbage collection:

```
#!/usr/bin/perl
package Subtle;

sub new {
    my $test;
    $test = \$test;
    warn "CREATING " . \$test;
    return bless \$test;
}

sub DESTROY {
    my $self = shift;
    warn "DESTROYING $self";
}

package main;

warn "starting program";
{
    my $a = Subtle->new;
    my $b = Subtle->new;
    $$a = 0;  # break selfref
    warn "leaving block";
}

warn "just exited block";
warn "time to die...";
exit;
```

When run as */foo/test*, the following output is produced:

```
starting program at /foo/test line 18.
CREATING SCALAR(0x8e5b8) at /foo/test line 7.
CREATING SCALAR(0x8e57c) at /foo/test line 7.
leaving block at /foo/test line 23.
DESTROYING Subtle=SCALAR(0x8e5b8) at /foo/test line 13.
just exited block at /foo/test line 26.
time to die... at /foo/test line 27.
DESTROYING Subtle=SCALAR(0x8e57c) during global destruction.
```

Notice that "global destruction" bit there? That's the thread garbage collector reaching the unreachable.

Objects are always destructed, even when regular refs aren't. Objects are destructed in a separate pass before ordinary refs just to prevent object destructors from using refs that have been themselves destructed. Plain refs are only garbage-collected if the destruct level is greater than 0. You can test the higher levels of global destruction by setting the PERL_DESTRUCT_LEVEL environment variable, presuming `-DDEBUGGING` was enabled during perl build time. See PERL_DESTRUCT_LEVEL in *perlhack* for more information.

A more complete garbage collection strategy will be implemented at a future date.

In the meantime, the best solution is to create a non-recursive container class that holds a pointer to the self-referential data structure. Define a DESTROY method for the containing object's class that manually breaks the circularities in the self-referential structure.

44.2 SEE ALSO

A kinder, gentler tutorial on object-oriented programming in Perl can be found in *perltoot*, *perlboot* and *perltooc*. You should also check out *perlbot* for other object tricks, traps, and tips, as well as *perlmodlib* for some style guides on constructing both modules and classes.

Chapter 45

perltie

How to hide an object class in a simple variable

45.1 SYNOPSIS

```
 tie VARIABLE, CLASSNAME, LIST

 $object = tied VARIABLE

 untie VARIABLE
```

45.2 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.

45.2.1 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 The `untie` Gotcha 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.

45.2.2 Tying Arrays

A class implementing a tied ordinary array should define the following methods: TIEARRAY, FETCH, STORE, FETCHSIZE, STORESIZE 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 The `untie` Gotcha 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.

45.2.3 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 $user =~ /^\d+$/;
    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 $dot ( grep /^\./ && -f "$dir/$_", readdir(DIR)) {
        $dot =~ s/^\.//;
        $node->{LIST}{$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(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 The `untie` Gotcha 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);
```

45.2.4 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 %SIG in *perlvar* 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.

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()` function. Beyond its self reference it also expects the list that was passed to the print function.

```
    sub PRINT { $r = shift; $$r++; print join($,,map(uc($_),@_)),$\ }
```

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 will be called when the handle is read from via <HANDLE>. The method should return undef when there is no more data.

```
        sub READLINE { $r = shift; "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"; }
```

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 The `untie` Gotcha 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>;
```

45.2.5 UNTIE this

You can define for all tie types an UNTIE method that will be called at untie(). See The `untie` Gotcha below.

45.2.6 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 = new IO::File "> $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;
    }
```

45.3 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*.

45.4 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 experimental module that does attempt to address this need partially is the MLDBM module. Check your nearest CPAN site as described in *perlmodlib* for source code to MLDBM.

Tied filehandles are still incomplete. sysopen(), truncate(), flock(), fcntl(), stat() and -X can't currently be trapped.

45.5 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*>

Chapter 46

perldbmfilter

Perl DBM Filters

46.1 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 { ... } ) ;
```

46.2 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 work identically, and are 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` in not.

To delete a filter pass `undef` to it.

46.2.1 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.

46.2.2 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.

46.2.3 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.

46.3 SEE ALSO

DB_File, GDBM_File, NDBM_File, ODBM_File and SDBM_File.

46.4 AUTHOR

Paul Marquess

Chapter 47

perlipc

Perl interprocess communication (signals, fifos, pipes, safe subprocesses, sockets, and semaphores)

47.1 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.

47.2 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 process running out of stack space, or hitting file size limit.

For example, to trap an interrupt signal, set up a handler like this:

```
sub catch_zap {
    my $signame = shift;
    $shucks++;
    die "Somebody sent me a SIG$signame";
}
$SIG{INT} = 'catch_zap';  # could fail in modules
$SIG{INT} = \&catch_zap;  # best strategy
```

Prior to Perl 5.7.3 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 Deferred Signals (Safe Signals) below.

The names of the signals are the ones listed out by `kill -l` on your system, or you can retrieve them from the Config module. Set up an @signame list indexed by number to get the name and a %signo table indexed by name to get the number:

```
use Config;
defined $Config{sig_name} || die "No sigs?";
foreach $name (split(' ', $Config{sig_name})) {
    $signo{$name} = $i;
    $signame[$i] = $name;
    $i++;
}
```

So to check whether signal 17 and SIGALRM were the same, do just this:

```
    print "signal #17 = $signame[17]\n";
    if ($signo{ALRM}) {
        print "SIGALRM is $signo{ALRM}\n";
    }
```

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. One strategy for temporarily ignoring signals is to use a local() statement, which will be automatically restored once your block is exited. (Remember that local() values are "inherited" by functions called from within that block.)

```
    sub precious {
        local $SIG{INT} = 'IGNORE';
        &more_functions;
    }
    sub more_functions {
        # interrupts still ignored, for now...
    }
```

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 sets $SIG{HUP} to IGNORE so it doesn't kill itself):

```
    {
        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 UID.

```
    unless (kill 0 => $kid_pid) {
        warn "something wicked happened to $kid_pid";
    }
```

When directed at a process whose UID is not identical to that of the sending process, signal number zero may fail because you lack permission to send the signal, even though the process is alive. You may be able to determine the cause of failure using `%!`.

```
    unless (kill 0 => $pid or $!{EPERM}) {
        warn "$pid looks dead";
    }
```

You might also want to employ anonymous functions for simple signal handlers:

```
    $SIG{INT} = sub { die "\nOutta here!\n" };
```

But that will be problematic for the more complicated handlers that need to reinstall themselves. Because Perl's signal mechanism is currently based on the signal(3) function from the C library, you may sometimes be so misfortunate as to run on systems where that function is "broken", that is, it behaves in the old unreliable SysV way rather than the newer, more reasonable BSD and POSIX fashion. So you'll see defensive people writing signal handlers like this:

```
sub REAPER {
    $waitedpid = wait;
    # loathe sysV: it makes us not only reinstate
    # the handler, but place it after the wait
    $SIG{CHLD} = \&REAPER;
}
$SIG{CHLD} = \&REAPER;
# now do something that forks...
```

or better still:

```
use POSIX ":sys_wait_h";
sub REAPER {
    my $child;
    # If a second child dies while in the signal handler caused by the
    # first death, we won't get another signal. So must loop here else
    # we will leave the unreaped child as a zombie. And the next time
    # two children die we get another zombie. And so on.
    while (($child = waitpid(-1,WNOHANG)) > 0) {
        $Kid_Status{$child} = $?;
    }
    $SIG{CHLD} = \&REAPER;  # still loathe sysV
}
$SIG{CHLD} = \&REAPER;
# do something that forks...
```

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, much as you might using longjmp() or throw() in other languages.

Here's an example:

```
eval {
    local $SIG{ALRM} = sub { die "alarm clock restart" };
    alarm 10;
    flock(FH, 2);   # blocking write lock
    alarm 0;
};
if ($@ and $@ !~ /alarm clock restart/) { 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.

47.2.1 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 re-read its configuration file, without stopping the process.

Many daemons provide this mechanism using the `SIGHUP` signal handler. When you want to tell the daemon to re-read the file you simply send it the `SIGHUP` signal.

Not all platforms automatically reinstall their (native) signal handlers after a signal delivery. This means that the handler works only the first time the signal is sent. The solution to this problem is to use `POSIX` signal handlers if available, their behaviour is well-defined.

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 simply prints some debug info to show that it works and should be replaced with the real code.

```
#!/usr/bin/perl -w

use POSIX ();
use FindBin ();
use File::Basename ();
use File::Spec::Functions;

$|=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
my $sigset = POSIX::SigSet->new();
my $action = POSIX::SigAction->new('sigHUP_handler',
                                   $sigset,
                                   &POSIX::SA_NODEFER);
POSIX::sigaction(&POSIX::SIGHUP, $action);

sub sigHUP_handler {
    print "got SIGHUP\n";
    exec($SELF, @ARGV) or die "Couldn't restart: $!\n";
}

code();

sub code {
    print "PID: $$\n";
    print "ARGV: @ARGV\n";
    my $c = 0;
    while (++$c) {
        sleep 2;
        print "$c\n";
    }
}
__END__
```

47.3 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 a regular, connected anonymous pipes, except that the processes rendezvous using a filename and don't have to be related.

To create a named pipe, use the Unix command mknod(1) or on some systems, mkfifo(1). These may not be in your normal path.

```
    # 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 block and your program will supply the new signature. We'll use the pipe-checking file test **-p** to find out whether anyone (or anything) has accidentally removed our fifo.

```
    chdir; # go home
    $FIFO = '.signature';
    $ENV{PATH} .= ":/etc:/usr/games";

    while (1) {
        unless (-p $FIFO) {
            unlink $FIFO;
            system('mknod', $FIFO, 'p')
                && die "can't mknod $FIFO: $!";
        }

        # next line blocks until there's a reader
        open (FIFO, "> $FIFO") || die "can't write $FIFO: $!";
        print FIFO "John Smith (smith\@host.org)\n", `fortune -s`;
        close FIFO;
        sleep 2;    # to avoid dup signals
    }
```

Deferred Signals (Safe Signals)

In Perls before Perl 5.7.3 by installing Perl code to deal with signals, you were exposing yourself 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 behaviour 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 longjump(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.

In Perl 5.7.3 and later to avoid these problems signals are "deferred"– 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 handler as we know 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 Perl interpreter only looks at the signal flags when it about to execute a new opcode if a signal arrives during a long running opcode (e.g. a regular expression operation on a very large string) then signal will not be seen until operation completes.

Interrupting IO

When a signal is delivered (e.g. INT control-C) the operating system breaks into IO operations like `read` (used to implement Perls <> 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 system's `stdio` library that library may re-start the `read` without returning to Perl and giving it a chance to call the %SIG handler. If this happens on your system the solution is to use `:perlio` layer to do IO - at least on those handles which 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.)

Note that the default in Perl 5.7.3 and later is to automatically use the `:perlio` layer.

Note that some networking library functions like gethostbyname() are known to have their own implementations of timeouts which may conflict with your timeouts. If you are having problems with such functions, you can try using the POSIX sigaction() function, which bypasses the Perl safe signals (note that this means subjecting yourself to possible memory corruption, as described above). Instead of setting `$SIG{ALRM}` try something like the following:

```
use POSIX;
sigaction SIGALRM, new POSIX::SigAction sub { die "alarm\n" }
    or die "Error setting SIGALRM handler: $!\n";
```

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 5.7.3 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.

Note that the default `:perlio` layer will retry `read`, `write` and `close` as described above and that interrupted `wait` and `waitpid` calls will always be retried.

Signals as "faults"

Certain signals e.g. SEGV, ILL, BUS are generated as a result of virtual memory or other "faults". These are normally fatal and there is little a Perl-level handler can do with them. (In particular the old signal scheme was particularly unsafe in such cases.) However if a %SIG handler is set the new scheme simply sets a flag and returns as described above. This may cause the operating system to try the offending machine instruction again and - as nothing has changed - it will generate the signal again. The result of this is a rather odd "loop". In future Perl's signal mechanism may be changed to avoid this - perhaps by simply disallowing %SIG handlers on signals of that type. Until then the work-round is not to set a %SIG handler on those signals. (Which signals they are is operating system dependant.)

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 re-issue the signal as there are un-waited-for completed child processes.

If you want the old signal behaviour back regardless of possible memory corruption, set the environment variable `PERL_SIGNALS` to `"unsafe"` (a new feature since Perl 5.8.1).

47.3.1 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 that is expecting filenames in @ARGV, the clever programmer can write something like this:

```
% program f1 "cmd1|" - f2 "cmd2|" f3 < tmpfile
```

and irrespective of which 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 netstat" 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 to kill off the child process early if you'd like.

Be careful to check both the open() and the close() return values. 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 end of file, writers to bogus command will trigger a signal they'd better be prepared to handle. Consider:

```
open(FH, "|bogus")  or die "can't fork: $!";
print FH "bang\n"   or die "can't write: $!";
close FH            or die "can't close: $!";
```

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")  or die "can't fork: $!";
print FH "bang\n"   or die "can't write: $!";
close FH            or die "can't close: status=$?";
```

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.

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 more details).

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 '/'               or die "Can't chdir to /: $!";
        open STDIN, '/dev/null' or die "Can't read /dev/null: $!";
        open STDOUT, '>/dev/null'
                                or die "Can't write to /dev/null: $!";
        defined(my $pid = fork) or die "Can't fork: $!";
        exit if $pid;
        setsid                  or die "Can't start a new session: $!";
        open STDERR, '>&STDOUT' or die "Can't dup stdout: $!";
    }
```

The fork() has to come before the setsid() to ensure that 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 solutions.

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 '-no_match_vars';
    my $sleep_count = 0;
    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) {  # parent
        print KID_TO_WRITE @some_data;
        close(KID_TO_WRITE) || warn "kid exited $?";
    } else {     # child
        ($EUID, $EGID) = ($UID, $GID); # suid progs only
        open (FILE, "> /safe/file")
            || die "can't open /safe/file: $!";
        while (<STDIN>) {
            print FILE; # child's STDIN is parent's KID
        }
        exit;  # 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:

```
    # add error processing as above
    $pid = open(KID_TO_READ, "-|");

    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:

```
    # add error processing as above
    $pid = open(KID_TO_WRITE, "|-");
    $SIG{PIPE} = sub { die "whoops, $program pipe broke" };

    if ($pid) {  # parent
        for (@data) {
            print KID_TO_WRITE;
        }
        close(KID_TO_WRITE) || warn "kid exited $?";
    } else {     # child
        ($EUID, $EGID) = ($UID, $GID);
        exec($program, @options, @args)
            || die "can't exec program: $!";
        # NOTREACHED
    }
```

Since Perl 5.8.0, you can also use the list form of `open` for pipes : the syntax

```
    open KID_PS, "-|", "ps", "aux" or die $!;
```

forks the ps(1) command (without spawning a shell, as there are more than three arguments to open()), and reads its standard output via the `KID_PS` filehandle. The corresponding syntax to read from command pipes (with `"|-"` in place of `"-|"`) is also implemented.

Note that these operations are full Unix forks, which means they may not be correctly implemented on alien systems. Additionally, these are not true multithreading. If you'd like to learn more about threading, see the *modules* file mentioned below in the SEE ALSO section.

Bidirectional Communication with Another Process

While this works reasonably well for unidirectional communication, what about bidirectional communication? The obvious thing you'd like to do doesn't actually work:

```
    open(PROG_FOR_READING_AND_WRITING, "| some program |")
```

and if you forget to use the `use warnings` pragma or the **-w** flag, then you'll miss out entirely on the diagnostic message:

```
    Can't do bidirectional pipe at -e line 1.
```

If you really want to, you can use the standard open2() library function to catch both ends. There's also an 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 Unix pipe() and exec() calls to create all the connections. While it might have been slightly more efficient by using socketpair(), it would have then been even less portable than it already is. The open2() and open3() functions are unlikely to work anywhere except on a Unix system or some other one purporting to be POSIX compliant.

Here's an example of using open2():

```
    use FileHandle;
    use IPC::Open2;
    $pid = open2(*Reader, *Writer, "cat -u -n" );
    print Writer "stuff\n";
    $got = <Reader>;
```

The problem with this is that Unix buffering is really going to ruin your day. Even though your `Writer` filehandle is auto-flushed, and the process on the other end will get your data in a timely manner, you can't usually do anything to force it to give it back to you in a similarly quick fashion. In this case, we could, because we gave *cat* a **-u** flag to make it unbuffered. But very few Unix 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 the nonstandard *Comm.pl* library. It uses pseudo-ttys to make your program behave more reasonably:

```
require 'Comm.pl';
$ph = open_proc('cat -n');
for (1..10) {
    print $ph "a line\n";
    print "got back ", scalar <$ph>;
}
```

This way you don't have to have control over the source code of the program you're using. The *Comm* library also has expect() and interact() functions. Find the library (and we hope its successor *IPC::Chat*) at your nearest CPAN archive as detailed in the SEE ALSO section below.

The newer Expect.pm 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 using talking to the terminal device driver. If your system is amongst those supported, this may be your best bet.

Bidirectional Communication with Yourself

If you want, you may make low-level pipe() and fork() 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.

```
#!/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: failure?
pipe(CHILD_RDR,  PARENT_WTR);              # XXX: 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;
}
```

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)
                            or  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;
    }
```

47.3.2 Sockets: Client/Server Communication

While not limited to Unix-derived operating systems (e.g., WinSock on PCs provides socket support, as do some VMS libraries), you may 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 (i.e., TCP streams) and datagrams (i.e., UDP packets). You may be able to do even more depending on your system.

The Perl function calls 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 old 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 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).

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 (but 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, you'll probably be ok.

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: $!";
while (defined($line = <SOCK>)) {
    print $line;
}

close (SOCK)            || die "close: $!";
exit;
```

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), you should fill this in with your real address instead.

```
#!/usr/bin/perl -Tw
use strict;
BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
use Socket;
use Carp;
my $EOL = "\015\012";

sub logmsg { print "$0 $$: @_ at ", scalar localtime, "\n" }

my $port = shift || 2345;
my $proto = getprotobyname('tcp');

($port) = $port =~ /^(\d+)$/                        or die "invalid port";

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 multithreaded version. It's multithreaded in that like most typical servers, it spawns (forks) 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/ucb:/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;
    my $proto = getprotobyname('tcp');

    ($port) = $port =~ /^(\d+)$/                        or die "invalid port";

    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";
    sub REAPER {
        my $child;
        while (($waitedpid = waitpid(-1,WNOHANG)) > 0) {
            logmsg "reaped $waitedpid" . ($? ? " with exit $?" : '');
        }
        $SIG{CHLD} = \&REAPER;  # loathe sysV
    }

    $SIG{CHLD} = \&REAPER;

    for ( $waitedpid = 0;
          ($paddr = accept(Client,Server)) || $waitedpid;
          $waitedpid = 0, close Client)
    {
        next if $waitedpid and not $paddr;
        my($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: $!";
        };

    }

    sub spawn {
        my $coderef = shift;

        unless (@_ == 0 && $coderef && ref($coderef) eq 'CODE') {
            confess "usage: spawn CODEREF";
        }

        my $pid;
        if (!defined($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.

We suggest that you use the **-T** flag to use taint checking (see *perlsec*) even if we aren't running setuid or setgid. This is always a good idea for servers and other programs 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) }
    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(time());

    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 "bind: $!";
        my $rtime = '    ';
        read(SOCKET, $rtime, 4);
        close(SOCKET);
        my $histime = unpack("N", $rtime) - $SECS_of_70_YEARS ;
        printf "%8d %s\n", $histime - time, ctime($histime);
    }
```

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;
```

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/ucb:/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' or die "can't exec fortune: $!";
    };
}

sub spawn {
    my $coderef = shift;

    unless (@_ == 0 && $coderef && ref($coderef) eq 'CODE') {
        confess "usage: spawn CODEREF";
    }
```

```
        my $pid;
        if (!defined($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 exactly 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 from the World Wide Web to be able to access, 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.

47.3.3 TCP Clients with IO::Socket

For those preferring a higher-level interface to socket programming, the IO::Socket module provides an object-oriented approach. IO::Socket is included as part of the standard Perl distribution as of the 5.004 release. If you're running an earlier version of Perl, 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–just to name a few.

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;
    $remote = IO::Socket::INET->new(
                        Proto    => "tcp",
                        PeerAddr => "localhost",
                        PeerPort => "daytime(13)",
                    )
                  or die "cannot connect to daytime port at localhost";
    while ( <$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 `"204.148.40.9"`. 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 `"127.1"`, 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 in */etc/services* under Unix] but just in case, we've specified the port number (13) in parentheses. Using just the number would also have worked, but constant numbers 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";
```

A Webget Client

Here's a simple client that takes a remote host to fetch a document from, and then a list of documents 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 document ..." }
    $host = shift(@ARGV);
    $EOL = "\015\012";
    $BLANK = $EOL x 2;
    foreach $document ( @ARGV ) {
        $remote = IO::Socket::INET->new( Proto     => "tcp",
                                         PeerAddr  => $host,
                                         PeerPort  => "http(80)",
                                       );
        unless ($remote) { die "cannot connect to http daemon on $host" }
        $remote->autoflush(1);
        print $remote "GET $document HTTP/1.0" . $BLANK;
        while ( <$remote> ) { print }
        close $remote;
    }
```

The web server handing the "http" service, which is assumed to be at its standard port, number 80. If the web server you're trying to connect to is at a different port (like 1080 or 8080), you should specify 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 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 fully-featured version of this program, you should look to the *lwp-request* program included with the LWP modules from CPAN.

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)
           or 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;
        }
    }
```

The `kill` function in the parent's `if` block is there to send a signal to our child process (current 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.

47.3.4 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. Note that unlike most network servers, this one only handles one incoming client at a time. Multithreaded servers are covered in Chapter 6 of the Camel.

Here's the code. We'll

```
#!/usr/bin/perl -w
use IO::Socket;
use Net::hostent;              # for OO 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;
}
```

47.3.5 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.

Note that 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      = 2208988800;

$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 time;
$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--;
}
```

Note that 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 list of hosts to contact is sufficiently large.

47.3.6 SysV IPC

While System V IPC isn't so widely used as sockets, it still has some interesting uses. You can't, however, effectively use SysV IPC or Berkeley mmap() to have shared memory so as to share a variable amongst several processes. That's because Perl would reallocate your string when you weren't wanting it to.

Here's a small example showing shared memory usage.

```
use IPC::SysV qw(IPC_PRIVATE IPC_RMID S_IRWXU);

$size = 2000;
$id = shmget(IPC_PRIVATE, $size, S_IRWXU) || die "$!";
print "shm key $id\n";

$message = "Message #1";
shmwrite($id, $message, 0, 60) || die "$!";
print "wrote: '$message'\n";
shmread($id, $buff, 0, 60) || die "$!";
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 "$!";
```

Here's an example of a semaphore:

```
use IPC::SysV qw(IPC_CREAT);

$IPC_KEY = 1234;
$id = semget($IPC_KEY, 10, 0666 | IPC_CREAT ) || die "$!";
print "shm key $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 );
die if !defined($id);

$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 "$!";
```

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 if !defined($id);

    $semnum = 0;
    $semflag = 0;

    # Decrement the semaphore count
    $semop = -1;
    $opstring = pack("s!s!s!", $semnum, $semop, $semflag);

    semop($id,$opstring) || die "$!";
```

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 which is included with Perl starting from Perl 5.005.

A small example demonstrating SysV message queues:

```
    use IPC::SysV qw(IPC_PRIVATE IPC_RMID IPC_CREAT S_IRWXU);

    my $id = msgget(IPC_PRIVATE, IPC_CREAT | S_IRWXU);

    my $sent = "message";
    my $type_sent = 1234;
    my $rcvd;
    my $type_rcvd;

    if (defined $id) {
        if (msgsnd($id, pack("l! a*", $type_sent, $sent), 0)) {
            if (msgrcv($id, $rcvd, 60, 0, 0)) {
                ($type_rcvd, $rcvd) = unpack("l! a*", $rcvd);
                if ($rcvd eq $sent) {
                    print "okay\n";
                } else {
                    print "not okay\n";
                }
            } else {
                die "# msgrcv failed\n";
            }
        } else {
            die "# msgsnd failed\n";
        }
        msgctl($id, IPC_RMID, 0) || die "# msgctl failed: $!\n";
    } else {
        die "# msgget failed\n";
    }
```

47.3.7 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 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 **-T** taint checking flag to the #! line for servers:

```
#!/usr/bin/perl -Tw
use strict;
use sigtrap;
use Socket;
```

47.3.8 BUGS

All these routines create system-specific portability problems. As noted elsewhere, Perl is at the mercy of your C libraries for much of its system behaviour. 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.

47.3.9 AUTHOR

Tom Christiansen, with occasional vestiges of Larry Wall's original version and suggestions from the Perl Porters.

47.3.10 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). Note that 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 *perlfunc*, you should also check out the *modules* file at your nearest CPAN site. (See *perlmodlib* or best yet, the *Perl FAQ* for a description of what CPAN is and where to get it.)

Section 5 of the *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–just to name a few.

Chapter 48

perlfork

Perl's fork() emulation

48.1 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.

48.2 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.

48.2.1 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() can be used to terminate a pseudo-process by passing it the ID returned by fork(). This 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. Note that using 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.

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().

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.

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.

48.2.2 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.

48.2.3 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.

48.2.4 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.

A way to mark a pseudo-processes as running detached from their parent (so that the parent would not have to wait() for them if it doesn't want to) will be provided in future.

48.2.5 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.

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.

48.3 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.

48.4 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>.

48.5 SEE ALSO

fork in *perlfunc*, *perlipc*

Chapter 49

perlnumber

Semantics of numbers and numeric operations in Perl

49.1 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
```

49.2 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.

49.3 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 §49.4 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.

49.4 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.

49.5 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 *perlop* 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.

49.6 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>

49.7 SEE ALSO

overload, *perlop*

Chapter 50

perlthrtut

Tutorial on threads in Perl

50.1 DESCRIPTION

NOTE: this tutorial describes the new Perl threading flavour introduced in Perl 5.6.0 called interpreter threads, or **ithreads** for short. In this model each thread runs in its own Perl interpreter, and any data sharing between threads must be explicit.

There is another older Perl threading flavour called the 5.005 model, unsurprisingly for 5.005 versions of Perl. The old model is known to have problems, deprecated, and will probably be removed around 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 user-level interface to the 5.005 threads was via the *Threads* class, while ithreads uses the *threads* class. Note the change in case.

50.2 Status

The ithreads code has been available since Perl 5.6.0, and is considered stable. The user-level interface to ithreads (the *threads* classes) appeared in the 5.8.0 release, and as of this time is considered stable although it should be treated with caution as with all new features.

50.3 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.

50.4 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.

50.4.1 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.

50.4.2 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.

50.4.3 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.)

50.5 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 systems threads allows it. So if your system blocks the entire process on sleep(), Perl usually will as well.

Perl Threads Are Different.

50.6 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 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 §50.9.5.

See also §50.15.

50.7 Thread Basics

The core *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.

50.7.1 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:

```
    $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.

50.7.2 A Note about the Examples

Although thread support is considered to be stable, there are still a number of quirks that may startle you when you try out any of the examples below. 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. Future versions of Perl may fix this problem.

50.7.3 Creating Threads

The *threads* package 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 new():

```
    use threads;

    $thr = threads->new(\&sub1);

    sub sub1 {
        print "In the thread\n";
    }
```

The new() method takes a reference to a subroutine and creates a new thread, which 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::new` call, like this:

```
    use threads;

    $Param3 = "foo";
    $thr = threads->new(\&sub1, "Param 1", "Param 2", $Param3);
    $thr = threads->new(\&sub1, @ParamList);
    $thr = threads->new(\&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.

`create()` is a synonym for `new()`.

50.7.4 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;

    $thr = threads->new(\&sub1);
    @ReturnData = $thr->join;
    print "Thread returned @ReturnData";

    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.

50.7.5 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;

    $thr = threads->new(\&sub1); # Spawn the thread

    $thr->detach; # Now we officially don't care any more

    sub sub1 {
        $a = 0;
        while (1) {
            $a++;
            print "\$a is $a\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.

50.8 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.

50.8.1 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->new(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.

50.8.2 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 $a : shared = 1;
    $thr1 = threads->new(\&sub1);
    $thr2 = threads->new(\&sub2);

    $thr1->join;
    $thr2->join;
    print "$a\n";

    sub sub1 { my $foo = $a; $a = $foo + 1; }
    sub sub2 { my $bar = $a; $a = $bar + 1; }
```

What do you think $a will be? The answer, unfortunately, is "it depends." Both sub1() and sub2() access the global variable $a, once to read and once to write. Depending on factors ranging from your thread implementation's scheduling algorithm to the phase of the moon, $a 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 $a : shared = 2;
my $b : shared;
my $c : shared;
my $thr1 = threads->create(sub { $b = $a; $a = $b + 1; });
my $thr2 = threads->create(sub { $c = $a; $a = $c + 1; });
$thr1->join;
$thr2->join;
```

Two threads both access $a. Each thread can potentially be interrupted at any point, or be executed in any order. At the end, $a could be 3 or 4, and both $b and $c could be 2 or 3.

Even `$a += 5` or `$a++` 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.

50.9 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.

50.9.1 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 outermost block that contains `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 {
    for (;;) {
        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->new(\&calc);
    my $thr2 = threads->new(\&calc);
    my $thr3 = threads->new(\&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 outermost 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.

50.9.2 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 $a : shared = 4;
    my $b : shared = "foo";
    my $thr1 = threads->new(sub {
        lock($a);
        sleep 20;
        lock($b);
    });
    my $thr2 = threads->new(sub {
        lock($b);
        sleep 20;
        lock($a);
    });
```

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 $a, then, after a pause during which the second thread has probably had time to do some work, try to grab a lock on $b. Meanwhile, the second thread grabs a lock on $b, then later tries to grab a lock on $a. 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 $a, $b, and $c, always lock $a before $b, and $b before $c. 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.

50.9.3 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;
    $thr = threads->new(sub {
        while ($DataElement = $DataQueue->dequeue) {
            print "Popped $DataElement off the queue\n";
        }
    });

    $DataQueue->enqueue(12);
    $DataQueue->enqueue("A", "B", "C");
    $DataQueue->enqueue(\$thr);
    sleep 10;
    $DataQueue->enqueue(undef);
    $thr->join;
```

You create the queue with `new Thread::Queue`. 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.

50.9.4 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 thay 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.

50.9.5 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 = new Thread::Semaphore;
my $GlobalVariable : shared = 0;

$thr1 = new threads \&sample_sub, 1;
$thr2 = new threads \&sample_sub, 2;
$thr3 = new threads \&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.

50.9.6 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

    $thr1 = threads->new(\&sub1);
    $thr2 = threads->new(\&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/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' 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.

50.9.7 cond_wait() and cond_signal()

These two functions 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.

50.9.8 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 $thread1 = threads->new(\&loop, 'first');
my $thread2 = threads->new(\&loop, 'second');
my $thread3 = threads->new(\&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.

50.10 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.

50.10.1 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.

50.10.2 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.

50.10.3 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.

50.10.4 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:

```
# Loop through all the threads
foreach $thr (threads->list) {
    # Don't join the main thread or ourselves
    if ($thr->tid && !threads::equal($thr, threads->self)) {
        $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

50.11 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.

```
 1 #!/usr/bin/perl -w
 2 # prime-pthread, courtesy of Tom Christiansen
 3
 4 use strict;
 5
 6 use threads;
 7 use Thread::Queue;
 8
 9 my $stream = new Thread::Queue;
10 my $kid    = new threads(\&check_num, $stream, 2);
11
12 for my $i ( 3 .. 1000 ) {
13     $stream->enqueue($i);
14 }
15
16 $stream->enqueue(undef);
17 $kid->join;
18
19 sub check_num {
20     my ($upstream, $cur_prime) = @_;
21     my $kid;
22     my $downstream = new Thread::Queue;
23     while (my $num = $upstream->dequeue) {
24         next unless $num % $cur_prime;
25         if ($kid) {
26            $downstream->enqueue($num);
27                   } else {
28            print "Found prime $num\n";
29                $kid = new threads(\&check_num, $downstream, $num);
30         }
31     }
32     $downstream->enqueue(undef) if $kid;
33     $kid->join           if $kid;
34 }
```

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's checking (line 20), we create a new queue (line 22) and reserve a scalar for the thread that we're likely to create later (line 21).

The while loop from lines 23 to line 31 grabs a scalar off the input queue and checks against the prime this thread is responsible for. Line 24 checks to see if there's a remainder when we modulo the number to be checked against 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 26) or create a new thread if we haven't.

The new thread creation is line 29. We pass on to it a reference to the queue we've created, and the prime number we've found.

Finally, once the loop terminates (because we got a 0 or undef in the queue, which serves as a note to die), we pass on the notice to our child and wait for it to exit if we've created a child (lines 32 and 37).

Meanwhile, back in the main thread, we create a queue (line 9) and the initial child thread (line 10), and pre-seed it with the first prime: 2. Then we queue all the numbers from 3 to 1000 for checking (lines 12-14), then queue a die notice (line 16) and wait for the first child thread to terminate (line 17). Because a child won't die until its child has died, we know that we're done once we return from the join.

That's how it works. It's pretty simple; as with many Perl programs, the explanation is much longer than the program.

50.12 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 `$a = $a + 2` can behave unpredictably with kernel threads if $a is visible to other threads, as another thread may have changed $a 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.

50.13 Performance considerations

The main thing to bear in mind when comparing 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.

50.14 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/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 should not be attempted. 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).

50.15 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(), get{gr,host,net,proto,serv,pw}*(), 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.

50.16 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.

50.17 Bibliography

Here’s a short bibliography courtesy of Jürgen Christoffel:

50.17.1 Introductory Texts

Birrell, Andrew D. An Introduction to Programming with Threads. Digital Equipment Corporation, 1989, DEC-SRC Research Report #35 online as http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-035.html (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).

50.17.2 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

50.17.3 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

50.18 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.

50.19 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().

50.20 Copyrights

For more information please see *threads* and *threads::shared*.

Chapter 51

perlothrtut

Old tutorial on threads in Perl

51.1 DESCRIPTION

WARNING: This tutorial describes the old-style thread model that was introduced in release 5.005. This model is now deprecated, and will be removed, probably in version 5.10. The interfaces described here were considered experimental, and are likely to be buggy.

For information about the new interpreter threads ("ithreads") model, see the *perlthrtut* tutorial, and the *threads* and *threads::shared* modules.

You are strongly encouraged to migrate any existing threads code to the new model as soon as possible.

51.2 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 5.005, though, you can create extra threads. We're going to show you how, when, and why.

51.3 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.

51.3.1 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.

51.3.2 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.

51.3.3 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.)

51.4 Native threads

There are several different ways to implement threads on a system. How threads are implemented depends both on the vendor and, in some cases, the version of the operating system. Often the first implementation will be relatively simple, but later versions of the OS will be more sophisticated.

While the information in this section is useful, it's not necessary, so you can skip it if you don't feel up to it.

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 `$a = $a + 2` can behave unpredictably with kernel threads if $a is visible to other threads, as another thread may have changed $a 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.)

51.5 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.

51.6 Threadsafe Modules

The addition of threads has changed Perl's internals substantially. There are implications for people who write modules–especially modules with XS code or external libraries. While most modules won't encounter any problems, modules that aren't explicitly tagged as thread-safe should be tested 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 beta feature, and even some of the standard modules aren't thread-safe.

If you're using a module that's not thread-safe for some reason, you can protect yourself by using semaphores and lots of programming discipline to control access to the module. Semaphores are covered later in the article. Perl Threads Are Different

51.7 Thread Basics

The core Thread 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 Thread module that make threaded programming easier.

51.7.1 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.

Remember that the threading support in 5.005 is in beta release, and should be treated as such. You should expect that it may not function entirely properly, and the thread interface may well change some before it is a fully supported, production release. The beta version shouldn't be used for mission-critical projects. Having said that, threaded Perl is pretty nifty, and worth a look.

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:

```
  $Config{usethreads} 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;

    if ($Config{usethreads}) {
        # 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.

51.7.2 Creating Threads

The Thread package provides the tools you need to create new threads. Like any other module, you need to tell Perl you want to use it; use Thread imports all the pieces you need to create basic threads.

The simplest, straightforward way to create a thread is with new():

```
    use Thread;

    $thr = new Thread \&sub1;

    sub sub1 {
        print "In the thread\n";
    }
```

The new() method takes a reference to a subroutine and creates a new thread, which 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 `Thread::new` call, like this:

```
    use Thread;
    $Param3 = "foo";
    $thr = new Thread \&sub1, "Param 1", "Param 2", $Param3;
    $thr = new Thread \&sub1, @ParamList;
    $thr = new Thread \&sub1, qw(Param1 Param2 $Param3);

    sub sub1 {
        my @InboundParameters = @_;
        print "In the thread\n";
        print "got parameters >", join("<>", @InboundParameters), "<\n";
    }
```

The subroutine runs like a normal Perl subroutine, and the call to new Thread returns whatever the subroutine returns.

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.

The other way to spawn a new thread is with async(), which is a way to spin off a chunk of code like eval(), but into its own thread:

```
    use Thread qw(async);

    $LineCount = 0;

    $thr = async {
        while(<>) {$LineCount++}
        print "Got $LineCount lines\n";
    };

    print "Waiting for the linecount to end\n";
    $thr->join;
    print "All done\n";
```

You'll notice we did a use Thread qw(async) in that example. async is not exported by default, so if you want it, you'll either need to import it before you use it or fully qualify it as Thread::async. You'll also note that there's a semicolon after the closing brace. That's because async() treats the following block as an anonymous subroutine, so the semicolon is necessary.

Like eval(), the code executes in the same context as it would if it weren't spun off. Since both the code inside and after the async start executing, you need to be careful with any shared resources. Locking and other synchronization techniques are covered later.

51.7.3 Giving up control

There are times when you may find it useful to have a thread explicitly give up the CPU to another thread. Your threading package might not support preemptive multitasking for threads, for example, or you may be doing something compute-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 Thread qw(yield async);
    async {
        my $foo = 50;
        while ($foo--) { print "first async\n" }
        yield;
        $foo = 50;
        while ($foo--) { print "first async\n" }
    };
    async {
        my $foo = 50;
        while ($foo--) { print "second async\n" }
        yield;
        $foo = 50;
        while ($foo--) { print "second async\n" }
    };
```

51.7.4 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 scalars it might return, you can use the join() method.

```
use Thread;
$thr = new Thread \&sub1;

@ReturnData = $thr->join;
print "Thread returned @ReturnData";

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. detach() is covered later in the article.

51.7.5 Errors In Threads

So what happens when an error occurs in a thread? Any errors that could be caught with eval() are postponed until the thread is joined. If your program never joins, the errors appear when your program exits.
Errors deferred until a join() can be caught with eval():

```
use Thread qw(async);
$thr = async {$b = 3/0};   # Divide by zero error
$foo = eval {$thr->join};
if ($@) {
    print "died with error $@\n";
} else {
    print "Hey, why aren't you dead?\n";
}
```

eval() passes any results from the joined thread back unmodified, so if you want the return value of the thread, this is your only chance to get them.

51.7.6 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 Thread;
$thr = new Thread \&sub1; # Spawn the thread

$thr->detach; # Now we officially don't care any more

sub sub1 {
    $a = 0;
    while (1) {
        $a++;
        print "\$a is $a\n";
        sleep 1;
    }
}
```

Once a thread is detached, it may not be joined, and any output that it might have produced (if it was done and waiting for a join) is lost.

51.8 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.

51.8.1 Shared And Unshared Data

The single most important thing to remember when using threads is that all threads potentially have access to all the data anywhere in your program. While this is true with a nonthreaded Perl program as well, it's especially important to remember with a threaded program, since more than one thread can be accessing this data at once.

Perl's scoping rules don't change because you're using threads. If a subroutine (or block, in the case of async()) could see a variable if you weren't running with threads, it can see it if you are. This is especially important for the subroutines that create, and makes `my` variables even more important. Remember–if your variables aren't lexically scoped (declared with `my`) you're probably sharing them between threads.

51.8.2 Thread Pitfall: Races

While threads bring a new set of useful tools, they also bring a number of pitfalls. One pitfall is the race condition:

```
    use Thread;
    $a = 1;
    $thr1 = Thread->new(\&sub1);
    $thr2 = Thread->new(\&sub2);

    sleep 10;
    print "$a\n";

    sub sub1 { $foo = $a; $a = $foo + 1; }
    sub sub2 { $bar = $a; $a = $bar + 1; }
```

What do you think $a will be? The answer, unfortunately, is "it depends." Both sub1() and sub2() access the global variable $a, once to read and once to write. Depending on factors ranging from your thread implementation's scheduling algorithm to the phase of the moon, $a 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 Thread qw(async);
    $a = 2;
    async{ $b = $a; $a = $b + 1; };
    async{ $c = $a; $a = $c + 1; };
```

Two threads both access $a. Each thread can potentially be interrupted at any point, or be executed in any order. At the end, $a could be 3 or 4, and both $b and $c could be 2 or 3.

Whenever your program accesses data or resources that can be accessed by other threads, you must take steps to coordinate access or risk data corruption and race conditions.

51.8.3 Controlling access: lock()

The lock() function takes a variable (or subroutine, but we'll get to that later) and puts a lock on it. No other thread may lock the variable until the locking thread exits the innermost block containing the lock. Using lock() is straightforward:

```
use Thread qw(async);
$a = 4;
$thr1 = async {
    $foo = 12;
    {
        lock ($a); # Block until we get access to $a
        $b = $a;
        $a = $b * $foo;
    }
    print "\$foo was $foo\n";
};
$thr2 = async {
    $bar = 7;
    {
        lock ($a); # Block until we can get access to $a
        $c = $a;
        $a = $c * $bar;
    }
    print "\$bar was $bar\n";
};
$thr1->join;
$thr2->join;
print "\$a is $a\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 innermost 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. Locked subroutines behave differently, however. We'll cover that later in the article.

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.

Finally, 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.

51.8.4 Thread Pitfall: Deadlocks

Locks are a handy tool to synchronize access to data. Using them properly is the key to safe shared data. Unfortunately, locks aren't without their dangers. Consider the following code:

```
use Thread qw(async yield);
$a = 4;
$b = "foo";
async {
    lock($a);
    yield;
    sleep 20;
    lock ($b);
};
async {
    lock($b);
```

```
        yield;
        sleep 20;
        lock ($a);
    };
```

This program will probably hang until you kill it. The only way it won't hang is if one of the two async() routines acquires both locks first. A guaranteed-to-hang version is more complicated, but the principle is the same.

The first thread spawned by async() will grab a lock on $a then, a second or two later, try to grab a lock on $b. Meanwhile, the second thread grabs a lock on $b, then later tries to grab a lock on $a. 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 $a, $b, and $c, always lock $a before $b, and $b before $c. It's also best to hold on to locks for as short a period of time to minimize the risks of deadlock.

51.8.5 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 Thread qw(async);
    use Thread::Queue;

    my $DataQueue = new Thread::Queue;
    $thr = async {
        while ($DataElement = $DataQueue->dequeue) {
            print "Popped $DataElement off the queue\n";
        }
    };

    $DataQueue->enqueue(12);
    $DataQueue->enqueue("A", "B", "C");
    $DataQueue->enqueue(\$thr);
    sleep 10;
    $DataQueue->enqueue(undef);
```

You create the queue with new Thread::Queue. 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.

51.9 Threads And Code

In addition to providing thread-safe access to data via locks and queues, threaded Perl also provides general-purpose semaphores for coarser synchronization than locks provide and thread-safe access to entire subroutines.

51.9.1 Semaphores: Synchronizing Data Access

Semaphores are a kind of generic locking mechanism. Unlike lock, which gets a lock on a particular scalar, Perl doesn't associate any particular thing with a semaphore so you can use them to control access to anything you like. In addition, semaphores can allow more than one thread to access a resource at once, though by default semaphores only allow one thread access at a time.

Basic semaphores

Semaphores have two methods, down and up. down decrements the resource count, while up increments it. down calls will block if the semaphore's current count would decrement below zero. This program gives a quick demonstration:

```
use Thread qw(yield);
use Thread::Semaphore;
my $semaphore = new Thread::Semaphore;
$GlobalVariable = 0;

$thr1 = new Thread \&sample_sub, 1;
$thr2 = new Thread \&sample_sub, 2;
$thr3 = new Thread \&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
        yield;
        sleep 2;
        $LocalCopy++;
        $GlobalVariable = $LocalCopy;
        $semaphore->up;
    }
}
```

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.

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. down() decrements the counter and up() increments the counter. By default, semaphores are created with the counter set to one, down() decrements by one, and up() increments by one. 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. $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/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' 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.

51.9.2 Attributes: Restricting Access To Subroutines

In addition to synchronizing access to data or resources, you might find it useful to synchronize access to subroutines. You may be accessing a singular machine resource (perhaps a vector processor), or find it easier to serialize calls to a particular subroutine than to have a set of locks and semaphores.

One of the additions to Perl 5.005 is subroutine attributes. The Thread package uses these to provide several flavors of serialization. It's important to remember that these attributes are used in the compilation phase of your program so you can't change a subroutine's behavior while your program is actually running.

51.9.3 Subroutine Locks

The basic subroutine lock looks like this:

```
    sub test_sub :locked {
    }
```

This ensures that only one thread will be executing this subroutine at any one time. Once a thread calls this subroutine, any other thread that calls it will block until the thread in the subroutine exits it. A more elaborate example looks like this:

```
    use Thread qw(yield);

    new Thread \&thread_sub, 1;
    new Thread \&thread_sub, 2;
    new Thread \&thread_sub, 3;
    new Thread \&thread_sub, 4;

    sub sync_sub :locked {
        my $CallingThread = shift @_;
        print "In sync_sub for thread $CallingThread\n";
        yield;
        sleep 3;
        print "Leaving sync_sub for thread $CallingThread\n";
    }

    sub thread_sub {
        my $ThreadID = shift @_;
        print "Thread $ThreadID calling sync_sub\n";
        sync_sub($ThreadID);
        print "$ThreadID is done with sync_sub\n";
    }
```

The `locked` attribute tells perl to lock sync_sub(), and if you run this, you can see that only one thread is in it at any one time.

51.9.4 Methods

Locking an entire subroutine can sometimes be overkill, especially when dealing with Perl objects. When calling a method for an object, for example, you want to serialize calls to a method, so that only one thread will be in the subroutine for a particular object, but threads calling that subroutine for a different object aren't blocked. The method attribute indicates whether the subroutine is really a method.

```
    use Thread;

    sub tester {
        my $thrnum = shift @_;
        my $bar = new Foo;
        foreach (1..10) {
            print "$thrnum calling per_object\n";
            $bar->per_object($thrnum);
            print "$thrnum out of per_object\n";
            yield;
            print "$thrnum calling one_at_a_time\n";
            $bar->one_at_a_time($thrnum);
            print "$thrnum out of one_at_a_time\n";
            yield;
        }
    }

    foreach my $thrnum (1..10) {
        new Thread \&tester, $thrnum;
    }

    package Foo;
    sub new {
        my $class = shift @_;
        return bless [@_], $class;
    }

    sub per_object :locked :method {
        my ($class, $thrnum) = @_;
        print "In per_object for thread $thrnum\n";
        yield;
        sleep 2;
        print "Exiting per_object for thread $thrnum\n";
    }

    sub one_at_a_time :locked {
        my ($class, $thrnum) = @_;
        print "In one_at_a_time for thread $thrnum\n";
        yield;
        sleep 2;
        print "Exiting one_at_a_time for thread $thrnum\n";
    }
```

As you can see from the output (omitted for brevity; it's 800 lines) all the threads can be in per_object() simultaneously, but only one thread is ever in one_at_a_time() at once.

51.9.5 Locking A Subroutine

You can lock a subroutine as you would lock a variable. Subroutine locks work the same as specifying a `locked` attribute for the subroutine, and block all access to the subroutine for other threads until the lock goes out of scope. When the subroutine isn't locked, any number of threads can be in it at once, and getting a lock on a subroutine doesn't affect threads already in the subroutine. Getting a lock on a subroutine looks like this:

```
    lock(\&sub_to_lock);
```

Simple enough. Unlike the `locked` attribute, which is a compile time option, locking and unlocking a subroutine can be done at runtime at your discretion. There is some runtime penalty to using lock(\&sub) instead of the `locked` attribute, so make sure you're choosing the proper method to do the locking.

You'd choose lock(\&sub) when writing modules and code to run on both threaded and unthreaded Perl, especially for code that will run on 5.004 or earlier Perls. In that case, it's useful to have subroutines that should be serialized lock themselves if they're running threaded, like so:

```
    package Foo;
    use Config;
    $Running_Threaded = 0;

    BEGIN { $Running_Threaded = $Config{'usethreads'} }

    sub sub1 { lock(\&sub1) if $Running_Threaded }
```

This way you can ensure single-threadedness regardless of which version of Perl you're running.

51.10 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.

51.10.1 What Thread Am I In?

The Thread->self 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 the thread creation.

51.10.2 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.

51.10.3 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.

51.10.4 What Threads Are Running?

Thread->list returns a list of thread objects, one for each thread that's currently running. Handy for a number of things, including cleaning up at the end of your program:

```
# Loop through all the threads
foreach $thr (Thread->list) {
    # Don't join the main thread or ourselves
    if ($thr->tid && !Thread::equal($thr, Thread->self)) {
        $thr->join;
    }
}
```

The example above is just for illustration. It isn't strictly necessary to join all the threads you create, since Perl detaches all the threads before it exits.

51.11 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 -w
# prime-pthread, courtesy of Tom Christiansen

use strict;

use Thread;
use Thread::Queue;

my $stream = new Thread::Queue;
my $kid    = new Thread(\&check_num, $stream, 2);

for my $i ( 3 .. 1000 ) {
    $stream->enqueue($i);
}

$stream->enqueue(undef);
$kid->join();

sub check_num {
    my ($upstream, $cur_prime) = @_;
    my $kid;
    my $downstream = new Thread::Queue;
    while (my $num = $upstream->dequeue) {
        next unless $num % $cur_prime;
        if ($kid) {
           $downstream->enqueue($num);
                 } else {
           print "Found prime $num\n";
               $kid = new Thread(\&check_num, $downstream, $num);
        }
    }
    $downstream->enqueue(undef) if $kid;
    $kid->join()         if $kid;
}
```

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 that it funnels numbers that have failed the check into. 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 lets 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's checking (line 20), we create a new queue (line 22) and reserve a scalar for the thread that we're likely to create later (line 21).

The while loop from lines 23 to line 31 grabs a scalar off the input queue and checks against the prime this thread is responsible for. Line 24 checks to see if there's a remainder when we modulo the number to be checked against 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 26) or create a new thread if we haven't.

The new thread creation is line 29. We pass on to it a reference to the queue we've created, and the prime number we've found.

Finally, once the loop terminates (because we got a 0 or undef in the queue, which serves as a note to die), we pass on the notice to our child and wait for it to exit if we've created a child (Lines 32 and 37).

Meanwhile, back in the main thread, we create a queue (line 9) and the initial child thread (line 10), and pre-seed it with the first prime: 2. Then we queue all the numbers from 3 to 1000 for checking (lines 12-14), then queue a die notice (line 16) and wait for the first child thread to terminate (line 17). Because a child won't die until its child has died, we know that we're done once we return from the join.

That's how it works. It's pretty simple; as with many Perl programs, the explanation is much longer than the program.

51.12 Conclusion

A complete thread tutorial could fill a book (and has, many times), but this should get you well on your way. The final authority on how Perl's threads behave is the documentation bundled with the Perl distribution, but with what we've covered in this article, you should be well on your way to becoming a threaded Perl expert.

51.13 Bibliography

Here's a short bibliography courtesy of Jürgen Christoffel:

51.13.1 Introductory Texts

Birrell, Andrew D. An Introduction to Programming with Threads. Digital Equipment Corporation, 1989, DEC-SRC Research Report #35 online as http://www.research.digital.com/SRC/staff/birrell/bib.html (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).

51.13.2 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

51.13.3 Other References

Arnold, Ken and James Gosling. The Java Programming Language, 2nd ed. Addison-Wesley, 1998, ISBN 0-201-31006-6.

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).

51.14 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.

51.15 AUTHOR

Dan Sugalski <sugalskd@ous.edu>

51.16 Copyrights

Chapter 52

perlport

Writing portable Perl

52.1 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 have 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, Mac OS, 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 (§52.2, platform-specific issues (§52.4, and built-in perl functions that behave differently on various ports (§52.5.

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">`).

52.2 ISSUES

52.2.1 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`, and Mac OS uses `\015`.

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`. In DOSish perls, `\n` usually means `\012`, but when accessing a file in "text" mode, STDIO 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.

A common cause of unportable programs is the misuse of chop() to trim newlines:

```
# XXX UNPORTABLE!
while(<FILE>) {
    chop;
    @array = split(/:/);
    #...
}
```

You can get away with this on Unix and Mac OS (they have a single character end-of-line), but the same program will break under DOSish perls because you're only chop()ing half the end-of-line. Instead, chomp() should be used to trim newlines. The Dunce::Files module can help audit your code for misuses of chop().

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 in safety.

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>) {
    # ...
}
```

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
```

52.2.2 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 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.

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 (included in the standard distribution as of Perl 5.005) 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 (0x7FFFFFFF), that's how far EBCDIC, or more precisely UTF-EBCDIC will go.

52.2.3 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 uses `:` 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
    $file = catfile(curdir(), 'temp', 'file.txt');
    # on Unix and Win32, './temp/file.txt'
    # on Mac OS, ':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) 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(FILE, '<', $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 to 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.)

52.2.4 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.

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 `$!`, neither numeric nor especially the strings values– users may switch their locales causing error messages to be translated into their languages. If you can trust a POSIXish environment, you can portably use the symbols defined by the Errno module, like ENOENT. And don’t trust on the values of `$!` at all except immediately after a failed system call.

52.2.5 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; $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; $thisperl = $Config{perlpath}; if ($^O ne ’VMS’) {$thisperl .= $Config{_exe} unless $thisperl =~ m/$Config{_exe}$/i;}

52.2.6 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 things like 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.

52.2.7 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 expose a common interface).

52.2.8 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.

52.2.9 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 NBDM_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.

52.2.10 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 to 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;
    $offset = Time::Local::timegm(0, 0, 0, 1, 0, 70);
```

The value for `$offset` in Unix will be `0`, but in Mac OS will be some large number. `$offset` can then be added to a Unix time value to get what should be the proper value on any system.

On Windows (at least), you shouldn't pass a negative value to `gmtime` or `localtime`.

52.2.11 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); use for example symbolic character classes like `[:print:]`.

Do not assume that the alphabetic characters are encoded contiguously (in the numeric sense). There may be gaps.

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'.

52.2.12 Internationalisation

If you may assume POSIX (a rather large assumption), you may read more about the POSIX locale system from *perllocale*. 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 *perluniintro* and *perlunicode* 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 expression being a curious string), you can often also use the `\xHH` notation instead of embedding the bytes as-is. If they are in some particular legacy encoding (ether single-byte or something more complicated), you can use the `encoding` pragma. (If you want to write your code in UTF-8, you can use either the `utf8` pragma, or the `encoding` pragma.) The `bytes` and `utf8` pragmata are available since Perl 5.6.0, and the `encoding` pragma since Perl 5.8.0.

52.2.13 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:

```
    # NOTE: this is no longer "bad" in perl5.005
    for (0..10000000) {}                       # bad
    for (my $x = 0; $x <= 10000000; ++$x) {}   # good

    @lines = <VERY_LARGE_FILE>;                # bad

    while (<FILE>) {$file .= $_}               # sometimes bad
    $file = join('', <FILE>);                  # 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 that the first.

52.2.14 Security

Most multi-user platforms provide basic levels of security, usually implemented at the filesystem level. Some, however, do not– unfortunately. 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 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 the `$<` and `$>` (or the `$(` and `$)`) 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.)

52.2.15 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 §52.4.

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.

52.3 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.

Mailing list: cpan-testers@perl.org

Testing results: http://testers.cpan.org/

52.4 PLATFORMS

As of version 5.002, 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.

52.4.1 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
    dgux          dgux       AViiON-dgux
    DYNIX/ptx     dynixptx   i386-dynixptx
    FreeBSD       freebsd    freebsd-i386
    Linux         linux      arm-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
    MachTen PPC   machten    powerpc-machten
    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`.

52.4.2 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:

```
    $filespec0 = "c:/foo/bar/file.txt";
    $filespec1 = "c:\\foo\\bar\\file.txt";
    $filespec2 = 'c:\foo\bar\file.txt';
    $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 §52.2.1). `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 xx
Windows XP    MSWin32    MSWin32-x86        2      ?
Windows CE    MSWin32    ?                  3
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, http://www.leo.org/pub/comp/os/os2/leo/gnu/emx+gcc/index.html or 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*

52.4.3 Mac OS

Any module requiring XS compilation is right out for most people, because MacPerl is built using non-free (and non-cheap!) compilers. Some XS modules that can work with MacPerl are built and distributed in binary form on CPAN.

Directories are specified as:

```
    volume:folder:file              for absolute pathnames
    volume:folder:                  for absolute pathnames
    :folder:file                    for relative pathnames
    :folder:                        for relative pathnames
    :file                           for relative pathnames
    file                            for relative pathnames
```

Files are stored in the directory in alphabetical order. Filenames are limited to 31 characters, and may include any character except for null and `:`, which is reserved as the path separator.

Instead of `flock`, see `FSpSetFLock` and `FSpRstFLock` in the Mac::Files module, or `chmod(0444, ...)` and `chmod(0666, ...)`.

In the MacPerl application, you can't run a program from the command line; programs that expect `@ARGV` to be populated can be edited with something like the following, which brings up a dialog box asking for the command line arguments.

```
    if (!@ARGV) {
        @ARGV = split /\s+/, MacPerl::Ask('Arguments?');
    }
```

A MacPerl script saved as a "droplet" will populate `@ARGV` with the full pathnames of the files dropped onto the script.

Mac users can run programs under a type of command line interface under MPW (Macintosh Programmer's Workshop, a free development environment from Apple). MacPerl was first introduced as an MPW tool, and MPW can be used like a shell:

```
    perl myscript.plx some arguments
```

ToolServer is another app from Apple that provides access to MPW tools from MPW and the MacPerl app, which allows MacPerl programs to use `system`, backticks, and piped `open`.

"Mac OS" is the proper name for the operating system, but the value in `$^O` is "MacOS". To determine architecture, version, or whether the application or MPW tool version is running, check:

```
    $is_app    = $MacPerl::Version =~ /App/;
    $is_tool   = $MacPerl::Version =~ /MPW/;
    ($version) = $MacPerl::Version =~ /^(\S+)/;
    $is_ppc    = $MacPerl::Architecture eq 'MacPPC';
    $is_68k    = $MacPerl::Architecture eq 'Mac68K';
```

Mac OS X, based on NeXT's OpenStep OS, runs MacPerl natively, under the "Classic" environment. There is no "Carbon" version of MacPerl to run under the primary Mac OS X environment. Mac OS X and its Open Source version, Darwin, both run Unix perl natively.

Also see:

- MacPerl Development, http://dev.macperl.org/ .
- The MacPerl Pages, http://www.macperl.com/ .
- The MacPerl mailing lists, http://lists.perl.org/ .

52.4.4 VMS

Perl on VMS is discussed in *perlvms* in the perl distribution. 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
```

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>;`.

Filenames are in the format "name.extension;version". The maximum length for filenames is 39 characters, and the maximum length for extensions is also 39 characters. Version is a number from 1 to 32767. Valid characters are `/[A-Z0-9$_-]/`.

VMS's RMS filesystem is case-insensitive and does not preserve case. `readdir` returns lowercased filenames, but specifying a file for opening remains case-insensitive. Files without extensions have a trailing period on them, so doing a `readdir` with a file named *A.;5* will return *a.* (though that file could be opened with `open(FH, 'A')`).

RMS had an eight level limit on directory depths from any rooted logical (allowing 16 levels overall) prior to VMS 7.2. Hence `PERL_ROOT:[LIB.2.3.4.5.6.7.8]` is a valid directory specification but `PERL_ROOT:[LIB.2.3.4.5.6.7.8.9]` is not. *Makefile.PL* authors might have to take this into account, but at least they can refer to the former as `/PERL_ROOT/lib/2/3/4/5/6/7/8/`.

The VMS::Filespec module, which gets installed as part of the build process on VMS, is a pure Perl module that can easily be installed on non-VMS platforms and can be helpful for conversions to and from RMS native formats.

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 organiztion and record format. The VMS::Stdio module provides access to the special fopen() requirements of files with unusual attributes on VMS.

TCP/IP stacks are optional on VMS, so socket routines might not be implemented. UDP sockets may not be supported.

The value of `$^O` on OpenVMS is "VMS". 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 (grep(/VMS_AXP/, @INC)) {
        print "I'm on Alpha!\n";

    } elsif (grep(/VMS_VAX/, @INC)) {
        print "I'm on VAX!\n";

    } else {
        print "I'm not so sure about where $^O is...\n";
    }
```

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), *perlvms*
- vmsperl list, majordomo@perl.org

  (Put the words `subscribe vmsperl` in message body.)
- vmsperl on the web, http://www.sidhe.org/vmsperl/index.html

52.4.5 VOS

Perl on VOS 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:

```
    C<< $ perl -ne "print if /perl_setup/i" >system>notices >>
    C<< $ perl -ne "print if /perl_setup/i" /system/notices >>
```

or even a mixture of both as in:

```
    C<< $ 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. Note that VOS limits file names to 32 or fewer characters.

Perl on VOS can be built using two different compilers and two different versions of the POSIX runtime. The recommended method for building full Perl is with the GNU C compiler and the generally-available version of VOS POSIX support. See *README.vos* (installed as *perlvos*) for restrictions that apply when Perl is built using the VOS Standard C compiler or the alpha version of VOS POSIX support.

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;
    }

    if (grep(/860/, @INC)) {
        print "This box is a Stratus XA/R!\n";
```

```
} elsif (grep(/7100/, @INC)) {
    print "This box is a Stratus HP 7100 or 8xxx!\n";

} elsif (grep(/8000/, @INC)) {
    print "This box is a Stratus HP 8xxx!\n";

} else {
    print "This box is a Stratus 68K!\n";
}
```

Also see:

- *README.vos* (installed as *perlvos*)
- The VOS mailing list.

  There is no specific mailing list for Perl on VOS. You can post comments to the comp.sys.stratus newsgroup, or subscribe to the general Stratus mailing list. Send a letter with "subscribe Info-Stratus" in the message body to majordomo@list.stratagy.com.
- VOS Perl on the web at http://ftp.stratus.com/pub/vos/posix/posix.html

52.4.6 EBCDIC Platforms

Recent versions of Perl have been ported to platforms such as OS/400 on AS/400 minicomputers as well as OS/390, 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). 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 5.8.1/5.9.0 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 §52.2.1).

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 OS/390 & VM/ESA):

```
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
VM/ESA        vmesa      vmesa
```

Some simple tricks for determining if you are running on an EBCDIC platform could include any of the following (perhaps all):

```
if ("\t" eq "\05")   { 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, *README.os390*, *perlbs2000*, *README.vmesa*, *perlebcdic*.
- 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.

52.4.7 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).

52.4.8 Other perls

Perl has been ported to many platforms that do not fit into any of the categories listed above. Some, such as AmigaOS, Atari MiNT, BeOS, HP MPE/iX, 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
    BeOS          beos
    MPE/iX        mpeix      PA-RISC1.1
```

See also:

- Amiga, *README.amiga* (installed as *perlamiga*).
- Atari, *README.mint* and Guido Flohr's web page http://stud.uni-sb.de/˜gufl0000/
- Be OS, *README.beos*
- HP 300 MPE/iX, *README.mpeix* and Mark Bixby's web page http://www.bixby.org/mark/perlix.html
- 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*

52.5 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.

52.5.1 Alphabetical Listing of Perl Functions

-X FILEHANDLE

-X EXPR

-X

`-r`, `-w`, and `-x` have a limited meaning only; directories and applications are executable, and there are no uid/gid considerations. `-o` is not supported. (Mac OS)

`-r`, `-w`, `-x`, and `-o` tell whether the file is accessible, which may not reflect UIC-based file protections. (VMS)

`-s` returns the size of the data fork, not the total size of data fork plus resource fork. (Mac OS).

`-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`. (Mac OS, Win32, VMS, RISC OS)

`-b`, `-c`, `-k`, `-g`, `-p`, `-u`, `-A` are not implemented. (Mac OS)

`-g`, `-k`, `-l`, `-p`, `-u`, `-A` are not particularly meaningful. (Win32, VMS, RISC OS)

`-d` is true if passed a device spec without an explicit directory. (VMS)

`-T` and `-B` are implemented, but might misclassify Mac text files with foreign characters; this is the case will all platforms, but may affect Mac OS often. (Mac OS)

`-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)

binmode FILEHANDLE

Meaningless. (Mac OS, 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 LIST

Only limited meaning. Disabling/enabling write permission is mapped to locking/unlocking the file. (Mac OS)

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)

chown LIST

Not implemented. (Mac OS, Win32, Plan 9, RISC OS, VOS)

Does nothing, but won't fail. (Win32)

chroot FILENAME

chroot

Not implemented. (Mac OS, Win32, VMS, Plan 9, RISC OS, VOS, VM/ESA)

crypt PLAINTEXT,SALT

May not be available if library or source was not provided when building perl. (Win32)

Not implemented. (VOS)

dbmclose HASH

Not implemented. (VMS, Plan 9, VOS)

dbmopen HASH,DBNAME,MODE

Not implemented. (VMS, Plan 9, VOS)

dump LABEL

Not useful. (Mac OS, RISC OS)

Not implemented. (Win32)

Invokes VMS debugger. (VMS)

exec LIST

Not implemented. (Mac OS)

Implemented via Spawn. (VM/ESA)

Does not automatically flush output handles on some platforms. (SunOS, Solaris, HP-UX)

exit EXPR

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. (VMS)

fcntl FILEHANDLE,FUNCTION,SCALAR

Not implemented. (Win32, VMS)

flock FILEHANDLE,OPERATION

Not implemented (Mac OS, VMS, RISC OS, VOS).

Available only on Windows NT (not on Windows 95). (Win32)

fork

Not implemented. (Mac OS, AmigaOS, RISC OS, VOS, VM/ESA, VMS)

Emulated using multiple interpreters. See *perlfork*. (Win32)

Does not automatically flush output handles on some platforms. (SunOS, Solaris, HP-UX)

getlogin

Not implemented. (Mac OS, RISC OS)

getpgrp PID

Not implemented. (Mac OS, Win32, VMS, RISC OS, VOS)

getppid

Not implemented. (Mac OS, Win32, RISC OS)

getpriority WHICH,WHO

Not implemented. (Mac OS, Win32, VMS, RISC OS, VOS, VM/ESA)

getpwnam NAME

Not implemented. (Mac OS, Win32)

Not useful. (RISC OS)

getgrnam NAME

Not implemented. (Mac OS, Win32, VMS, RISC OS)

getnetbyname NAME

Not implemented. (Mac OS, Win32, Plan 9)

getpwuid UID

Not implemented. (Mac OS, Win32)

Not useful. (RISC OS)

getgrgid GID

Not implemented. (Mac OS, Win32, VMS, RISC OS)

getnetbyaddr ADDR,ADDRTYPE

Not implemented. (Mac OS, Win32, Plan 9)

getprotobynumber NUMBER

Not implemented. (Mac OS)

getservbyport PORT,PROTO

Not implemented. (Mac OS)

getpwent

Not implemented. (Mac OS, Win32, VM/ESA)

getgrent

Not implemented. (Mac OS, Win32, VMS, VM/ESA)

gethostbyname

`gethostbyname('localhost')` does not work everywhere: you may have to use `gethostbyname('127.0.0.1')`. (Mac OS, Irix 5)

gethostent

Not implemented. (Mac OS, Win32)

getnetent

Not implemented. (Mac OS, Win32, Plan 9)

getprotoent

Not implemented. (Mac OS, Win32, Plan 9)

getservent

Not implemented. (Win32, Plan 9)

sethostent STAYOPEN

Not implemented. (Mac OS, Win32, Plan 9, RISC OS)

setnetent STAYOPEN

Not implemented. (Mac OS, Win32, Plan 9, RISC OS)

setprotoent STAYOPEN

Not implemented. (Mac OS, Win32, Plan 9, RISC OS)

setservent STAYOPEN

Not implemented. (Plan 9, Win32, RISC OS)

endpwent

Not implemented. (Mac OS, MPE/iX, VM/ESA, Win32)

endgrent

Not implemented. (Mac OS, MPE/iX, RISC OS, VM/ESA, VMS, Win32)

endhostent

Not implemented. (Mac OS, Win32)

endnetent

Not implemented. (Mac OS, Win32, Plan 9)

endprotoent

Not implemented. (Mac OS, Win32, Plan 9)

endservent

Not implemented. (Plan 9, Win32)

getsockopt SOCKET,LEVEL,OPTNAME

Not implemented. (Plan 9)

glob EXPR

glob

This operator is implemented via the File::Glob extension on most platforms. See *File::Glob* for portability information.

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 SIGNAL, LIST

`kill(0, LIST)` is implemented for the sake of taint checking; use with other signals is unimplemented. (Mac OS)

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)

link OLDFILE,NEWFILE

Not implemented. (Mac OS, MPE/iX, VMS, RISC OS)

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 (Windows NT and Windows 2000) under NTFS only.

lstat FILEHANDLE

lstat EXPR

lstat

Not implemented. (VMS, RISC OS)

Return values (especially for device and inode) may be bogus. (Win32)

msgctl ID,CMD,ARG

msgget KEY,FLAGS

msgsnd ID,MSG,FLAGS

msgrcv ID,VAR,SIZE,TYPE,FLAGS

Not implemented. (Mac OS, Win32, VMS, Plan 9, RISC OS, VOS)

open FILEHANDLE,EXPR

open FILEHANDLE

The | variants are supported only if ToolServer is installed. (Mac OS)

open to |- and -| are unsupported. (Mac OS, Win32, RISC OS)

Opening a process does not automatically flush output handles on some platforms. (SunOS, Solaris, HP-UX)

pipe READHANDLE,WRITEHANDLE

Very limited functionality. (MiNT)

readlink EXPR

readlink

Not implemented. (Win32, VMS, RISC OS)

select RBITS,WBITS,EBITS,TIMEOUT

Only implemented on sockets. (Win32, VMS)

Only reliable on sockets. (RISC OS)

Note that the `select FILEHANDLE` form is generally portable.

semctl ID,SEMNUM,CMD,ARG

semget KEY,NSEMS,FLAGS

semop KEY,OPSTRING

Not implemented. (Mac OS, Win32, VMS, RISC OS, VOS)

setgrent

Not implemented. (Mac OS, MPE/iX, VMS, Win32, RISC OS)

setpgrp PID,PGRP

Not implemented. (Mac OS, Win32, VMS, RISC OS, VOS)

setpriority WHICH,WHO,PRIORITY

Not implemented. (Mac OS, Win32, VMS, RISC OS, VOS)

setpwent

Not implemented. (Mac OS, MPE/iX, Win32, RISC OS)

setsockopt SOCKET,LEVEL,OPTNAME,OPTVAL

Not implemented. (Plan 9)

shmctl ID,CMD,ARG

shmget KEY,SIZE,FLAGS

shmread ID,VAR,POS,SIZE

shmwrite ID,STRING,POS,SIZE

Not implemented. (Mac OS, Win32, VMS, RISC OS, VOS)

sockatmark SOCKET

A relatively recent addition to socket functions, may not be implemented even in UNIX platforms.

socketpair SOCKET1,SOCKET2,DOMAIN,TYPE,PROTOCOL

Not implemented. (Win32, VMS, RISC OS, VOS, VM/ESA)

stat FILEHANDLE

stat EXPR

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.

mtime and atime are the same thing, and ctime is creation time instead of inode change time. (Mac OS).

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)

symlink OLDFILE,NEWFILE

Not implemented. (Win32, VMS, RISC OS)

syscall LIST

Not implemented. (Mac OS, Win32, VMS, RISC OS, VOS, VM/ESA)

sysopen FILEHANDLE,FILENAME,MODE,PERMS

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, VM/ESA)

system LIST

In general, do not assume the UNIX/POSIX semantics that you can shift `$?` right by eight to get the exit value, or that `$? & 127` would give you the number of the signal that terminated the program, or that `$? & 128` would test true if the program was terminated by a coredump. Instead, use the POSIX W*() interfaces: for example, use WIFEXITED($?) and WEXITVALUE($?) to test for a normal exit and the exit value, WIFSIGNALED($?) and WTERMSIG($?) for a signal exit and the signal. Core dumping is not a portable concept, so there's no portable way to test for that.

Only implemented if ToolServer is installed. (Mac OS)

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)

Far from being POSIX compliant. Because there may be no underlying /bin/sh tries to work around the problem by forking and execing the first token in its argument string. Handles basic redirection ("<" or ">") on its own behalf. (MiNT)

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'`). For more details see `$?` in *perlvms*. (VMS)

times

Only the first entry returned is nonzero. (Mac OS)

"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 FILEHANDLE,LENGTH

truncate EXPR,LENGTH

Not implemented. (Older versions of VMS)

Truncation to zero-length 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 EXPR

umask

Returns undef where unavailable, as of version 5.005.

`umask` works but the correct permissions are set only when the file is finally closed. (AmigaOS)

utime LIST

Only the modification time is updated. (BeOS, Mac OS, 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 PID,FLAGS

Not implemented. (Mac OS, VOS)

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)

52.6 CHANGES

v1.48, 02 February 2001

Various updates from perl5-porters over the past year, supported platforms update from Jarkko Hietaniemi.

v1.47, 22 March 2000

Various cleanups from Tom Christiansen, including migration of long platform listings from *perl*.

v1.46, 12 February 2000

Updates for VOS and MPE/iX. (Peter Prymmer) Other small changes.

v1.45, 20 December 1999

Small changes from 5.005_63 distribution, more changes to EBCDIC info.

v1.44, 19 July 1999

A bunch of updates from Peter Prymmer for `$^O` values, endianness, File::Spec, VMS, BS2000, OS/400.

v1.43, 24 May 1999

Added a lot of cleaning up from Tom Christiansen.

v1.42, 22 May 1999

Added notes about tests, sprintf/printf, and epoch offsets.

v1.41, 19 May 1999

Lots more little changes to formatting and content.

Added a bunch of `$^O` and related values for various platforms; fixed mail and web addresses, and added and changed miscellaneous notes. (Peter Prymmer)

v1.40, 11 April 1999

Miscellaneous changes.

v1.39, 11 February 1999

Changes from Jarkko and EMX URL fixes Michael Schwern. Additional note about newlines added.

v1.38, 31 December 1998

More changes from Jarkko.

v1.37, 19 December 1998

More minor changes. Merge two separate version 1.35 documents.

v1.36, 9 September 1998

Updated for Stratus VOS. Also known as version 1.35.

v1.35, 13 August 1998

Integrate more minor changes, plus addition of new sections under §52.2: §52.2.2, §52.2.11, §52.2.12.

v1.33, 06 August 1998

Integrate more minor changes.

v1.32, 05 August 1998

Integrate more minor changes.

v1.30, 03 August 1998

Major update for RISC OS, other minor changes.

v1.23, 10 July 1998

First public release with perl5.005.

52.7 Supported Platforms

As of September 2003 (the Perl release 5.8.1), the following platforms are able to build Perl from the standard source code distribution available at http://www.cpan.org/src/index.html

```
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
LynxOS
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)
        SVR4
        Tru64 UNIX      (formerly DEC OSF/1, Digital UNIX)
        UNICOS
        UNICOS/mk
        UTS
        VOS
        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 5.8.1 release. There is a very good chance that many of these will work fine with the 5.8.1.

```
        DomainOS
        Hurd
        MachTen
        PowerMAX
        SCO SV
        Unixware
        Windows 3.1
```

Known to be broken for 5.8.0 and 5.8.1 (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
        MiNT
        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.

52.8 SEE ALSO

perlaix, *perlamiga*, *perlapollo*, *perlbeos*, *perlbs2000*, *perlce*, *perlcygwin*, *perldgux*, *perldos*, *perlepoc*, *perlebcdic*, *perlfreebsd*, *perlhurd*, *perlhpux*, *perlirix*, *perlmachten*, *perlmacos*, *perlmacosx*, *perlmint*, *perlmpeix*, *perlnetware*, *perlos2*, *perlos390*, *perlos400*, *perlplan9*, *perlqnx*, *perlsolaris*, *perltru64*, *perlunicode*, *perlvmesa*, *perlvms*, *perlvos*, *perlwin32*, and *Win32*.

52.9 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>.

Chapter 53

perllocale

Perl locale handling (internationalization and localization)

53.1 DESCRIPTION

Perl supports language-specific notions of data such as "is this a letter", "what is the uppercase equivalent of this letter", and "which of these letters comes first". These are important issues, especially for languages other than English–but also for English: it would be naïve to imagine that `A-Za-z` defines all the "letters" needed to write in English. Perl is also aware that some character other than '.' may be preferred as a decimal point, and that output date representations may be language-specific. The process of making an application take account of its users' preferences in such matters is called **internationalization** (often abbreviated as **i18n**); telling such an application about a particular set of preferences is known as **localization** (**l10n**).

Perl can understand language-specific data via the standardized (ISO C, XPG4, POSIX 1.c) method called "the locale system". The locale system is controlled per application using one pragma, one function call, and several environment variables.

NOTE: This feature is new in Perl 5.004, and does not apply unless an application specifically requests it–see Backward compatibility. The one exception is that write() now **always** uses the current locale - see §53.7.

53.2 PREPARING TO USE LOCALES

If Perl applications are to understand and present your data correctly according a locale of your choice, **all** of the following must be true:

- **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 The use locale pragma) where appropriate, and **at least one** of the following must be true:

- **The locale-determining environment variables (see §53.6) must be correctly set up** at the time the application is started, either by yourself or by whoever set up your system account.
- **The application must set its own locale** using the method described in The setlocale function.

53.3 USING LOCALES

53.3.1 The use locale pragma

By default, Perl ignores the current locale. The `use locale` pragma tells Perl to use the current locale for some operations:

- **The comparison operators** (`lt`, `le`, `cmp`, `ge`, and `gt`) and the POSIX string collation functions strcoll() and strxfrm() 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.
- **Regular expressions and case-modification functions** (uc(), lc(), ucfirst(), and lcfirst()) use `LC_CTYPE`
- **The formatting functions** (printf(), sprintf() and write()) use `LC_NUMERIC`
- **The POSIX date formatting function** (strftime()) uses `LC_TIME`.

`LC_COLLATE`, `LC_CTYPE`, and so on, are discussed further in LOCALE CATEGORIES.

The default behavior is restored with the `no locale` pragma, or upon reaching the end of block enclosing `use locale`.

The string result of any operation that uses locale information is tainted, as it is possible for a locale to be untrustworthy. See §53.5.

53.3.2 The setlocale function

You can switch locales as often as you wish at run time with the POSIX::setlocale() function:

```
# This functionality not usable prior to Perl 5.004
require 5.004;

# Import locale-handling tool set from POSIX module.
# This example uses: setlocale -- the function call
#                    LC_CTYPE -- explained below
use POSIX qw(locale_h);

# 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 default defined by LC_ALL/LC_CTYPE/LANG
# environment variables.  See below for documentation.
```

```
        # restore the old locale
        setlocale(LC_CTYPE, $old_locale);
```

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 LOCALE CATEGORIES and §53.6. 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 else 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().

If no second argument is provided and the category is LC_ALL, the result is implementation-dependent. It may be a string of concatenated locales names (separator also implementation-dependent) or a single locale name. Please consult your *setlocale*(3) 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.

If the second argument does not correspond to a valid locale, the locale for the category is not changed, and the function returns *undef*.

For further information about the categories, consult *setlocale*(3).

53.3.3 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.

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.

53.3.4 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. 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.

53.3.5 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 a zero value, for example "0". 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 *ENVIRONMENT* for the full list of relevant environment variables and USING LOCALES 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 in 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.

53.3.6 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 Finding locales. 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 Permanently fixing your system's locale configuration.

53.3.7 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 Finding locales about general rules.

53.3.8 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 Finding locales section is unfortunately a bit vague about the exact commands and places because these things are not that standardized.

53.3.9 The localeconv function

The POSIX::localeconv() function allows you to get particulars of the locale-dependent numeric formatting information specified by the current `LC_NUMERIC` and `LC_MONETARY` locales. (If you just want the name of the current locale for a particular category, use POSIX::setlocale() with a single parameter–see The setlocale function.)

```
        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 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:

```
        # See comments in previous example
        require 5.004;
        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";
```

53.3.10 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.

53.4 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 §53.6 for a discussion of these.

53.4.1 Category LC_COLLATE: Collation

In the scope of `use locale`, 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 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 USING LOCALES, `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.

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 Magic Variables in *perlguts*) 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() exist only to generate locale-dependent results, and so always obey the current `LC_COLLATE` locale.

53.4.2 Category LC_CTYPE: Character Types

In the scope of `use locale`, Perl obeys the `LC_CTYPE` locale setting. This controls the application's notion of which characters are alphabetic. This affects Perl's `\w` regular expression metanotation, which stands for alphanumeric characters–that is, alphabetic, numeric, and including other special characters such as the underscore or hyphen. (Consult *perlre* 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.

The `LC_CTYPE` locale also provides the map used in transliterating characters between lower and uppercase. This affects the case-mapping functions–lc(), lcfirst, uc(), and ucfirst(); case-mapping interpolation with `\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 POSIX character-class test functions–isalpha(), islower(), and so on. For example, if you move from the "C" locale to a 7-bit Scandinavian one, you may find–possibly to your surprise–that "|" moves from the ispunct() class to isalpha().

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) letters and digits–for example, in command strings–locale-aware applications should use `\w` inside a `no locale` block. See §53.5.

53.4.3 Category LC_NUMERIC: Numeric Formatting

In the scope of `use locale`, Perl obeys the `LC_NUMERIC` locale information, which controls an application's idea of how numbers should be formatted for human readability by the printf(), sprintf(), and write() functions. String-to-numeric conversion by the POSIX::strtod() function is also affected. In most implementations the only effect is to change the character used for the decimal point–perhaps from '.' to ','. These functions aren't aware of such niceties as thousands separation and so on. (See The localeconv function if you care about these things.)

Output produced by print() is also affected by the current locale: it depends on whether `use locale` or `no locale` is in effect, and corresponds to what you'd get from printf() in the "C" locale. The same is true for Perl's internal conversions between numeric and string formats:

```
use POSIX qw(strtod);
use locale;

$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`.

53.4.4 Category LC_MONETARY: Formatting of monetary amounts

The C standard defines the `LC_MONETARY` category, but no 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 takes no notice of it. If you really want to use `LC_MONETARY`, you can query its contents–see The localeconv function–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`.

53.4.5 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: as a function that exists only to generate locale-dependent results, strftime() 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`.

53.4.6 Other categories

The remaining locale category, `LC_MESSAGES` (possibly supplemented by others in particular implementations) is not currently used by Perl–except possibly to affect the behavior of library functions called by 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*.

53.5 SECURITY

Although the main discussion of Perl security issues can be found in *perlsec*, 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 *perlsec*) 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` or `\U`)

  Result string containing interpolated material is tainted if `use locale` is in effect.
- **Matching operator** (`m//`):

  Scalar true/false result never tainted.

  Subpatterns, either delivered as a list-context result or as $1 etc. are tainted if `use locale` is in effect, and the subpattern regular expression contains `\w` (to match an alphanumeric character), `\W` (non-alphanumeric character), `\s` (white-space character), or `\S` (non white-space character). The matched-pattern variable, $&, $‘ (pre-match), $’ (post-match), and $+ (last match) are also tainted if `use locale` is in effect and the regular expression contains `\w`, `\W`, `\s`, or `\S`.
- **Substitution operator** (`s///`):

  Has the same behavior as the match operator. Also, the left operand of `=~` becomes tainted when `use locale` in effect if modified as a result of a substitution based on a regular expression match involving `\w`, `\W`, `\s`, or `\S`; or of case-mapping with `\l`, `\L`,`\u` or `\U`.
- **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 `use locale` is in effect.
- **POSIX locale-dependent functions** (localeconv(), strcoll(), strftime(), strxfrm()):

  Results are never tainted.
- **POSIX character class tests** (isalnum(), isalpha(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), 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 $untainted_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.

53.6 ENVIRONMENT

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 that does not evaluate to integer zero–that is, "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.

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_....

53.7 NOTES

53.7.1 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 The setlocale function). 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 The use locale pragma) 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.

53.7.2 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.

53.7.3 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.

53.7.4 write() and LC_NUMERIC

Formats are the only part of Perl that unconditionally use information from a program's locale; if a program's environment specifies an LC_NUMERIC locale, it is always used to specify the decimal point character in formatted output. Formatted output cannot be controlled by `use locale` because the pragma is tied to the block structure of the program, and, for historical reasons, formats exist outside that block structure.

53.7.5 Freely available locale definitions

There is a large collection of locale definitions at ftp://dkuug.dk/i18n/WG15-collection . 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.

53.7.6 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**.

53.7.7 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. But, for now, it's the only standard we've got. This may be construed as a bug.

53.8 Unicode and UTF-8

The support of Unicode is new starting from Perl version 5.6, and more fully implemented in the version 5.8. See *perluniintro* and *perlunicode* for more details.

Usually locale settings and Unicode do not affect each other, but there are exceptions, see Locales in *perlunicode* for examples.

53.9 BUGS

53.9.1 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 the `use locale` is in effect. When confronted with such a system, please report in excruciating detail to <*perlbug@perl.org*>, and complain to your vendor: bug fixes may exist for these problems in your operating system. Sometimes such bug fixes are called an operating system upgrade.

53.10 SEE ALSO

I18N::Langinfo, *perluniintro*, *perlunicode*, *open*, isalnum in *POSIX*, isalpha in *POSIX*, isdigit in *POSIX*, isgraph in *POSIX*, islower in *POSIX*, isprint in *POSIX*, ispunct in *POSIX*, isspace in *POSIX*, isupper in *POSIX*, isxdigit in *POSIX*, localeconv in *POSIX*, setlocale in *POSIX*, strcoll in *POSIX*, strftime in *POSIX*, strtod in *POSIX*, strxfrm in *POSIX*.

53.11 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.

Last update: Thu Jun 11 08:44:13 MDT 1998

Chapter 54

perluniintro

Perl Unicode introduction

54.1 DESCRIPTION

This document gives a general idea of Unicode and how to use Unicode in Perl.

54.1.1 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 provide code points for characters in almost all modern character set standards, covering more than 30 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 4.0 in April 2003.

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 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*.

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, §54.1.14. 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 character consists either of a single code point, or 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*.

Whether to call these combining character sequences "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". The whole sequence could be seen as one "character", however, from the user's point of view, since that's probably what it looks like in the context of the user's language.

With this "whole sequence" view of characters, the total number of characters is open-ended. But in the programmer's "one unit is one character" point of view, the concept of "characters" is more deterministic. In this document, we take that second point of view: one "character" is one Unicode code point, be it a base character or a combining character.

For some combinations, there are *precomposed* characters. `LATIN CAPITAL LETTER A WITH ACUTE`, for example, is defined as a single code point. 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 the ISO 8859). In the general case, the composing method is more extensible. To support conversion between different compositions of the characters, various *normalization forms* to standardize representations are also defined.

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 also 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.

A common myth about Unicode is that it would be "16-bit", that is, Unicode is only represented as `0x10000` (or 65536) characters from `0x0000` to `0xFFFF`. **This is untrue.** Since Unicode 2.0 (July 1996), Unicode has been defined all the way up to 21 bits (`0x10FFFF`), and since Unicode 3.1 (March 2001), characters have been defined beyond `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.

Another myth is that the 256-character blocks have something to do with languages–that each block would define the characters used by a language or a set of languages. **This is also untrue.** The division into blocks exists, but it is almost completely accidental–an artifact of how the characters have been and still are allocated. Instead, there is a concept called *scripts*, which is more useful: there is `Latin` script, `Greek` script, and so on. Scripts usually span varied parts of several blocks. For further information see *Unicode::UCD*.

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 perhaps the most popular. UTF-8 is a variable length encoding that encodes Unicode characters as 1 to 6 bytes (only 4 with the currently defined characters). 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 *perlunicode*.

54.1.2 Perl's Unicode Support

Starting from Perl 5.6.0, Perl has had the capacity to handle Unicode natively. Perl 5.8.0, 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.

Starting from Perl 5.8.0, the use of `use utf8` is no longer necessary. In earlier releases the `utf8` pragma 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. 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*.

54.1.3 Perl's Unicode Model

Perl supports both pre-5.6 strings of eight-bit native bytes, and strings of Unicode characters. The 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.

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 `: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 *perlrun* for the documentation of the `-C` switch.

Note that this means that Perl expects other software to work, too: 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 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`.

54.1.4 Unicode and EBCDIC

Perl 5.8.0 also supports Unicode on EBCDIC platforms. There, Unicode support is somewhat more complex to implement since additional conversions are needed at every step. Some problems remain, see *perlebcdic* for details.

In any case, the Unicode support on EBCDIC platforms is better than in the 5.6 series, which didn't work much at all for EBCDIC platform. 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".

54.1.5 Creating Unicode

To create Unicode characters in literals for code points above `0xFF`, use the `\x{...}` notation in double-quoted strings:

```
    my $smiley = "\x{263a}";
```

Similarly, it can be used in regular expression literals

```
    $smiley =~ /\x{263a}/;
```

At run-time you can use `chr()`:

```
    my $hebrew_alef = chr(0x05d0);
```

See §54.1.15 for how to find all these numeric codes.

Naturally, `ord()` will do the reverse: it turns a character into a code point.

Note that `\x..` (no `{}` and only two hexadecimal digits), `\x{...}`, and `chr(...)` for arguments less than `0x100` (decimal 256) generate an eight-bit character for backward compatibility with older Perls. For arguments of `0x100` or more, Unicode characters are always produced. If you want to force the production of Unicode characters regardless of the numeric value, use `pack("U", ...)` instead of `\x..`, `\x{...}`, or `chr()`.

You can also use the `charnames` pragma to invoke characters by name in double-quoted strings:

```
    use charnames ':full';
    my $arabic_alef = "\N{ARABIC LETTER ALEF}";
```

And, as mentioned above, you can also `pack()` numbers into Unicode characters:

```
   my $georgian_an  = pack("U", 0x10a0);
```

Note that both `\x{...}` and `\N{...}` are compile-time string constants: you cannot use variables in them. if you want similar run-time functionality, use `chr()` and `charnames::vianame()`.

If you want to force the result to Unicode characters, use the special `"U0"` prefix. It consumes no arguments but forces the result to be in Unicode characters, instead of bytes.

```
   my $chars = pack("U0C*", 0x80, 0x42);
```

Likewise, you can force the result to be bytes by using the special `"C0"` prefix.

54.1.6 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 *perlunicode* and *perlretut*).

Note that Perl considers combining character sequences to be separate characters, so for example

```
    use charnames ':full';
    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 a combining character sequence.

Life is not quite so transparent, however, when working with legacy encodings, I/O, and certain special cases:

54.1.7 Legacy Encodings

When you combine legacy data and Unicode the legacy data needs to be upgraded to Unicode. Normally ISO 8859-1 (or EBCDIC, if applicable) is assumed. You can override this assumption by using the `encoding` pragma, for example

```
    use encoding 'latin2'; # ISO 8859-2
```

in which case literals (string or regular expressions), `chr()`, and `ord()` in your whole script are assumed to produce Unicode characters from ISO 8859-2 code points. Note that the matching for encoding names is forgiving: instead of `latin2` you could have said `Latin 2`, or `iso8859-2`, or other variations. With just

```
    use encoding;
```

the environment variable `PERL_ENCODING` will be consulted. If that variable isn't set, the encoding pragma will fail.

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
```

54.1.8 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.

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,'<: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 ':utf8'; # input and output default layer 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;
```

or you can also use the `':encoding(...)'` layer

```
    open(my $epic,'<:encoding(iso-8859-7)','iliad.greek');
    my $line_of_unicode = <$epic>;
```

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 binmode in *perlfunc*.

The `:locale` does not currently (as of Perl 5.8.0) 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, is similar to the `encoding` pragma in that it 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, ">: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 ':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 the new PerlIO feature (which is the default on most systems).

54.1.9 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("U*", $_[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.

54.1.10 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 or `unpack("H*", ...)` to display the bytes:

  ```
  # this prints  c4 80  for the UTF-8 bytes 0xc4 0x80
  print join(" ", unpack("H*", 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.

54.1.11 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 As should be considered equal, or even As of any case.

  The long answer is that you need to consider character normalization and casing issues: see *Unicode::Normalize*, Unicode Technical Reports #15 and #21, *Unicode Normalization Forms* and *Case Mappings*, http://www.unicode.org/unicode/reports/tr15/ and http://www.unicode.org/unicode/reports/tr21/

  As of Perl 5.8.0, the "Full" case-folding of *Case Mappings/SpecialCasing* is implemented.

- 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/

54.1.12 Miscellaneous

- Character Ranges and Classes

  Character ranges in regular expression character classes (`/[a-z]/`) and in the `tr///` (also known as `y///`) operator are not magically Unicode-aware. What this means that `[A-Za-z]` will not magically start to mean "all alphabetic letters"; not that it does mean that even for 8-bit characters, you should be using `/[[:alpha:]]/` in that case.

  For specifying character classes like that in regular expressions, you can use the various Unicode properties–`\pL`, or perhaps `\p{Alphabetic}`, in this particular case. You can use Unicode code points as the end points of character ranges, but there is no magic associated with specifying a certain range. For further information–there are dozens of Unicode character classes–see *perlunicode*.

- 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).

54.1.13 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.

- How Do I Know Whether My String Is In Unicode?

  You shouldn't care. No, you really shouldn't. No, really. If you have to care–beyond the cases described above–it means that we didn't get the transparency of Unicode quite right.

  Okay, if you insist:

  ```
  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 (multi-byte, variable-length) UTF-8 encoded code points of the characters. Bytes added to an

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, and 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 its only defined function `length()`:

```
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)
```

- 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 (decode_utf8($string_of_bytes_that_I_think_is_utf8)) {
    # valid
} else {
    # invalid
}
```

  For UTF-8 only, you can use:

```
use warnings;
@chars = unpack("U0U*", $string_of_bytes_that_I_think_is_utf8);
```

  If invalid, a `Malformed UTF-8 character (byte 0x##) in unpack` warning is produced. The "U0" means "expect strictly UTF-8 encoded Unicode". Without that the `unpack("U*", ...)` would accept also data like `chr(0xFF)`, similarly to the `pack` as we saw earlier.

- 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("C*", unpack("U*", $Unicode_string));
$Unicode_string = pack("U*", unpack("C*", $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);
```

You can convert well-formed UTF-8 to a sequence of bytes, but if you just want to convert random binary data into UTF-8, you can't. **Any random collection of bytes isn't well-formed UTF-8**. You can use `unpack("C*", $string)` for the former, and you can create well-formed Unicode data by `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?

  In Perl, not very well. Avoid using locales through the `locale` pragma. Use only one or the other. But see *perlrun* for the description of the `-C` switch and its environment counterpart, `$ENV{PERL_UNICODE}` to see how to enable various Unicode features, for example by using locale settings.

54.1.14 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
```

54.1.15 Further Resources

- Unicode Consortium

  ```
  http://www.unicode.org/
  ```

- Unicode FAQ

  ```
  http://www.unicode.org/unicode/faq/
  ```

- Unicode Glossary

  ```
  http://www.unicode.org/glossary/
  ```

- Unicode 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/
  ```

- The Unicode support files live within the Perl installation in the directory

  ```
  $Config{installprivlib}/unicore
  ```

  in Perl 5.8.0 or newer, and

  ```
  $Config{installprivlib}/unicode
  ```

  in the Perl 5.6 series. (The renaming to *lib/unicore* was done to avoid naming conflicts with lib/Unicode in case-insensitive filesystems.) The main Unicode data file is *UnicodeData.txt* (or *Unicode.301* in Perl 5.6.1.) You can find the `$Config{installprivlib}` by

  ```
  perl "-V:installprivlib"
  ```

  You can explore various information from the Unicode data files using the `Unicode::UCD` module.

54.2 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;
```

54.3 SEE ALSO

perlunicode, *Encode*, *encoding*, *open*, *utf8*, *bytes*, *perlretut*, *perlrun*, *Unicode::Collate*, *Unicode::Normalize*, *Unicode::UCD*

54.4 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.

54.5 AUTHOR, COPYRIGHT, AND LICENSE

Chapter 55

perlunicode

Unicode support in Perl

55.1 DESCRIPTION

55.1.1 Important Caveats

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.

Input and Output Layers

Perl knows when a filehandle uses Perl's internal Unicode encodings (UTF-8, or UTF-EBCDIC if in EBCDIC) if the filehandle is opened with the ":utf8" layer. Other encodings can be converted to Perl's encoding on input or from Perl's encoding on output by use of the ":encoding(...)" layer. See *open*.

To indicate that Perl source itself is using a particular encoding, see *encoding*.

Regular Expressions

The regular expression compiler produces polymorphic opcodes. That is, the pattern adapts to the data and automatically switches to the Unicode character scheme when presented with Unicode data–or instead uses a traditional byte scheme when presented with byte data.

`use utf8` still needed to enable UTF-8/UTF-EBCDIC in scripts

As a compatibility measure, the `use utf8` pragma must be explicitly included to enable recognition of UTF-8 in the Perl scripts themselves (in string or regular expression literals, or in identifier names) on ASCII-based machines or to recognize UTF-EBCDIC on EBCDIC-based machines. **These are the only times when an explicit `use utf8` is needed.** See *utf8*.

You can also use the `encoding` pragma to change the default encoding of the data in your script; see *encoding*.

BOM-marked scripts and UTF-16 scripts autodetected

If a Perl script begins marked 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 Unicode. (BOMless UTF-8 cannot be effectively recognized or differentiated from ISO 8859-1 or other eight-bit encodings.)

`use encoding` needed to upgrade non-Latin-1 byte strings

By default, there is a fundamental asymmetry in Perl's unicode model: implicit upgrading from byte strings to Unicode strings assumes that they were encoded in *ISO 8859-1 (Latin-1)*, but Unicode strings are downgraded with UTF-8 encoding. This happens because the first 256 codepoints in Unicode happens to agree with Latin-1.

If you wish to interpret byte strings as UTF-8 instead, use the `encoding` pragma:

```
use encoding 'utf8';
```

See §55.1.2 for more details.

55.1.2 Byte and Character Semantics

Beginning with version 5.6, Perl uses logically-wide characters to represent strings internally.

In future, Perl-level operations will be expected to work with characters rather than bytes.

However, as an interim compatibility measure, Perl aims to provide a safe migration path from byte semantics to character semantics for programs. For operations where Perl can unambiguously decide that the input data are characters, Perl switches to character semantics. For operations where this determination cannot be made without additional information from the user, Perl decides in favor of compatibility and chooses to use byte semantics.

This behavior preserves compatibility with earlier versions of Perl, which allowed byte semantics in Perl operations only if none of the program's inputs were marked as being as source of Unicode character data. Such data may come from filehandles, from calls to external programs, from information provided by the system (such as %ENV), or from literals and constants in the source text.

The `bytes` pragma will always, regardless of platform, force byte semantics in a particular lexical scope. See *bytes*.

The `utf8` pragma is primarily a compatibility device that enables recognition of UTF-(8|EBCDIC) in literals encountered by the parser. Note that this pragma is only required while Perl defaults to byte semantics; when character semantics become the default, this pragma may become a no-op. See *utf8*.

Unless explicitly stated, Perl operators use character semantics for Unicode data and byte semantics for non-Unicode data. The decision to use character semantics is made transparently. If input data comes from a Unicode source–for example, if a character encoding layer is added to a filehandle or a literal Unicode string constant appears in a program–character semantics apply. Otherwise, byte semantics are in effect. The `bytes` pragma should be used to force byte semantics on Unicode data.

If strings operating under byte semantics and strings with Unicode character data are concatenated, the new string will be created by decoding the byte strings as *ISO 8859-1 (Latin-1)*, even if the old Unicode string used EBCDIC. This translation is done without regard to the system's native 8-bit encoding. To change this for systems with non-Latin-1 and non-EBCDIC native encodings, use the `encoding` pragma. See *encoding*.

Under character semantics, many operations that formerly operated on bytes now operate on characters. A character in Perl is logically just a number ranging from 0 to 2**31 or so. Larger characters may encode into longer sequences of bytes internally, but this internal detail is mostly hidden for Perl code. See *perluniintro* for more.

55.1.3 Effects of Character Semantics

Character semantics have the following effects:

- Strings–including hash keys–and regular expression patterns may contain characters that have an ordinal value larger than 255.

  If you use a Unicode editor to edit your program, Unicode characters may occur directly within the literal strings in one of the various Unicode encodings (UTF-8, UTF-EBCDIC, UCS-2, etc.), but will be recognized as such and converted to Perl's internal representation only if the appropriate *encoding* is specified.

  Unicode characters can also be added to a string by using the `\x{...}` notation. The Unicode code for the desired character, in hexadecimal, should be placed in the braces. For instance, a smiley face is `\x{263A}`. This encoding scheme only works for characters with a code of 0x100 or above.

  Additionally, if you

  ```
  use charnames ':full';
  ```

  you can use the `\N{...}` notation and put the official Unicode character name within the braces, such as `\N{WHITE SMILING FACE}`.

- If an appropriate *encoding* is specified, identifiers within the Perl script may contain Unicode alphanumeric characters, including ideographs. Perl does not currently attempt to canonicalize variable names.

- Regular expressions match characters instead of bytes. "." matches a character instead of a byte. The `\C` pattern is provided to force a match a single byte–a `char` in C, hence `\C`.

- Character classes in regular expressions match characters instead of bytes and match against the character properties specified in the Unicode properties database. `\w` can be used to match a Japanese ideograph, for instance.

  (However, and as a limitation of the current implementation, using `\w` or `\W` *inside* a `[...]` character class will still match with byte semantics.)

- Named Unicode properties, scripts, and block ranges may be used like character classes via the `\p{}` "matches property" construct and the `\P{}` negation, "doesn't match property".

  For instance, `\p{Lu}` matches any character with the Unicode "Lu" (Letter, uppercase) property, while `\p{M}` matches any character with an "M" (mark–accents and such) property. Brackets are not required for single letter properties, so `\p{M}` is equivalent to `\pM`. Many predefined properties are available, such as `\p{Mirrored}` and `\p{Tibetan}`.

  The official Unicode script and block names have spaces and dashes as separators, but for convenience you can use dashes, spaces, or underbars, and case is unimportant. It is recommended, however, that for consistency you use the following naming: the official Unicode script, property, or block name (see below for the additional rules that apply to block names) with whitespace and dashes removed, and the words "uppercase-first-lowercase-rest". `Latin-1 Supplement` thus becomes `Latin1Supplement`.

  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}`.

  NOTE: the properties, scripts, and blocks listed here are as of Unicode 3.2.0, March 2002, or Perl 5.8.0, July 2002. Unicode 4.0.0 came out in April 2003, and Perl 5.8.1 in September 2003.

  Here are the basic Unicode General Category properties, followed by their long form. You can use either; `\p{Lu}` and `\p{UppercaseLetter}`, for instance, are identical.

```
    Short       Long

    L           Letter
    LC          CasedLetter
    Lu          UppercaseLetter
    Ll          LowercaseLetter
    Lt          TitlecaseLetter
    Lm          ModifierLetter
    Lo          OtherLetter

    M           Mark
    Mn          NonspacingMark
    Mc          SpacingMark
    Me          EnclosingMark

    N           Number
    Nd          DecimalNumber
    Nl          LetterNumber
    No          OtherNumber

    P           Punctuation
    Pc          ConnectorPunctuation
    Pd          DashPunctuation
    Ps          OpenPunctuation
    Pe          ClosePunctuation
    Pi          InitialPunctuation
                (may behave like Ps or Pe depending on usage)
    Pf          FinalPunctuation
                (may behave like Ps or Pe depending on usage)
    Po          OtherPunctuation
```

S	Symbol
Sm	MathSymbol
Sc	CurrencySymbol
Sk	ModifierSymbol
So	OtherSymbol
Z	Separator
Zs	SpaceSeparator
Zl	LineSeparator
Zp	ParagraphSeparator
C	Other
Cc	Control
Cf	Format
Cs	Surrogate (not usable)
Co	PrivateUse
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 cases, which are aliases for the set of `Ll`, `Lu`, and `Lt`.

Because Perl hides the need for the user to understand the internal representation of Unicode characters, there is no need to implement the somewhat messy concept of surrogates. `Cs` is therefore not supported.

Because scripts differ in their directionality–Hebrew is written right to left, for example–Unicode supplies these properties in the BidiClass class:

Property	Meaning
L	Left-to-Right
LRE	Left-to-Right Embedding
LRO	Left-to-Right Override
R	Right-to-Left
AL	Right-to-Left Arabic
RLE	Right-to-Left Embedding
RLO	Right-to-Left Override
PDF	Pop Directional Format
EN	European Number
ES	European Number Separator
ET	European Number Terminator
AN	Arabic Number
CS	Common Number Separator
NSM	Non-Spacing Mark
BN	Boundary Neutral
B	Paragraph Separator
S	Segment Separator
WS	Whitespace
ON	Other Neutrals

For example, `\p{BidiClass:R}` matches characters that are normally written right to left.

55.1.4 Scripts

The script names which can be used by `\p{...}` and `\P{...}`, such as in `\p{Latin}` or `\p{Cyrillic}`, are as follows:

```
    Arabic
    Armenian
    Bengali
    Bopomofo
    Buhid
    CanadianAboriginal
    Cherokee
    Cyrillic
    Deseret
    Devanagari
    Ethiopic
    Georgian
    Gothic
    Greek
    Gujarati
    Gurmukhi
    Han
    Hangul
    Hanunoo
    Hebrew
    Hiragana
    Inherited
    Kannada
    Katakana
    Khmer
    Lao
    Latin
    Malayalam
    Mongolian
    Myanmar
    Ogham
    OldItalic
    Oriya
    Runic
    Sinhala
    Syriac
    Tagalog
    Tagbanwa
    Tamil
    Telugu
    Thaana
    Thai
    Tibetan
    Yi
```

Extended property classes can supplement the basic properties, defined by the *PropList* Unicode database:

```
    ASCIIHexDigit
    BidiControl
    Dash
    Deprecated
    Diacritic
    Extender
    GraphemeLink
    HexDigit
    Hyphen
    Ideographic
    IDSBinaryOperator
    IDSTrinaryOperator
    JoinControl
    LogicalOrderException
    NoncharacterCodePoint
    OtherAlphabetic
    OtherDefaultIgnorableCodePoint
    OtherGraphemeExtend
    OtherLowercase
    OtherMath
    OtherUppercase
    QuotationMark
    Radical
    SoftDotted
    TerminalPunctuation
    UnifiedIdeograph
    WhiteSpace
```

and there are further derived properties:

```
    Alphabetic      Lu + Ll + Lt + Lm + Lo + OtherAlphabetic
    Lowercase       Ll + OtherLowercase
    Uppercase       Lu + OtherUppercase
    Math            Sm + OtherMath

    ID_Start        Lu + Ll + Lt + Lm + Lo + Nl
    ID_Continue     ID_Start + Mn + Mc + Nd + Pc

    Any             Any character
    Assigned        Any non-Cn character (i.e. synonym for \P{Cn})
    Unassigned      Synonym for \p{Cn}
    Common          Any character (or unassigned code point)
                    not explicitly assigned to a script
```

For backward compatibility (with Perl 5.6), all properties mentioned so far may have `Is` prepended to their name, so `\P{IsLu}`, for example, is equal to `\P{Lu}`.

55.1.5 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 256 Unicode characters. For example, the `Latin` script contains letters from many blocks but does not contain all the characters from those blocks. It does not, for example, contain digits, because digits are shared across many scripts. Digits and similar groups, like punctuation, are in a category called `Common`.

For more about scripts, see the UTR #24:

```
   http://www.unicode.org/unicode/reports/tr24/
```

For more about blocks, see:

```
   http://www.unicode.org/Public/UNIDATA/Blocks.txt
```

Block names are given with the `In` prefix. For example, the Katakana block is referenced via `\p{InKatakana}`. The `In` prefix may be omitted if there is no naming conflict with a script or any other property, but it is recommended that `In` always be used for block tests to avoid confusion.

These block names are supported:

```
    InAlphabeticPresentationForms
    InArabic
    InArabicPresentationFormsA
    InArabicPresentationFormsB
    InArmenian
    InArrows
    InBasicLatin
    InBengali
    InBlockElements
    InBopomofo
    InBopomofoExtended
    InBoxDrawing
    InBraillePatterns
    InBuhid
    InByzantineMusicalSymbols
    InCJKCompatibility
    InCJKCompatibilityForms
    InCJKCompatibilityIdeographs
    InCJKCompatibilityIdeographsSupplement
    InCJKRadicalsSupplement
    InCJKSymbolsAndPunctuation
    InCJKUnifiedIdeographs
    InCJKUnifiedIdeographsExtensionA
    InCJKUnifiedIdeographsExtensionB
    InCherokee
    InCombiningDiacriticalMarks
    InCombiningDiacriticalMarksforSymbols
    InCombiningHalfMarks
    InControlPictures
    InCurrencySymbols
    InCyrillic
    InCyrillicSupplementary
    InDeseret
    InDevanagari
    InDingbats
    InEnclosedAlphanumerics
    InEnclosedCJKLettersAndMonths
    InEthiopic
    InGeneralPunctuation
    InGeometricShapes
    InGeorgian
    InGothic
    InGreekExtended
    InGreekAndCoptic
    InGujarati
    InGurmukhi
    InHalfwidthAndFullwidthForms
    InHangulCompatibilityJamo
    InHangulJamo
    InHangulSyllables
    InHanunoo
```

```
InHebrew
InHighPrivateUseSurrogates
InHighSurrogates
InHiragana
InIPAExtensions
InIdeographicDescriptionCharacters
InKanbun
InKangxiRadicals
InKannada
InKatakana
InKatakanaPhoneticExtensions
InKhmer
InLao
InLatin1Supplement
InLatinExtendedA
InLatinExtendedAdditional
InLatinExtendedB
InLetterlikeSymbols
InLowSurrogates
InMalayalam
InMathematicalAlphanumericSymbols
InMathematicalOperators
InMiscellaneousMathematicalSymbolsA
InMiscellaneousMathematicalSymbolsB
InMiscellaneousSymbols
InMiscellaneousTechnical
InMongolian
InMusicalSymbols
InMyanmar
InNumberForms
InOgham
InOldItalic
InOpticalCharacterRecognition
InOriya
InPrivateUseArea
InRunic
InSinhala
InSmallFormVariants
InSpacingModifierLetters
InSpecials
InSuperscriptsAndSubscripts
InSupplementalArrowsA
InSupplementalArrowsB
InSupplementalMathematicalOperators
InSupplementaryPrivateUseAreaA
InSupplementaryPrivateUseAreaB
InSyriac
InTagalog
InTagbanwa
InTags
InTamil
InTelugu
InThaana
InThai
InTibetan
InUnifiedCanadianAboriginalSyllabics
```

```
InVariationSelectors
InYiRadicals
InYiSyllables
```

- The special pattern `\X` matches any extended Unicode sequence–"a combining character sequence" in Standardese–where the first character is a base character and subsequent characters are mark characters that apply to the base character. `\X` is equivalent to `(?:\PM\pM*)`.
- The `tr///` operator translates characters instead of bytes. Note that the `tr///CU` functionality has been removed. For similar functionality see pack('U0', ...) and pack('C0', ...).
- Case translation operators use the Unicode case translation tables when character input is provided. 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.
- Most operators that deal with positions or lengths in a string will automatically switch to using character positions, including `chop()`, `chomp()`, `substr()`, `pos()`, `index()`, `rindex()`, `sprintf()`, `write()`, and `length()`. Operators that specifically do not switch include `vec()`, `pack()`, and `unpack()`. Operators that really don't care include operators that treats strings as a bucket of bits such as `sort()`, and operators dealing with filenames.
- The `pack()`/`unpack()` letters `c` and `C` do *not* change, since they are often used for byte-oriented formats. Again, think `char` in the C language.

  There is a new `U` specifier that converts between Unicode characters and code points.
- The `chr()` and `ord()` functions work on characters, similar to `pack("U")` and `unpack("U")`, *not* `pack("C")` and `unpack("C")`. `pack("C")` and `unpack("C")` are methods for emulating byte-oriented `chr()` and `ord()` on Unicode strings. While these methods reveal the internal encoding of Unicode strings, that is not something one normally needs to care about at all.
- The bit string operators, `& | ^ ~`, can operate on character data. However, for backward compatibility, such as when using bit string operations when characters are all less than 256 in ordinal value, one should not use `~` (the bit complement) with characters of both values less than 256 and values greater than 256. Most importantly, DeMorgan's laws (`~($x|$y) eq ~$x&~$y` and `~($x&$y) eq ~$x|~$y`) will not hold. The reason for this mathematical *faux pas* is that the complement cannot return **both** the 8-bit (byte-wide) bit complement **and** the full character-wide bit complement.
- lc(), uc(), lcfirst(), and ucfirst() work for the following cases:
  - the case mapping is from a single Unicode character to another single Unicode character, or
  - the case mapping is from a single Unicode character to more than one Unicode character.

  Things to do with locales (Lithuanian, Turkish, Azeri) do **not** work since Perl does not understand the concept of Unicode locales.

  See the Unicode Technical Report #21, Case Mappings, for more details.
- And finally, `scalar reverse()` reverses by character rather than by byte.

55.1.6 User-Defined Character Properties

You can define your own character properties by defining subroutines whose names begin with "In" or "Is". 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 IsForeign 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.

The subroutines must return a specially-formatted string, with one or more newline-separated lines. Each line must be one of the following:

- Two hexadecimal numbers separated by horizontal whitespace (space or tabular characters) denoting a range of Unicode code points to include.
- Something to include, prefixed by "+": a built-in character property (prefixed by "utf8::") or a 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 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 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 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 non-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
    }
```

Intersection is useful for getting the common characters matched by two (or more) classes.

```
    sub InFooAndBar {
        return <<'END';
    +main::Foo
    &main::Bar
    END
    }
```

It's important to remember not to use "&" for the first set – that would be intersecting with nothing (resulting in an empty set).

You can also define your own mappings to be used in the lc(), lcfirst(), uc(), and ucfirst() (or their string-inlined versions). The principle is the same: define subroutines in the `main` package with names like `ToLower` (for lc() and lcfirst()), `ToTitle` (for the first character in ucfirst()), and `ToUpper` (for uc(), and the rest of the characters in ucfirst()).

The string returned by the subroutines needs now to be three hexadecimal numbers separated by tabulators: start of the source range, end of the source range, and start of the destination range. For example:

```
    sub ToUpper {
        return <<END;
    0061\t0063\t0041
    END
    }
```

defines an uc() mapping that causes only the characters "a", "b", and "c" to be mapped to "A", "B", "C", all other characters will remain unchanged.

If there is no source range to speak of, that is, the mapping is from a single character to another single character, leave the end of the source range empty, but the two tabulator characters are still needed. For example:

```
    sub ToLower {
        return <<END;
    0041\t\t0061
    END
    }
```

defines a lc() mapping that causes only "A" to be mapped to "a", all other characters will remain unchanged.

(For serious hackers only) If you want to introspect the default mappings, you can find the data in the directory `$Config{privlib}`*/unicore/To/*. The mapping data is returned as the here-document, and the `utf8::ToSpecFoo` are special exception mappings derived from <$Config{privlib}>*/unicore/SpecialCasing.txt*. The `Digit` and `Fold` mappings that one can see in the directory are not directly user-accessible, one can use either the `Unicode::UCD` module, or just match case-insensitively (that's when the `Fold` mapping is used).

A final note on the user-defined property tests and mappings: they will be used only if the scalar has been marked as having Unicode characters. Old byte-style strings will not be affected.

55.1.7 Character Encodings for Input and Output

See *Encode*.

55.1.8 Unicode Regular Expression Support Level

The following list of Unicode support for regular expressions describes all the features currently supported. The references to "Level N" and the section numbers refer to the Unicode Technical Report 18, "Unicode Regular Expression Guidelines", version 6 (Unicode 3.2.0, Perl 5.8.0).

- Level 1 - Basic Unicode Support

```
        2.1 Hex Notation                        - done          [1]
            Named Notation                      - done          [2]
        2.2 Categories                          - done          [3][4]
        2.3 Subtraction                         - MISSING       [5][6]
        2.4 Simple Word Boundaries              - done          [7]
        2.5 Simple Loose Matches                - done          [8]
        2.6 End of Line                         - MISSING       [9][10]

        [ 1] \x{...}
        [ 2] \N{...}
        [ 3] . \p{...} \P{...}
        [ 4] support for scripts (see UTR#24 Script Names), blocks,
             binary properties, enumerated non-binary properties, and
             numeric properties (as listed in UTR#18 Other Properties)
        [ 5] have negation
        [ 6] can use regular expression look-ahead [a]
             or user-defined character properties [b] to emulate subtraction
        [ 7] include Letters in word characters
        [ 8] note that Perl does Full case-folding in matching, not Simple:
             for example U+1F88 is equivalent with U+1F00 U+03B9,
             not with 1F80.  This difference matters 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.
        [ 9] see UTR #13 Unicode Newline Guidelines
        [10] should do ^ and $ also on \x{85}, \x{2028} and \x{2029}
             (should also affect <>, $., and script line numbers)
             (the \x{85}, \x{2028} and \x{2029} do match \s)
```

[a] You can mimic class subtraction using lookahead. For example, what UTR #18 might write as

```
    [{Greek}-[{UNASSIGNED}]]
```

in Perl can be written as:

```
    (?!\p{Unassigned})\p{InGreekAndCoptic}
    (?=\p{Assigned})\p{InGreekAndCoptic}
```

But in this particular example, you probably really want

```
    \p{GreekAndCoptic}
```

which will match assigned characters known to be part of the Greek script.

Also see the Unicode::Regex::Set module, it does implement the full UTR #18 grouping, intersection, union, and removal (subtraction) syntax.

[b] See §55.1.6.

- Level 2 - Extended Unicode Support

```
3.1 Surrogates                          - MISSING       [11]
3.2 Canonical Equivalents               - MISSING       [12][13]
3.3 Locale-Independent Graphemes        - MISSING       [14]
3.4 Locale-Independent Words            - MISSING       [15]
3.5 Locale-Independent Loose Matches    - MISSING       [16]

[11] Surrogates are solely a UTF-16 concept and Perl's internal
     representation is UTF-8.  The Encode module does UTF-16, though.
[12] see UTR#15 Unicode Normalization
[13] have Unicode::Normalize but not integrated to regexes
[14] have \X but at this level . should equal that
[15] need three classes, not just \w and \W
[16] see UTR#21 Case Mappings
```

- Level 3 - Locale-Sensitive Support

```
4.1 Locale-Dependent Categories         - MISSING
4.2 Locale-Dependent Graphemes          - MISSING       [16][17]
4.3 Locale-Dependent Words              - MISSING
4.4 Locale-Dependent Loose Matches      - MISSING
4.5 Locale-Dependent Ranges             - MISSING

[16] see UTR#10 Unicode Collation Algorithms
[17] have Unicode::Collate but not integrated to regexes
```

55.1.9 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 6 bytes, current character allocations require 4 bytes), byte-order independent encoding. For ASCII (and we really do mean 7-bit ASCII, not another 8-bit encoding), UTF-8 is transparent.

  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       ******* ill-formed *******
   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 `A0..BF` in `U+0800..U+0FFF`, the `80..9F` in `U+D000...U+D7FF`, the `90..BF` in `U+10000..U+3FFFF`, and the `80...8F` in `U+100000..U+10FFFF`. The "gaps" 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. So that's 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 the are in the encoded character.

- UTF-EBCDIC

  Like UTF-8 but EBCDIC-safe, in the way that UTF-8 is ASCII-safe.

- UTF-16, UTF-16BE, UTF-16LE, Surrogates, and BOMs (Byte Order Marks)

  The followings items are mostly for reference and general Unicode knowledge, Perl doesn't use these constructs internally.

  UTF-16 is a 2 or 4 byte encoding. The Unicode code points `U+0000..U+FFFF` are stored in a single 16-bit unit, and the 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);
```

  If you try to generate surrogates (for example by using chr()), you will get a warning if warnings are turned on, because those code points are not valid for a Unicode character.

  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 BOMs, 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 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 guaranteed not to be a valid Unicode character, 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".

- UTF-32, UTF-32BE, UTF-32LE

  The UTF-32 family is pretty much like the UTF-16 family, expect that the units are 32-bit, and therefore the surrogate scheme is not needed. The BOM signatures will be `0x00 0x00 0xFE 0xFF` for BE and `0xFF 0xFE 0x00 0x00` for LE.

- UCS-2, UCS-4

  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.

- 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.

55.1.10 Security Implications of Unicode

- Malformed UTF-8

  Unfortunately, the 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 real Unicode code points.

- Regular expressions behave slightly differently between byte data and character (Unicode) data. For example, the "word character" character class `\w` will work differently depending on if data is eight-bit bytes or Unicode.

  In the first case, the set of `\w` characters is either small–the default set of alphabetic characters, digits, and the "_"–or, if you are using a locale (see *perllocale*), the `\w` might contain a few more letters according to your language and country.

  In the second case, the `\w` set of characters is much, much larger. Most importantly, even in the set of the first 256 characters, it will probably match different characters: unlike most locales, which are specific to a language and country pair, Unicode classifies all the characters that are letters *somewhere* 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.

  As discussed elsewhere, Perl has one foot (two hooves?) planted in each of two worlds: the old world of bytes and the new world of characters, upgrading from bytes to characters when necessary. If your legacy code does not explicitly use Unicode, no automatic switch-over to characters should happen. Characters shouldn't get downgraded to bytes, either. It is possible to accidentally mix bytes and characters, however (see *perluniintro*), in which case `\w` in regular expressions might start behaving differently. Review your code. Use warnings and the `strict` pragma.

55.1.11 Unicode in Perl on EBCDIC

The way Unicode is handled on EBCDIC platforms is still experimental. On such platforms, references to UTF-8 encoding in this document and elsewhere should be read as meaning the UTF-EBCDIC specified in Unicode Technical Report 16, unless ASCII vs. EBCDIC issues are specifically discussed. There is no `utfebcdic` pragma or ":utfebcdic" layer; rather, "utf8" and ":utf8" are reused to mean the platform's "natural" 8-bit encoding of Unicode. See *perlebcdic* for more discussion of the issues.

55.1.12 Locales

Usually locale settings and Unicode do not affect each other, but there are a couple of exceptions:

- 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 *perlrun* for the documentation of the `-C` switch.

- Perl tries really hard to work both with Unicode and the old byte-oriented world. Most often this is nice, but sometimes Perl's straddling of the proverbial fence causes problems.

55.1.13 When Unicode Does Not Happen

While Perl does have extensive ways to input and output in Unicode, and few other 'entry points' like the @ARGV which can be interpreted as Unicode (UTF-8), there still are many places where Unicode (in some encoding or another) could be given as arguments or received as results, or both, but it is not.

The following are such interfaces. For all of these interfaces Perl currently (as of 5.8.3) simply assumes byte strings both as arguments and results, or UTF-8 strings if the `encoding` pragma has been used.

One reason why Perl does not attempt to resolve the role of Unicode in this cases 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 the qx and system: how well will the 'command line interface' (and which of them?) handle Unicode?

- chmod, 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

55.1.14 Forcing Unicode in Perl (Or Unforcing Unicode in Perl)

Sometimes (see §55.1.13) there are situations where you simply need to force Perl to believe that a byte string is UTF-8, or vice versa. The low-level calls utf8::upgrade($bytestring) and utf8::downgrade($utf8string) are the answers.

Do not use them without careful thought, though: Perl may easily get very confused, angry, or even crash, if you suddenly change the 'nature' of scalar like that. Especially careful you have to be if you use the utf8::upgrade(): any random byte string is not valid UTF-8.

55.1.15 Using Unicode in XS

If you want to handle Perl Unicode in XS extensions, you may find the following C APIs useful. See also Unicode Support in *perlguts* for an explanation about Unicode at the XS level, and *perlapi* for the API details.

- `DO_UTF8(sv)` returns true if the `UTF8` flag is on and the bytes pragma is not in effect. `SvUTF8(sv)` returns true is the `UTF8` flag is on; the bytes pragma is ignored. The `UTF8` flag being on does **not** mean that there are any characters of code points greater than 255 (or 127) in the scalar or that there are even any characters in the scalar. What the `UTF8` flag means is that the sequence of octets in the representation of the scalar is the sequence of UTF-8 encoded code points of the characters of a string. The `UTF8` flag being off means that each octet in this representation encodes a single character with code point 0..255 within the string. Perl's Unicode model is not to use UTF-8 until it is absolutely necessary.
- `uvuni_to_utf8(buf, chr)` writes a Unicode character code point into a buffer encoding the code point as UTF-8, and returns a pointer pointing after the UTF-8 bytes.
- `utf8_to_uvuni(buf, lenp)` reads UTF-8 encoded bytes from a buffer and returns the Unicode character code point and, optionally, the length of the UTF-8 byte sequence.
- `utf8_length(start, end)` returns the length of the UTF-8 encoded buffer in characters. `sv_len_utf8(sv)` returns the length of the UTF-8 encoded scalar.

- `sv_utf8_upgrade(sv)` converts the string of the scalar to its UTF-8 encoded form. `sv_utf8_downgrade(sv)` does the opposite, if possible. `sv_utf8_encode(sv)` is like sv_utf8_upgrade except that it does not set the `UTF8` flag. `sv_utf8_decode()` does the opposite of `sv_utf8_encode()`. Note that none of these are to be used as general-purpose encoding or decoding interfaces: `use Encode` for that. `sv_utf8_upgrade()` is affected by the encoding pragma but `sv_utf8_downgrade()` is not (since the encoding pragma is designed to be a one-way street).
- `is_utf8_char(s)` returns true if the pointer points to a valid UTF-8 character.
- `is_utf8_string(buf, len)` returns true if `len` bytes of the buffer are valid UTF-8.
- `UTF8SKIP(buf)` will return the number of bytes in the UTF-8 encoded character in the buffer. `UNISKIP(chr)` will return the number of bytes required to UTF-8-encode the Unicode character code point. `UTF8SKIP()` is useful for example for iterating over the characters of a UTF-8 encoded buffer; `UNISKIP()` is useful, for example, in computing the size required for a UTF-8 encoded buffer.
- `utf8_distance(a, b)` will tell the distance in characters between the two pointers pointing to the same UTF-8 encoded buffer.
- `utf8_hop(s, off)` will return a pointer to an UTF-8 encoded buffer that is `off` (positive or negative) Unicode characters displaced from the UTF-8 buffer `s`. Be careful not to overstep the buffer: `utf8_hop()` will merrily run off the end or the beginning of the buffer if told to do so.
- `pv_uni_display(dsv, spv, len, pvlim, flags)` and `sv_uni_display(dsv, ssv, pvlim, flags)` are useful for debugging the output of Unicode strings and scalars. By default they are useful only for debugging–they display **all** characters as hexadecimal code points–but with the flags `UNI_DISPLAY_ISPRINT`, `UNI_DISPLAY_BACKSLASH`, and `UNI_DISPLAY_QQ` you can make the output more readable.
- `ibcmp_utf8(s1, pe1, u1, l1, u1, s2, pe2, l2, u2)` can be used to compare two strings case-insensitively in Unicode. For case-sensitive comparisons you can just use `memEQ()` and `memNE()` as usual.

For more information, see *perlapi*, and *utf8.c* and *utf8.h* in the Perl source code distribution.

55.2 BUGS

55.2.1 Interaction with Locales

Use of locales with Unicode data may lead to odd results. Currently, Perl attempts to attach 8-bit locale info to characters in the range 0..255, but this technique is demonstrably incorrect for locales that use characters above that range when mapped into Unicode. Perl's Unicode support will also tend to run slower. Use of locales with Unicode is discouraged.

55.2.2 Interaction with Extensions

When Perl exchanges data with an extension, the extension should be able to understand the UTF-8 flag and act accordingly. If the extension doesn't know about the 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 them, you will be in a position to use the otherwise dangerous Encode::_utf8_on() 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.

55.2.3 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 will hopefully make the slowness somewhat less spectacular, at least for some operations. 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 `\d` (then again, there 268 Unicode characters matching `Nd` compared with the 10 ASCII characters matching `d`).

55.2.4 Porting code from perl-5.6.X

Perl 5.8 has 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 ($] > 5.007) {
    binmode $fh, ":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 UTF-8 flag is stripped off. Note that at the time of this writing (October 2002) the mentioned modules are not UTF-8-aware. Please check the documentation to verify if this is still true.

  ```
  if ($] > 5.007) {
    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 UTF-8 flag restored:

  ```
  if ($] > 5.007) {
    require Encode;
    $val = Encode::decode_utf8($val);
  }
  ```

- Same thing, if you are really sure it is UTF-8

  ```
  if ($] > 5.007) {
    require Encode;
    Encode::_utf8_on($val);
  }
  ```

- A wrapper for 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 (October 2002), the DBI has no standardized way to deal with UTF-8 data. Please check the documentation to verify if that is still true.

  ```
  sub fetchrow {
    my($self, $sth, $what) = @_; # $what is one of fetchrow_{array,hashref}
    if ($] < 5.007) {
      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 UTF-8 flag:

  ```
  utf8::downgrade($val) if $] > 5.007;
  ```

55.3 SEE ALSO

perluniintro, *encoding*, *Encode*, *open*, *utf8*, *bytes*, *perlretut*, ${^UNICODE} in *perlvar*

Chapter 56

perlsec

Perl security

56.1 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.

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 *perllocale*), 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.

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 Perl releases older than 5.6.0 the <*.c> and glob('*.c') would
    # have used an external program to do the filename expansion; but in
    # either case the result is 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}".

56.1.1 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 {
    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., that you knew what you were doing when you wrote the 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 SECURITY in *perllocale* for further discussion and examples.

56.1.2 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.)

56.1.3 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 *perlrun*. 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 later will not.

56.1.4 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 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 '-no_match_vars';
        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, 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.

56.1.5 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. If the latter is true, Perl can emulate the setuid and setgid mechanism when it notices the otherwise useless setuid/gid bits on Perl scripts. It does this via a special executable called **suidperl** that is automatically invoked for you if it's needed.

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.

Prior to release 5.6.1 of Perl, bugs in the code of **suidperl** could introduce a security hole.

56.1.6 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 licence 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 licence's wording will stand up in court.

56.1.7 Unicode

Unicode is a new and complex technology and one may easily overlook certain security pitfalls. See *perluniintro* for an overview and *perlunicode* for details, and Security Implications of Unicode in *perlunicode* for security implications in particular.

56.1.8 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 Function - the algorithm used to "order" hash elements has been changed several times during the development of Perl, mainly to be reasonably fast. In Perl 5.8.1 also the security aspect was taken into account.

  In Perls before 5.8.1 one could rather easily generate data that as hash keys would cause Perl to consume large amounts of time because internal structure of hashes would badly degenerate. In Perl 5.8.1 the hash function is randomly perturbed by a pseudorandom seed which makes generating such naughty hash keys harder. See PERL_HASH_SEED in *perlrun* for more information.

  The random perturbation is done by default but if one wants for some reason emulate the old behaviour one can set the environment variable PERL_HASH_SEED to zero (or any other integer). One possible reason for wanting to emulate the old behaviour is that in the new behaviour consecutive runs of Perl will order hash keys differently, which may confuse some applications (like Data::Dumper: the outputs of two different runs are no more identical).

  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.

  Also note that while the order of the hash elements might be randomised, this "pseudoordering" 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.

- Regular expressions - Perl's regular expression engine is so called NFA (Non-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. Nothing more is required than resorting a list already sorted. Starting from Perl 5.8.0 a different sorting algorithm, mergesort, is used. Mergesort is insensitive to its input data, so it cannot be similarly fooled.

See http://www.cs.rice.edu/˜scrosby/hash/ for more information, and any computer science text book on the algorithmic complexity.

56.2 SEE ALSO

perlrun for its description of cleaning up environment variables.

Chapter 57

perlmod

Perl modules (packages and symbol tables)

57.1 DESCRIPTION

57.1.1 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 Technical Note on the Syntax of Variable Names in *perlvar*.

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 *perldebug*.

The special symbol __PACKAGE__ contains the current package, but cannot (easily) be used to construct variable names.

See *perlsub* for other scoping issues related to my() and local(), and *perlref* regarding closures.

57.1.2 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. In fact, the following have the same effect, though the first is more efficient because it does the symbol table lookups at compile time:

```
    local *main::foo    = *main::bar;
    local $main::{foo}  = $main::{bar};
```

(Be sure to note the **vast** difference between the second line above and local $main::foo = $main::bar. The former is accessing the hash %main::, which is the symbol table of package main. The latter is simply assigning scalar $bar in package main to scalar $foo of the same package.)

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.

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 *perlsub* 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 *perlref*.

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 *perlobj*).

57.1.3 BEGIN, CHECK, INIT and END

Four specially named code blocks are executed at the beginning and at the end of a running Perl program. These are the `BEGIN`, `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.

It should be noted that `BEGIN` code blocks **are** executed inside string `eval()`'s. The `CHECK` and `INIT` code blocks are **not** executed inside a string eval, which e.g. can be a problem in a mod_perl environment.

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 polymorphing 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).

CHECK and INIT code blocks are useful to catch the transition between the compilation phase and the execution phase of the main program.

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.

INIT blocks are run just before the Perl runtime begins execution, in "first in, first out" (FIFO) order. For example, the code generators documented in *perlcc* make use of INIT blocks to initialize and resolve pointers to XSUBs.

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         " 8. Ordinary code runs at runtime.\n";

END { print   "14.   So this is the end of the tale.\n" }
INIT { print  " 5. INIT blocks run FIFO just before runtime.\n" }
CHECK { print " 4.   So this is the fourth line.\n" }

print         " 9.   It runs in order, of course.\n";

BEGIN { print " 1. BEGIN blocks run FIFO during compilation.\n" }
END { print   "13.   Read perlmod for the rest of the story.\n" }
CHECK { print " 3. CHECK blocks run LIFO at compilation's end.\n" }
INIT { print  " 6.   Run this again, using Perl's -c switch.\n" }

print         "10.   This is anti-obfuscated code.\n";

END { print   "12. END blocks run LIFO at quitting time.\n" }
BEGIN { print " 2.   So this line comes out second.\n" }
INIT { print  " 7.   You'll see the difference right away.\n" }

print         "11.   It merely _looks_ like it should be confusing.\n";

__END__
```

57.1.4 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 *perltoot* and *perlobj*.

57.1.5 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 {
    use Exporter   ();
    our ($VERSION, @ISA, @EXPORT, @EXPORT_OK, %EXPORT_TAGS);

    # set the version for version checking
    $VERSION     = 1.00;
    # if using RCS/CVS, this may be preferred
    $VERSION = sprintf "%d.%03d", q$Revision: 1.1 $ =~ /(\d+)/g;

    @ISA         = qw(Exporter);
    @EXPORT      = qw(&func1 &func2 &func4);
    %EXPORT_TAGS = ( );     # eg: TAG => [ qw!name1 name2! ],

    # your exported package globals go here,
    # as well as any optionally exported functions
    @EXPORT_OK   = qw($Var1 %Hashit &func3);
}
our @EXPORT_OK;

# exported package globals go here
our $Var1;
our %Hashit;

# non-exported package globals go here
our @more;
our $stuff;

# initialize package globals, first exported ones
$Var1   = '';
%Hashit = ();

# then the others (which are still accessible as $Some::Module::stuff)
$stuff  = '';
@more   = ();

# all file-scoped lexicals must be created before
# the functions below that use them.

# file-private lexicals go here
my $priv_var    = '';
my %secret_hash = ();
```

```
    # here's a file-private function as a closure,
    # callable as &$priv_func;  it cannot be prototyped.
    my $priv_func = sub {
        # stuff goes here.
    };

    # make all your functions, whether exported or not;
    # remember to put something interesting in the {} stubs
    sub func1      {}    # no prototype
    sub func2()    {}    # proto'd void
    sub func3($$)  {}    # proto'd to 2 scalars

    # this one isn't exported, but could be called!
    sub func4(\%)  {}    # proto'd to 1 hash ref

    END { }       # module clean-up code here (global destructor)

    ## YOUR CODE GOES HERE

    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; import Module; }
```

or

```
    BEGIN { require Module; import Module LIST; }
```

As a special case

```
    use Module ();
```

is exactly equivalent to

```
    BEGIN { require Module; }
```

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.

57.1.6 Making your module threadsafe

Since 5.6.0, Perl has had support for a new 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 5.7.2 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().

57.2 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, *perltoot* and *perltooc* for an in-depth tutorial on creating classes, *perlobj* for a hard-core reference document on objects, *perlsub* for an explanation of functions and scoping, and *perlxstut* and *perlguts* for more information on writing extension modules.

Chapter 58

perlmodlib

Constructing new Perl modules and finding existing ones

58.1 THE PERL MODULE LIBRARY

Many modules are included in the Perl distribution. These are described below, and all end in *.pm*. You may discover compiled library files (usually ending in *.so*) or small pieces of modules to be autoloaded (ending in *.al*); these were automatically generated by the installation process. You may also discover files in the library directory that end in either *.pl* or *.ph*. These are old libraries supplied so that old programs that use them still run. The *.pl* files will all eventually be converted into standard modules, and the *.ph* files made by **h2ph** will probably end up as extension modules made by **h2xs**. (Some *.ph* values may already be available through the POSIX, Errno, or Fcntl modules.) The **pl2pm** file in the distribution may help in your conversion, but it's just a mechanical process and therefore far from bulletproof.

58.1.1 Pragmatic Modules

They work somewhat like compiler directives (pragmata) in that they tend to affect the compilation of your program, and thus will usually work well only when used within a `use`, or `no`. Most of these are lexically scoped, so an inner BLOCK may countermand them by saying:

```
    no integer;
    no strict 'refs';
    no warnings;
```

which lasts until the end of that BLOCK.

Some pragmas are lexically scoped–typically those that affect the `$^H` hints variable. Others affect the current package instead, like `use vars` and `use subs`, which allow you to predeclare a variables or subroutines within a particular *file* rather than just a block. Such declarations are effective for the entire file for which they were declared. You cannot rescind them with `no vars` or `no subs`.

The following pragmas are defined (and have their own documentation).

attributes

Get/set subroutine or variable attributes

attrs

Set/get attributes of a subroutine (deprecated)

autouse

Postpone load of modules until a function is used

base

Establish IS-A relationship with base class at compile time

bigint

Transparent BigInteger support for Perl

bignum

Transparent BigNumber support for Perl

bigrat

Transparent BigNumber/BigRational support for Perl

blib

Use MakeMaker's uninstalled version of a package

bytes

Force byte semantics rather than character semantics

charnames

Define character names for `\N{named}` string literal escapes

constant

Declare constants

diagnostics

Produce verbose warning diagnostics

encoding

Allows you to write your script in non-ascii or non-utf8

fields

Compile-time class fields

filetest

Control the filetest permission operators

if

`use` a Perl module if a condition holds

integer

Use integer arithmetic instead of floating point

less

Request less of something from the compiler

lib

Manipulate @INC at compile time

locale

Use and avoid POSIX locales for built-in operations

open

Set default PerlIO layers for input and output

ops

Restrict unsafe operations when compiling

overload

Package for overloading perl operations

re

Alter regular expression behaviour

sigtrap

Enable simple signal handling

sort

Control sort() behaviour

strict

Restrict unsafe constructs

subs

Predeclare sub names

threads

Perl extension allowing use of interpreter based threads from perl

threads::shared

Perl extension for sharing data structures between threads

utf8

Enable/disable UTF-8 (or UTF-EBCDIC) in source code

vars

Predeclare global variable names (obsolete)

vmsish

Control VMS-specific language features

warnings

Control optional warnings

warnings::register

Warnings import function

58.1.2 Standard Modules

Standard, bundled modules are all expected to behave in a well-defined manner with respect to namespace pollution because they use the Exporter module. See their own documentation for details.

It's possible that not all modules listed below are installed on your system. For example, the GDBM_File module will not be installed if you don't have the gdbm library.

AnyDBM_File

Provide framework for multiple DBMs

Attribute::Handlers

Simpler definition of attribute handlers

AutoLoader

Load subroutines only on demand

AutoSplit

Split a package for autoloading

B

The Perl Compiler

B::Asmdata

Autogenerated data about Perl ops, used to generate bytecode

B::Assembler

Assemble Perl bytecode

B::Bblock

Walk basic blocks

B::Bytecode

Perl compiler's bytecode backend

B::C

Perl compiler's C backend

B::CC

Perl compiler's optimized C translation backend

B::Concise

Walk Perl syntax tree, printing concise info about ops

B::Debug

Walk Perl syntax tree, printing debug info about ops

B::Deparse

Perl compiler backend to produce perl code

B::Disassembler

Disassemble Perl bytecode

B::Lint

Perl lint

B::Showlex

Show lexical variables used in functions or files

B::Stackobj

Helper module for CC backend

B::Stash

Show what stashes are loaded

B::Terse

Walk Perl syntax tree, printing terse info about ops

B::Xref

Generates cross reference reports for Perl programs

Benchmark

Benchmark running times of Perl code

ByteLoader

Load byte compiled perl code

CGI

Simple Common Gateway Interface Class

CGI::Apache

Backward compatibility module for CGI.pm

CGI::Carp

CGI routines for writing to the HTTPD (or other) error log

CGI::Cookie

Interface to Netscape Cookies

CGI::Fast

CGI Interface for Fast CGI

CGI::Pretty

Module to produce nicely formatted HTML code

CGI::Push

Simple Interface to Server Push

CGI::Switch

Backward compatibility module for defunct CGI::Switch

CGI::Util

Internal utilities used by CGI module

CPAN

Query, download and build perl modules from CPAN sites

CPAN::FirstTime

Utility for CPAN::Config file Initialization

CPAN::Nox

Wrapper around CPAN.pm without using any XS module

Carp

Warn of errors (from perspective of caller)

Carp::Heavy

No user serviceable parts inside

Class::ISA

Report the search path for a class's ISA tree

Class::Struct

Declare struct-like datatypes as Perl classes

Config

Access Perl configuration information

Cwd

Get pathname of current working directory

DB

Programmatic interface to the Perl debugging API (draft, subject to

DB_File

Perl5 access to Berkeley DB version 1.x

Data::Dumper

Stringified perl data structures, suitable for both printing and `eval`

Devel::DProf

A Perl code profiler

Devel::PPPort

Perl/Pollution/Portability

Devel::Peek

A data debugging tool for the XS programmer

Devel::SelfStubber

Generate stubs for a SelfLoading module

Digest

Modules that calculate message digests

Digest::MD5

Perl interface to the MD5 Algorithm

Digest::base

Digest base class

DirHandle

Supply object methods for directory handles

Dumpvalue

Provides screen dump of Perl data.

DynaLoader

Dynamically load C libraries into Perl code

Encode

Character encodings

Encode::Alias

Alias definitions to encodings

Encode::Byte

Single Byte Encodings

Encode::CJKConstants

Internally used by Encode::??::ISO_2022_*

Encode::CN

China-based Chinese Encodings

Encode::CN::HZ

Internally used by Encode::CN

Encode::Config

Internally used by Encode

Encode::EBCDIC

EBCDIC Encodings

Encode::Encoder

Object Oriented Encoder

Encode::Encoding

Encode Implementation Base Class

Encode::Guess

Guesses encoding from data

Encode::JP

Japanese Encodings

Encode::JP::H2Z

Internally used by Encode::JP::2022_JP*

Encode::JP::JIS7

Internally used by Encode::JP

Encode::KR

Korean Encodings

Encode::KR::2022_KR

Internally used by Encode::KR

Encode::MIME::Header

MIME ’B’ and ’Q’ header encoding

Encode::PerlIO

A detailed document on Encode and PerlIO

Encode::Supported

Encodings supported by Encode

Encode::Symbol

Symbol Encodings

Encode::TW

Taiwan-based Chinese Encodings

Encode::Unicode

Various Unicode Transformation Formats

Encode::Unicode::UTF7

UTF-7 encoding

English

Use nice English (or awk) names for ugly punctuation variables

Env

Perl module that imports environment variables as scalars or arrays

Errno

System errno constants

Exporter

Implements default import method for modules

Exporter::Heavy

Exporter guts

ExtUtils::Command

Utilities to replace common UNIX commands in Makefiles etc.

ExtUtils::Command::MM

Commands for the MM’s to use in Makefiles

ExtUtils::Constant

Generate XS code to import C header constants

ExtUtils::Embed

Utilities for embedding Perl in C/C++ applications

ExtUtils::Install

Install files from here to there

ExtUtils::Installed

Inventory management of installed modules

ExtUtils::Liblist

Determine libraries to use and how to use them

ExtUtils::MM

OS adjusted ExtUtils::MakeMaker subclass

ExtUtils::MM_Any

Platform-agnostic MM methods

ExtUtils::MM_BeOS

Methods to override UN*X behaviour in ExtUtils::MakeMaker

ExtUtils::MM_Cygwin

Methods to override UN*X behaviour in ExtUtils::MakeMaker

ExtUtils::MM_DOS

DOS specific subclass of ExtUtils::MM_Unix

ExtUtils::MM_MacOS

Methods to override UN*X behaviour in ExtUtils::MakeMaker

ExtUtils::MM_NW5

Methods to override UN*X behaviour in ExtUtils::MakeMaker

ExtUtils::MM_OS2

Methods to override UN*X behaviour in ExtUtils::MakeMaker

ExtUtils::MM_UWIN

U/WIN specific subclass of ExtUtils::MM_Unix

ExtUtils::MM_Unix

Methods used by ExtUtils::MakeMaker

ExtUtils::MM_VMS

Methods to override UN*X behaviour in ExtUtils::MakeMaker

ExtUtils::MM_Win32

Methods to override UN*X behaviour in ExtUtils::MakeMaker

ExtUtils::MM_Win95

Method to customize MakeMaker for Win9X

ExtUtils::MY

ExtUtils::MakeMaker subclass for customization

ExtUtils::MakeMaker

Create a module Makefile

ExtUtils::MakeMaker::FAQ

Frequently Asked Questions About MakeMaker

ExtUtils::MakeMaker::Tutorial

Writing a module with MakeMaker

ExtUtils::MakeMaker::bytes

Version-agnostic bytes.pm

ExtUtils::MakeMaker::vmsish

Platform-agnostic vmsish.pm

ExtUtils::Manifest

Utilities to write and check a MANIFEST file

ExtUtils::Mkbootstrap

Make a bootstrap file for use by DynaLoader

ExtUtils::Mksymlists

Write linker options files for dynamic extension

ExtUtils::Packlist

Manage .packlist files

ExtUtils::testlib

Add blib/* directories to @INC

Fatal

Replace functions with equivalents which succeed or die

Fcntl

Load the C Fcntl.h defines

File::Basename

Split a pathname into pieces

File::CheckTree

Run many filetest checks on a tree

File::Compare

Compare files or filehandles

File::Copy

Copy files or filehandles

File::DosGlob

DOS like globbing and then some

File::Find

Traverse a directory tree.

File::Glob

Perl extension for BSD glob routine

File::Path

Create or remove directory trees

File::Spec

Portably perform operations on file names

File::Spec::Cygwin

Methods for Cygwin file specs

File::Spec::Epoc

Methods for Epoc file specs

File::Spec::Functions

Portably perform operations on file names

File::Spec::Mac

File::Spec for Mac OS (Classic)

File::Spec::OS2

Methods for OS/2 file specs

File::Spec::Unix

File::Spec for Unix, base for other File::Spec modules

File::Spec::VMS

Methods for VMS file specs

File::Spec::Win32

Methods for Win32 file specs

File::Temp

Return name and handle of a temporary file safely

File::stat

By-name interface to Perl's built-in stat() functions

FileCache

Keep more files open than the system permits

FileHandle

Supply object methods for filehandles

Filter::Simple

Simplified source filtering

Filter::Util::Call

Perl Source Filter Utility Module

FindBin

Locate directory of original perl script

GDBM_File

Perl5 access to the gdbm library.

Getopt::Long

Extended processing of command line options

Getopt::Std

Process single-character switches with switch clustering

Hash::Util

A selection of general-utility hash subroutines

I18N::Collate

Compare 8-bit scalar data according to the current locale

I18N::LangTags

Functions for dealing with RFC3066-style language tags

I18N::LangTags::List

Tags and names for human languages

I18N::Langinfo

Query locale information

IO

Load various IO modules

IO::Dir

Supply object methods for directory handles

IO::File

Supply object methods for filehandles

IO::Handle

Supply object methods for I/O handles

IO::Pipe

Supply object methods for pipes

IO::Poll

Object interface to system poll call

IO::Seekable

Supply seek based methods for I/O objects

IO::Select

OO interface to the select system call

IO::Socket
: Object interface to socket communications

IO::Socket::INET
: Object interface for AF_INET domain sockets

IO::Socket::UNIX
: Object interface for AF_UNIX domain sockets

IPC::Open2
: Open a process for both reading and writing

IPC::Open3
: Open a process for reading, writing, and error handling

IPC::SysV
: SysV IPC constants

IPC::SysV::Msg
: SysV Msg IPC object class

IPC::SysV::Semaphore
: SysV Semaphore IPC object class

List::Util
: A selection of general-utility list subroutines

Locale::Constants
: Constants for Locale codes

Locale::Country
: ISO codes for country identification (ISO 3166)

Locale::Currency
: ISO three letter codes for currency identification (ISO 4217)

Locale::Language
: ISO two letter codes for language identification (ISO 639)

Locale::Maketext
: Framework for localization

Locale::Maketext::TPJ13
: Article about software localization

Locale::Script
: ISO codes for script identification (ISO 15924)

MIME::Base64
: Encoding and decoding of base64 strings

MIME::Base64::QuotedPrint
: Encoding and decoding of quoted-printable strings

Math::BigFloat
: Arbitrary size floating point math package

Math::BigInt
: Arbitrary size integer math package

Math::BigInt::Calc
: Pure Perl module to support Math::BigInt

Math::BigRat
: Arbitrarily big rationals

Math::Complex
: Complex numbers and associated mathematical functions

Math::Trig
: Trigonometric functions

Memoize
: Make functions faster by trading space for time

Memoize::AnyDBM_File
: Glue to provide EXISTS for AnyDBM_File for Storable use

Memoize::Expire
: Plug-in module for automatic expiration of memoized values

Memoize::ExpireFile
: Test for Memoize expiration semantics

Memoize::ExpireTest
: Test for Memoize expiration semantics

Memoize::NDBM_File
: Glue to provide EXISTS for NDBM_File for Storable use

Memoize::SDBM_File
: Glue to provide EXISTS for SDBM_File for Storable use

Memoize::Storable
: Store Memoized data in Storable database

NDBM_File
: Tied access to ndbm files

NEXT
: Provide a pseudo-class NEXT (et al) that allows method redispatch

Net::Cmd
: Network Command class (as used by FTP, SMTP etc)

Net::Config
: Local configuration data for libnet

Net::Domain
: Attempt to evaluate the current host's internet name and domain

Net::FTP
: FTP Client class

Net::NNTP

NNTP Client class

Net::Netrc

OO interface to users netrc file

Net::POP3

Post Office Protocol 3 Client class (RFC1939)

Net::Ping

Check a remote host for reachability

Net::SMTP

Simple Mail Transfer Protocol Client

Net::Time

Time and daytime network client interface

Net::hostent

By-name interface to Perl's built-in gethost*() functions

Net::libnetFAQ

Libnet Frequently Asked Questions

Net::netent

By-name interface to Perl's built-in getnet*() functions

Net::protoent

By-name interface to Perl's built-in getproto*() functions

Net::servent

By-name interface to Perl's built-in getserv*() functions

O

Generic interface to Perl Compiler backends

ODBM_File

Tied access to odbm files

Opcode

Disable named opcodes when compiling perl code

POSIX

Perl interface to IEEE Std 1003.1

PerlIO

On demand loader for PerlIO layers and root of PerlIO::* name space

PerlIO::encoding

Encoding layer

PerlIO::scalar

In-memory IO, scalar IO

PerlIO::via

Helper class for PerlIO layers implemented in perl

PerlIO::via::QuotedPrint
: PerlIO layer for quoted-printable strings

Pod::Checker
: Check pod documents for syntax errors

Pod::Find
: Find POD documents in directory trees

Pod::Functions
: Group Perl's functions a la perlfunc.pod

Pod::Html
: Module to convert pod files to HTML

Pod::InputObjects
: Objects representing POD input paragraphs, commands, etc.

Pod::LaTeX
: Convert Pod data to formatted Latex

Pod::Man
: Convert POD data to formatted *roff input

Pod::ParseLink
: Parse an L<> formatting code in POD text

Pod::ParseUtils
: Helpers for POD parsing and conversion

Pod::Parser
: Base class for creating POD filters and translators

Pod::Perldoc::ToChecker
: Let Perldoc check Pod for errors

Pod::Perldoc::ToMan
: Let Perldoc render Pod as man pages

Pod::Perldoc::ToNroff
: Let Perldoc convert Pod to nroff

Pod::Perldoc::ToPod
: Let Perldoc render Pod as ... Pod!

Pod::Perldoc::ToRtf
: Let Perldoc render Pod as RTF

Pod::Perldoc::ToText
: Let Perldoc render Pod as plaintext

Pod::Perldoc::ToTk
: Let Perldoc use Tk::Pod to render Pod

Pod::Perldoc::ToXml
: Let Perldoc render Pod as XML

Pod::PlainText

Convert POD data to formatted ASCII text

Pod::Plainer

Perl extension for converting Pod to old style Pod.

Pod::Select

Extract selected sections of POD from input

Pod::Text

Convert POD data to formatted ASCII text

Pod::Text::Color

Convert POD data to formatted color ASCII text

Pod::Text::Overstrike

Convert POD data to formatted overstrike text

Pod::Text::Termcap

Convert POD data to ASCII text with format escapes

Pod::Usage

Print a usage message from embedded pod documentation

SDBM_File

Tied access to sdbm files

Safe

Compile and execute code in restricted compartments

Scalar::Util

A selection of general-utility scalar subroutines

Search::Dict

Search for key in dictionary file

SelectSaver

Save and restore selected file handle

SelfLoader

Load functions only on demand

Shell

Run shell commands transparently within perl

Socket

Load the C socket.h defines and structure manipulators

Storable

Persistence for Perl data structures

Switch

A switch statement for Perl

Symbol

Manipulate Perl symbols and their names

Sys::Hostname

Try every conceivable way to get hostname

Sys::Syslog

Perl interface to the UNIX syslog(3) calls

Term::ANSIColor

Color screen output using ANSI escape sequences

Term::Cap

Perl termcap interface

Term::Complete

Perl word completion module

Term::ReadLine

Perl interface to various `readline` packages.

Test

Provides a simple framework for writing test scripts

Test::Builder

Backend for building test libraries

Test::Harness

Run Perl standard test scripts with statistics

Test::Harness::Assert

Simple assert

Test::Harness::Iterator

Internal Test::Harness Iterator

Test::Harness::Straps

Detailed analysis of test results

Test::More

Yet another framework for writing test scripts

Test::Simple

Basic utilities for writing tests.

Test::Tutorial

A tutorial about writing really basic tests

Text::Abbrev

Create an abbreviation table from a list

Text::Balanced

Extract delimited text sequences from strings.

Text::ParseWords

Parse text into an array of tokens or array of arrays

Text::Soundex

Implementation of the Soundex Algorithm as Described by Knuth

Text::Tabs
Expand and unexpand tabs per the unix expand(1) and unexpand(1)

Text::Wrap
Line wrapping to form simple paragraphs

Thread
Manipulate threads in Perl (for old code only)

Thread::Queue
Thread-safe queues

Thread::Semaphore
Thread-safe semaphores

Thread::Signal
Start a thread which runs signal handlers reliably (for old code)

Thread::Specific
Thread-specific keys

Tie::Array
Base class for tied arrays

Tie::File
Access the lines of a disk file via a Perl array

Tie::Handle
Base class definitions for tied handles

Tie::Hash
Base class definitions for tied hashes

Tie::Memoize
Add data to hash when needed

Tie::RefHash
Use references as hash keys

Tie::Scalar
Base class definitions for tied scalars

Tie::SubstrHash
Fixed-table-size, fixed-key-length hashing

Time::HiRes
High resolution alarm, sleep, gettimeofday, interval timers

Time::Local
Efficiently compute time from local and GMT time

Time::gmtime
By-name interface to Perl's built-in gmtime() function

Time::localtime
By-name interface to Perl's built-in localtime() function

Time::tm

Internal object used by Time::gmtime and Time::localtime

UNIVERSAL

Base class for ALL classes (blessed references)

Unicode::Collate

Unicode Collation Algorithm

Unicode::Normalize

Unicode Normalization Forms

Unicode::UCD

Unicode character database

User::grent

By-name interface to Perl's built-in getgr*() functions

User::pwent

By-name interface to Perl's built-in getpw*() functions

Win32

Interfaces to some Win32 API Functions

XS::APItest

Test the perl C API

XS::Typemap

Module to test the XS typemaps distributed with perl

XSLoader

Dynamically load C libraries into Perl code

To find out *all* modules installed on your system, including those without documentation or outside the standard release, just use the following command (under the default win32 shell, double quotes should be used instead of single quotes).

```
% perl -MFile::Find=find -MFile::Spec::Functions -Tlwe \
  'find { wanted => sub { print canonpath $_ if /\.pm\z/ },
  no_chdir => 1 }, @INC'
```

(The -T is here to prevent '.' from being listed in @INC.) They should all have their own documentation installed and accessible via your system man(1) command. If you do not have a **find** program, you can use the Perl **find2perl** program instead, which generates Perl code as output you can run through perl. If you have a **man** program but it doesn't find your modules, you'll have to fix your manpath. See *perl* for details. If you have no system **man** command, you might try the **perldoc** program.

Note also that the command `perldoc perllocal` gives you a (possibly incomplete) list of the modules that have been further installed on your system. (The perllocal.pod file is updated by the standard MakeMaker install process.)

58.1.3 Extension Modules

Extension modules are written in C (or a mix of Perl and C). They are usually dynamically loaded into Perl if and when you need them, but may also be linked in statically. Supported extension modules include Socket, Fcntl, and POSIX.

Many popular C extension modules do not come bundled (at least, not completely) due to their sizes, volatility, or simply lack of time for adequate testing and configuration across the multitude of platforms on which Perl was beta-tested. You are encouraged to look for them on CPAN (described below), or using web search engines like Alta Vista or Google.

58.2 CPAN

CPAN stands for Comprehensive Perl Archive Network; it's a globally replicated trove of Perl materials, including documentation, style guides, tricks and traps, alternate ports to non-Unix systems and occasional binary distributions for these. Search engines for CPAN can be found at http://www.cpan.org/

Most importantly, CPAN includes around a thousand unbundled modules, some of which require a C compiler to build. Major categories of modules are:

- Language Extensions and Documentation Tools
- Development Support
- Operating System Interfaces
- Networking, Device Control (modems) and InterProcess Communication
- Data Types and Data Type Utilities
- Database Interfaces
- User Interfaces
- Interfaces to / Emulations of Other Programming Languages
- File Names, File Systems and File Locking (see also File Handles)
- String Processing, Language Text Processing, Parsing, and Searching
- Option, Argument, Parameter, and Configuration File Processing
- Internationalization and Locale
- Authentication, Security, and Encryption
- World Wide Web, HTML, HTTP, CGI, MIME
- Server and Daemon Utilities
- Archiving and Compression
- Images, Pixmap and Bitmap Manipulation, Drawing, and Graphing
- Mail and Usenet News
- Control Flow Utilities (callbacks and exceptions etc)
- File Handle and Input/Output Stream Utilities
- Miscellaneous Modules

The list of the registered CPAN sites as of this writing follows. Please note that the sorting order is alphabetical on fields:

Continent | |–>Country | |–>[state/province] | |–>ftp | |–>[http]

and thus the North American servers happen to be listed between the European and the South American sites.

You should try to choose one close to you.

58.2.1 Africa

South Africa

```
http://ftp.rucus.ru.ac.za/pub/perl/CPAN/
ftp://ftp.rucus.ru.ac.za/pub/perl/CPAN/
ftp://ftp.is.co.za/programming/perl/CPAN/
ftp://ftp.saix.net/pub/CPAN/
ftp://ftp.sun.ac.za/CPAN/CPAN/
```

58.2.2 Asia

China

```
http://cpan.linuxforum.net/
http://cpan.shellhung.org/
ftp://ftp.shellhung.org/pub/CPAN
ftp://mirrors.hknet.com/CPAN
```

Indonesia

```
http://mirrors.tf.itb.ac.id/cpan/
http://cpan.cbn.net.id/
ftp://ftp.cbn.net.id/mirror/CPAN
```

Israel

```
ftp://ftp.iglu.org.il/pub/CPAN/
http://cpan.lerner.co.il/
http://bioinfo.weizmann.ac.il/pub/software/perl/CPAN/
ftp://bioinfo.weizmann.ac.il/pub/software/perl/CPAN/
```

Japan

```
ftp://ftp.u-aizu.ac.jp/pub/CPAN
ftp://ftp.kddlabs.co.jp/CPAN/
ftp://ftp.ayamura.org/pub/CPAN/
ftp://ftp.jaist.ac.jp/pub/lang/perl/CPAN/
http://ftp.cpan.jp/
ftp://ftp.cpan.jp/CPAN/
ftp://ftp.dti.ad.jp/pub/lang/CPAN/
ftp://ftp.ring.gr.jp/pub/lang/perl/CPAN/
```

Malaysia

```
http://cpan.MyBSD.org.my
http://mirror.leafbug.org/pub/CPAN
http://ossig.mncc.com.my/mirror/pub/CPAN
```

Russian Federation

```
http://cpan.tomsk.ru
ftp://cpan.tomsk.ru/
```

Saudi Arabia

```
ftp://ftp.isu.net.sa/pub/CPAN/
```

Singapore

```
http://CPAN.en.com.sg/
ftp://cpan.en.com.sg/
http://mirror.averse.net/pub/CPAN
ftp://mirror.averse.net/pub/CPAN
http://cpan.oss.eznetsols.org
ftp://ftp.oss.eznetsols.org/cpan
```

South Korea

```
http://CPAN.bora.net/
ftp://ftp.bora.net/pub/CPAN/
http://mirror.kr.FreeBSD.org/CPAN
ftp://ftp.kr.FreeBSD.org/pub/CPAN
```

Taiwan

```
ftp://ftp.nctu.edu.tw/UNIX/perl/CPAN
http://cpan.cdpa.nsysu.edu.tw/
ftp://cpan.cdpa.nsysu.edu.tw/pub/CPAN
http://ftp.isu.edu.tw/pub/CPAN
ftp://ftp.isu.edu.tw/pub/CPAN
ftp://ftp1.sinica.edu.tw/pub1/perl/CPAN/
http://ftp.tku.edu.tw/pub/CPAN/
ftp://ftp.tku.edu.tw/pub/CPAN/
```

Thailand

```
ftp://ftp.loxinfo.co.th/pub/cpan/
ftp://ftp.cs.riubon.ac.th/pub/mirrors/CPAN/
```

58.2.3 Central America

Costa Rica

```
http://ftp.ucr.ac.cr/Unix/CPAN/
ftp://ftp.ucr.ac.cr/pub/Unix/CPAN/
```

58.2.4 Europe

Austria

```
http://cpan.inode.at/
ftp://cpan.inode.at
ftp://ftp.tuwien.ac.at/pub/CPAN/
```

Belgium

```
http://ftp.easynet.be/pub/CPAN/
ftp://ftp.easynet.be/pub/CPAN/
http://cpan.skynet.be
ftp://ftp.cpan.skynet.be/pub/CPAN
ftp://ftp.kulnet.kuleuven.ac.be/pub/mirror/CPAN/
```

Bosnia and Herzegovina

```
http://cpan.blic.net/
```

Bulgaria

```
http://cpan.online.bg
ftp://cpan.online.bg/cpan
http://cpan.zadnik.org
ftp://ftp.zadnik.org/mirrors/CPAN/
http://cpan.lirex.net/
ftp://ftp.lirex.net/pub/mirrors/CPAN
```

Croatia

```
http://ftp.linux.hr/pub/CPAN/
ftp://ftp.linux.hr/pub/CPAN/
```

Czech Republic

```
ftp://ftp.fi.muni.cz/pub/CPAN/
ftp://sunsite.mff.cuni.cz/MIRRORS/ftp.funet.fi/pub/languages/perl/CPAN/
```

Denmark

```
http://mirrors.sunsite.dk/cpan/
ftp://sunsite.dk/mirrors/cpan/
http://cpan.cybercity.dk
http://www.cpan.dk/CPAN/
ftp://www.cpan.dk/ftp.cpan.org/CPAN/
```

Estonia

```
ftp://ftp.ut.ee/pub/languages/perl/CPAN/
```

Finland

```
ftp://ftp.funet.fi/pub/languages/perl/CPAN/
http://mirror.eunet.fi/CPAN
```

France

```
http://www.enstimac.fr/Perl/CPAN
http://ftp.u-paris10.fr/perl/CPAN
ftp://ftp.u-paris10.fr/perl/CPAN
http://cpan.mirrors.easynet.fr/
ftp://cpan.mirrors.easynet.fr/pub/ftp.cpan.org/
ftp://ftp.club-internet.fr/pub/perl/CPAN/
http://fr.cpan.org/
ftp://ftp.lip6.fr/pub/perl/CPAN/
ftp://ftp.oleane.net/pub/mirrors/CPAN/
ftp://ftp.pasteur.fr/pub/computing/CPAN/
http://mir2.ovh.net/ftp.cpan.org
ftp://mir1.ovh.net/ftp.cpan.org
http://ftp.crihan.fr/mirrors/ftp.cpan.org/
ftp://ftp.crihan.fr/mirrors/ftp.cpan.org/
http://ftp.u-strasbg.fr/CPAN
ftp://ftp.u-strasbg.fr/CPAN
ftp://cpan.cict.fr/pub/CPAN/
ftp://ftp.uvsq.fr/pub/perl/CPAN/
```

Germany

```
ftp://ftp.rub.de/pub/CPAN/
ftp://ftp.freenet.de/pub/ftp.cpan.org/pub/CPAN/
ftp://ftp.uni-erlangen.de/pub/source/CPAN/
ftp://ftp-stud.fht-esslingen.de/pub/Mirrors/CPAN
http://pandemonium.tiscali.de/pub/CPAN/
ftp://pandemonium.tiscali.de/pub/CPAN/
http://ftp.gwdg.de/pub/languages/perl/CPAN/
```

```
ftp://ftp.gwdg.de/pub/languages/perl/CPAN/
ftp://ftp.uni-hamburg.de/pub/soft/lang/perl/CPAN/
ftp://ftp.leo.org/pub/CPAN/
http://cpan.noris.de/
ftp://cpan.noris.de/pub/CPAN/
ftp://ftp.mpi-sb.mpg.de/pub/perl/CPAN/
ftp://ftp.gmd.de/mirrors/CPAN/
```

Greece

```
ftp://ftp.acn.gr/pub/lang/perl
ftp://ftp.forthnet.gr/pub/languages/perl/CPAN
ftp://ftp.ntua.gr/pub/lang/perl/
```

Hungary

```
http://ftp.kfki.hu/packages/perl/CPAN/
ftp://ftp.kfki.hu/pub/packages/perl/CPAN/
```

Iceland

```
http://ftp.rhnet.is/pub/CPAN/
ftp://ftp.rhnet.is/pub/CPAN/
```

Ireland

```
http://cpan.indigo.ie/
ftp://cpan.indigo.ie/pub/CPAN/
http://ftp.heanet.ie/mirrors/ftp.perl.org/pub/CPAN
ftp://ftp.heanet.ie/mirrors/ftp.perl.org/pub/CPAN
http://sunsite.compapp.dcu.ie/pub/perl/
ftp://sunsite.compapp.dcu.ie/pub/perl/
```

Italy

```
http://cpan.nettuno.it/
http://gusp.dyndns.org/CPAN/
ftp://gusp.dyndns.org/pub/CPAN
http://softcity.iol.it/cpan
ftp://softcity.iol.it/pub/cpan
ftp://ftp.unina.it/pub/Other/CPAN/CPAN/
ftp://ftp.unipi.it/pub/mirror/perl/CPAN/
ftp://cis.uniRoma2.it/CPAN/
ftp://ftp.edisontel.it/pub/CPAN_Mirror/
http://cpan.flashnet.it/
ftp://ftp.flashnet.it/pub/CPAN/
```

Latvia

```
http://kvin.lv/pub/CPAN/
```

Lithuania

```
ftp://ftp.unix.lt/pub/CPAN/
```

Netherlands

```
ftp://download.xs4all.nl/pub/mirror/CPAN/
ftp://ftp.nl.uu.net/pub/CPAN/
ftp://ftp.nluug.nl/pub/languages/perl/CPAN/
http://cpan.cybercomm.nl/
ftp://mirror.cybercomm.nl/pub/CPAN
ftp://mirror.vuurwerk.nl/pub/CPAN/
ftp://ftp.cpan.nl/pub/CPAN/
http://ftp.easynet.nl/mirror/CPAN
ftp://ftp.easynet.nl/mirror/CPAN
http://archive.cs.uu.nl/mirror/CPAN/
ftp://ftp.cs.uu.nl/mirror/CPAN/
```

Norway

```
ftp://ftp.uninett.no/pub/languages/perl/CPAN
ftp://ftp.uit.no/pub/languages/perl/cpan/
```

Poland

```
ftp://ftp.mega.net.pl/CPAN
ftp://ftp.man.torun.pl/pub/doc/CPAN/
ftp://sunsite.icm.edu.pl/pub/CPAN/
```

Portugal

```
ftp://ftp.ua.pt/pub/CPAN/
ftp://perl.di.uminho.pt/pub/CPAN/
http://cpan.dei.uc.pt/
ftp://ftp.dei.uc.pt/pub/CPAN
ftp://ftp.nfsi.pt/pub/CPAN
http://ftp.linux.pt/pub/mirrors/CPAN
ftp://ftp.linux.pt/pub/mirrors/CPAN
http://cpan.ip.pt/
ftp://cpan.ip.pt/pub/cpan/
http://cpan.telepac.pt/
ftp://ftp.telepac.pt/pub/cpan/
```

Romania

```
ftp://ftp.bio-net.ro/pub/CPAN
ftp://ftp.kappa.ro/pub/mirrors/ftp.perl.org/pub/CPAN/
ftp://ftp.lug.ro/CPAN
ftp://ftp.roedu.net/pub/CPAN/
ftp://ftp.dntis.ro/pub/cpan/
ftp://ftp.iasi.roedu.net/pub/mirrors/ftp.cpan.org/
http://cpan.ambra.ro/
ftp://ftp.ambra.ro/pub/CPAN
ftp://ftp.dnttm.ro/pub/CPAN/
ftp://ftp.lasting.ro/pub/CPAN
ftp://ftp.timisoara.roedu.net/mirrors/CPAN/
```

Russia

```
ftp://ftp.chg.ru/pub/lang/perl/CPAN/
http://cpan.rinet.ru/
ftp://cpan.rinet.ru/pub/mirror/CPAN/
ftp://ftp.aha.ru/pub/CPAN/
```

```
ftp://ftp.corbina.ru/pub/CPAN/
http://cpan.sai.msu.ru/
ftp://ftp.sai.msu.su/pub/lang/perl/CPAN/
```

Slovakia

```
ftp://ftp.cvt.stuba.sk/pub/CPAN/
```

Slovenia

```
ftp://ftp.arnes.si/software/perl/CPAN/
```

Spain

```
http://cpan.imasd.elmundo.es/
ftp://ftp.rediris.es/mirror/CPAN/
ftp://ftp.ri.telefonica-data.net/CPAN
ftp://ftp.etse.urv.es/pub/perl/
```

Sweden

```
http://ftp.du.se/CPAN/
ftp://ftp.du.se/pub/CPAN/
http://mirror.dataphone.se/CPAN
ftp://mirror.dataphone.se/pub/CPAN
ftp://ftp.sunet.se/pub/lang/perl/CPAN/
```

Switzerland

```
http://cpan.mirror.solnet.ch/
ftp://ftp.solnet.ch/mirror/CPAN/
ftp://ftp.danyk.ch/CPAN/
ftp://sunsite.cnlab-switch.ch/mirror/CPAN/
```

Turkey

```
http://ftp.ulak.net.tr/perl/CPAN/
ftp://ftp.ulak.net.tr/perl/CPAN
ftp://sunsite.bilkent.edu.tr/pub/languages/CPAN/
```

Ukraine

```
http://cpan.org.ua/
ftp://cpan.org.ua/
ftp://ftp.perl.org.ua/pub/CPAN/
http://no-more.kiev.ua/CPAN/
ftp://no-more.kiev.ua/pub/CPAN/
```

United Kingdom

```
http://www.mirror.ac.uk/sites/ftp.funet.fi/pub/languages/perl/CPAN
ftp://ftp.mirror.ac.uk/sites/ftp.funet.fi/pub/languages/perl/CPAN/
http://cpan.teleglobe.net/
ftp://cpan.teleglobe.net/pub/CPAN
http://cpan.mirror.anlx.net/
ftp://ftp.mirror.anlx.net/CPAN/
http://cpan.etla.org/
```

```
ftp://cpan.etla.org/pub/CPAN
ftp://ftp.demon.co.uk/pub/CPAN/
http://cpan.m.flirble.org/
ftp://ftp.flirble.org/pub/languages/perl/CPAN/
ftp://ftp.plig.org/pub/CPAN/
http://cpan.hambule.co.uk/
http://cpan.mirrors.clockerz.net/
ftp://ftp.clockerz.net/pub/CPAN/
ftp://usit.shef.ac.uk/pub/packages/CPAN/
```

58.2.5 North America

Canada

Alberta

```
http://cpan.sunsite.ualberta.ca/
ftp://cpan.sunsite.ualberta.ca/pub/CPAN/
```

Manitoba

```
http://theoryx5.uwinnipeg.ca/pub/CPAN/
ftp://theoryx5.uwinnipeg.ca/pub/CPAN/
```

Nova Scotia

```
ftp://cpan.chebucto.ns.ca/pub/CPAN/
```

Ontario

```
ftp://ftp.nrc.ca/pub/CPAN/
```

Mexico

```
http://cpan.azc.uam.mx
ftp://cpan.azc.uam.mx/mirrors/CPAN
http://www.cpan.unam.mx/
ftp://ftp.unam.mx/pub/CPAN
http://www.msg.com.mx/CPAN/
ftp://ftp.msg.com.mx/pub/CPAN/
```

United States

Alabama

```
http://mirror.hiwaay.net/CPAN/
ftp://mirror.hiwaay.net/CPAN/
```

California

```
http://cpan.develooper.com/
http://www.cpan.org/
ftp://cpan.valueclick.com/pub/CPAN/
http://www.mednor.net/ftp/pub/mirrors/CPAN/
ftp://ftp.mednor.net/pub/mirrors/CPAN/
http://mirrors.gossamer-threads.com/CPAN
ftp://cpan.nas.nasa.gov/pub/perl/CPAN/
http://mirrors.kernel.org/cpan/
ftp://mirrors.kernel.org/pub/CPAN
http://cpan-sj.viaverio.com/
ftp://cpan-sj.viaverio.com/pub/CPAN/
http://cpan.digisle.net/
```

```
ftp://cpan.digisle.net/pub/CPAN
http://www.perl.com/CPAN/
http://www.uberlan.net/CPAN
```

Colorado

```
ftp://ftp.cs.colorado.edu/pub/perl/CPAN/
http://cpan.four10.com
```

Delaware

```
http://ftp.lug.udel.edu/pub/CPAN
ftp://ftp.lug.udel.edu/pub/CPAN
```

District of Columbia

```
ftp://ftp.dc.aleron.net/pub/CPAN/
```

Florida

```
ftp://ftp.cise.ufl.edu/pub/mirrors/CPAN/
http://mirror.csit.fsu.edu/pub/CPAN/
ftp://mirror.csit.fsu.edu/pub/CPAN/
http://cpan.mirrors.nks.net/
```

Indiana

```
ftp://ftp.uwsg.iu.edu/pub/perl/CPAN/
http://cpan.netnitco.net/
ftp://cpan.netnitco.net/pub/mirrors/CPAN/
http://archive.progeny.com/CPAN/
ftp://archive.progeny.com/CPAN/
http://fx.saintjoe.edu/pub/CPAN
ftp://ftp.saintjoe.edu/pub/CPAN
http://csociety-ftp.ecn.purdue.edu/pub/CPAN
ftp://csociety-ftp.ecn.purdue.edu/pub/CPAN
```

Kentucky

```
http://cpan.uky.edu/
ftp://cpan.uky.edu/pub/CPAN/
http://slugsite.louisville.edu/cpan
ftp://slugsite.louisville.edu/CPAN
```

Massachusetts

```
http://mirrors.towardex.com/CPAN
ftp://mirrors.towardex.com/pub/CPAN
ftp://ftp.ccs.neu.edu/net/mirrors/ftp.funet.fi/pub/languages/perl/CPAN/
```

Michigan

```
ftp://cpan.cse.msu.edu/
http://cpan.calvin.edu/pub/CPAN
ftp://cpan.calvin.edu/pub/CPAN
```

Nevada

```
http://www.oss.redundant.com/pub/CPAN
ftp://www.oss.redundant.com/pub/CPAN
```

New Jersey

```
http://ftp.cpanel.net/pub/CPAN/
ftp://ftp.cpanel.net/pub/CPAN/
http://cpan.teleglobe.net/
ftp://cpan.teleglobe.net/pub/CPAN
```

New York

```
http://cpan.belfry.net/
http://cpan.erlbaum.net/
ftp://cpan.erlbaum.net/
http://cpan.thepirtgroup.com/
ftp://cpan.thepirtgroup.com/
ftp://ftp.stealth.net/pub/CPAN/
http://www.rge.com/pub/languages/perl/
ftp://ftp.rge.com/pub/languages/perl/
```

North Carolina

```
http://www.ibiblio.org/pub/languages/perl/CPAN
ftp://ftp.ibiblio.org/pub/languages/perl/CPAN
ftp://ftp.duke.edu/pub/perl/
ftp://ftp.ncsu.edu/pub/mirror/CPAN/
```

Oklahoma

```
ftp://ftp.ou.edu/mirrors/CPAN/
```

Oregon

```
ftp://ftp.orst.edu/pub/CPAN
```

Pennsylvania

```
http://ftp.epix.net/CPAN/
ftp://ftp.epix.net/pub/languages/perl/
http://mirrors.phenominet.com/pub/CPAN/
ftp://mirrors.phenominet.com/pub/CPAN/
http://cpan.pair.com/
ftp://cpan.pair.com/pub/CPAN/
ftp://carroll.cac.psu.edu/pub/CPAN/
```

Tennessee

```
ftp://ftp.sunsite.utk.edu/pub/CPAN/
```

Texas

```
http://ftp.sedl.org/pub/mirrors/CPAN/
http://www.binarycode.org/cpan
ftp://mirror.telentente.com/pub/CPAN
http://mirrors.theonlinerecordstore.com/CPAN
```

Utah

```
ftp://mirror.xmission.com/CPAN/
```

Virginia

```
http://cpan-du.viaverio.com/
ftp://cpan-du.viaverio.com/pub/CPAN/
http://mirrors.rcn.net/pub/lang/CPAN/
ftp://mirrors.rcn.net/pub/lang/CPAN/
http://perl.secsup.org/
ftp://perl.secsup.org/pub/perl/
http://noc.cvaix.com/mirrors/CPAN/
```

Washington

```
http://cpan.llarian.net/
ftp://cpan.llarian.net/pub/CPAN/
http://cpan.mirrorcentral.com/
ftp://ftp.mirrorcentral.com/pub/CPAN/
ftp://ftp-mirror.internap.com/pub/CPAN/
```

Wisconsin

```
http://mirror.sit.wisc.edu/pub/CPAN/
ftp://mirror.sit.wisc.edu/pub/CPAN/
http://mirror.aphix.com/CPAN
ftp://mirror.aphix.com/pub/CPAN
```

58.2.6 Oceania

Australia

```
http://ftp.planetmirror.com/pub/CPAN/
ftp://ftp.planetmirror.com/pub/CPAN/
ftp://mirror.aarnet.edu.au/pub/perl/CPAN/
ftp://cpan.topend.com.au/pub/CPAN/
http://cpan.mirrors.ilisys.com.au
```

New Zealand

```
ftp://ftp.auckland.ac.nz/pub/perl/CPAN/
```

United States

```
http://aniani.ifa.hawaii.edu/CPAN/
ftp://aniani.ifa.hawaii.edu/CPAN/
```

58.2.7 South America

Argentina

```
ftp://mirrors.bannerlandia.com.ar/mirrors/CPAN/
http://www.linux.org.ar/mirrors/cpan
ftp://ftp.linux.org.ar/mirrors/cpan
```

Brazil

```
ftp://cpan.pop-mg.com.br/pub/CPAN/
ftp://ftp.matrix.com.br/pub/perl/CPAN/
http://cpan.hostsul.com.br/
ftp://cpan.hostsul.com.br/
```

Chile

```
http://cpan.netglobalis.net/
ftp://cpan.netglobalis.net/pub/CPAN/
```

58.2.8 RSYNC Mirrors

```
www.linux.org.ar::cpan
theoryx5.uwinnipeg.ca::CPAN
ftp.shellhung.org::CPAN
rsync.nic.funet.fi::CPAN
ftp.u-paris10.fr::CPAN
mir1.ovh.net::CPAN
rsync://ftp.crihan.fr::CPAN
ftp.gwdg.de::FTP/languages/perl/CPAN/
ftp.leo.org::CPAN
ftp.cbn.net.id::CPAN
rsync://ftp.heanet.ie/mirrors/ftp.perl.org/pub/CPAN
ftp.iglu.org.il::CPAN
gusp.dyndns.org::cpan
ftp.kddlabs.co.jp::cpan
ftp.ayamura.org::pub/CPAN/
mirror.leafbug.org::CPAN
rsync.en.com.sg::CPAN
mirror.averse.net::cpan
rsync.oss.eznetsols.org
ftp.kr.FreeBSD.org::CPAN
ftp.solnet.ch::CPAN
cpan.cdpa.nsysu.edu.tw::CPAN
cpan.teleglobe.net::CPAN
rsync://rsync.mirror.anlx.net::CPAN
ftp.sedl.org::cpan
ibiblio.org::CPAN
cpan-du.viaverio.com::CPAN
aniani.ifa.hawaii.edu::CPAN
archive.progeny.com::CPAN
rsync://slugsite.louisville.edu::CPAN
mirror.aphix.com::CPAN
cpan.teleglobe.net::CPAN
ftp.lug.udel.edu::cpan
mirrors.kernel.org::mirrors/CPAN
mirrors.phenominet.com::CPAN
cpan.pair.com::CPAN
cpan-sj.viaverio.com::CPAN
mirror.csit.fsu.edu::CPAN
csociety-ftp.ecn.purdue.edu::CPAN
```

For an up-to-date listing of CPAN sites, see http://www.cpan.org/SITES or ftp://www.cpan.org/SITES .

58.3 Modules: Creation, Use, and Abuse

(The following section is borrowed directly from Tim Bunce's modules file, available at your nearest CPAN site.)

Perl implements a class using a package, but the presence of a package doesn't imply the presence of a class. A package is just a namespace. A class is a package that provides subroutines that can be used as methods. A method is just a subroutine that expects, as its first argument, either the name of a package (for "static" methods), or a reference to something (for "virtual" methods).

A module is a file that (by convention) provides a class of the same name (sans the .pm), plus an import method in that class that can be called to fetch exported symbols. This module may implement some of its methods by loading dynamic C or C++ objects, but that should be totally transparent to the user of the module. Likewise, the module might set up an AUTOLOAD function to slurp in subroutine definitions on demand, but this is also transparent. Only the *.pm* file is required to exist. See *perlsub*, *perltoot*, and *AutoLoader* for details about the AUTOLOAD mechanism.

58.3.1 Guidelines for Module Creation

- Do similar modules already exist in some form?

  If so, please try to reuse the existing modules either in whole or by inheriting useful features into a new class. If this is not practical try to get together with the module authors to work on extending or enhancing the functionality of the existing modules. A perfect example is the plethora of packages in perl4 for dealing with command line options.

If you are writing a module to expand an already existing set of modules, please coordinate with the author of the package. It helps if you follow the same naming scheme and module interaction scheme as the original author.

- Try to design the new module to be easy to extend and reuse.

  Try to `use warnings;` (or `use warnings qw(...);`). Remember that you can add `no warnings qw(...);` to individual blocks of code that need less warnings.

  Use blessed references. Use the two argument form of bless to bless into the class name given as the first parameter of the constructor, e.g.,:

```
sub new {
    my $class = shift;
    return bless {}, $class;
}
```

  or even this if you'd like it to be used as either a static or a virtual method.

```
sub new {
    my $self  = shift;
    my $class = ref($self) || $self;
    return bless {}, $class;
}
```

  Pass arrays as references so more parameters can be added later (it's also faster). Convert functions into methods where appropriate. Split large methods into smaller more flexible ones. Inherit methods from other modules if appropriate.

  Avoid class name tests like: `die "Invalid" unless ref $ref eq 'FOO'`. Generally you can delete the `eq 'FOO'` part with no harm at all. Let the objects look after themselves! Generally, avoid hard-wired class names as far as possible.

  Avoid `$r->Class::func()` where using `@ISA=qw(... Class ...)` and `$r->func()` would work (see *perlbot* for more details).

  Use autosplit so little used or newly added functions won't be a burden to programs that don't use them. Add test functions to the module after __END__ either using AutoSplit or by saying:

```
eval join('',<main::DATA>) || die $@ unless caller();
```

  Does your module pass the 'empty subclass' test? If you say `@SUBCLASS::ISA = qw(YOURCLASS);` your applications should be able to use SUBCLASS in exactly the same way as YOURCLASS. For example, does your application still work if you change: `$obj = new YOURCLASS;` into: `$obj = new SUBCLASS;` ?

  Avoid keeping any state information in your packages. It makes it difficult for multiple other packages to use yours. Keep state information in objects.

  Always use **-w**.

  Try to `use strict;` (or `use strict qw(...);`). Remember that you can add `no strict qw(...);` to individual blocks of code that need less strictness.

  Always use **-w**.

  Follow the guidelines in the perlstyle(1) manual.

  Always use **-w**.

- Some simple style guidelines

  The perlstyle manual supplied with Perl has many helpful points.

  Coding style is a matter of personal taste. Many people evolve their style over several years as they learn what helps them write and maintain good code. Here's one set of assorted suggestions that seem to be widely used by experienced developers:

Use underscores to separate words. 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/Module names are an exception to this rule. Perl informally reserves lowercase module names for 'pragma' modules like integer and strict. Other modules normally begin with a capital letter and use mixed case with no underscores (need to be short and portable).

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.

- Select what to export.

  Do NOT export method names!

  Do NOT export anything else by default without a good reason!

  Exports pollute the namespace of the module user. If you must export try to use @EXPORT_OK in preference to @EXPORT and avoid short or common names to reduce the risk of name clashes.

  Generally anything not exported is still accessible from outside the module using the ModuleName::item_name (or `$blessed_ref->method`) syntax. By convention you can use a leading underscore on names to indicate informally that they are 'internal' and not for public use.

  (It is actually possible to get private functions by saying: `my $subref = sub { ...  }; &$subref;`. But there's no way to call that directly as a method, because a method must have a name in the symbol table.)

  As a general rule, if the module is trying to be object oriented then export nothing. If it's just a collection of functions then @EXPORT_OK anything but use @EXPORT with caution.

- Select a name for the module.

  This name should be as descriptive, accurate, and complete as possible. Avoid any risk of ambiguity. Always try to use two or more whole words. Generally the name should reflect what is special about what the module does rather than how it does it. Please use nested module names to group informally or categorize a module. There should be a very good reason for a module not to have a nested name. Module names should begin with a capital letter.

  Having 57 modules all called Sort will not make life easy for anyone (though having 23 called Sort::Quick is only marginally better :-). Imagine someone trying to install your module alongside many others. If in any doubt ask for suggestions in comp.lang.perl.misc.

  If you are developing a suite of related modules/classes it's good practice to use nested classes with a common prefix as this will avoid namespace clashes. For example: Xyz::Control, Xyz::View, Xyz::Model etc. Use the modules in this list as a naming guide.

  If adding a new module to a set, follow the original author's standards for naming modules and the interface to methods in those modules.

  If developing modules for private internal or project specific use, that will never be released to the public, then you should ensure that their names will not clash with any future public module. You can do this either by using the reserved Local::* category or by using a category name that includes an underscore like Foo_Corp::*.

  To be portable each component of a module name should be limited to 11 characters. If it might be used on MS-DOS then try to ensure each is unique in the first 8 characters. Nested modules make this easier.

- Have you got it right?

  How do you know that you've made the right decisions? Have you picked an interface design that will cause problems later? Have you picked the most appropriate name? Do you have any questions?

The best way to know for sure, and pick up many helpful suggestions, is to ask someone who knows. Comp.lang.perl.misc is read by just about all the people who develop modules and it's the best place to ask.

All you need to do is post a short summary of the module, its purpose and interfaces. A few lines on each of the main methods is probably enough. (If you post the whole module it might be ignored by busy people - generally the very people you want to read it!)

Don't worry about posting if you can't say when the module will be ready - just say so in the message. It might be worth inviting others to help you, they may be able to complete it for you!

- README and other Additional Files.

  It's well known that software developers usually fully document the software they write. If, however, the world is in urgent need of your software and there is not enough time to write the full documentation please at least provide a README file containing:

  - A description of the module/package/extension etc.
  - A copyright notice - see below.
  - Prerequisites - what else you may need to have.
  - How to build it - possible changes to Makefile.PL etc.
  - How to install it.
  - Recent changes in this release, especially incompatibilities
  - Changes / enhancements you plan to make in the future.

  If the README file seems to be getting too large you may wish to split out some of the sections into separate files: INSTALL, Copying, ToDo etc.

  - Adding a Copyright Notice.

    How you choose to license your work is a personal decision. The general mechanism is to assert your Copyright and then make a declaration of how others may copy/use/modify your work.

    Perl, for example, is supplied with two types of licence: The GNU GPL and The Artistic Licence (see the files README, Copying, and Artistic, or *perlgpl* and *perlartistic*). Larry has good reasons for NOT just using the GNU GPL.

    My personal recommendation, out of respect for Larry, Perl, and the Perl community at large is to state something simply like:

    ```
    Copyright (c) 1995 Your Name. All rights reserved.
    This program is free software; you can redistribute it and/or
    modify it under the same terms as Perl itself.
    ```

    This statement should at least appear in the README file. You may also wish to include it in a Copying file and your source files. Remember to include the other words in addition to the Copyright.

  - Give the module a version/issue/release number.

    To be fully compatible with the Exporter and MakeMaker modules you should store your module's version number in a non-my package variable called $VERSION. This should be a floating point number with at least two digits after the decimal (i.e., hundredths, e.g, `$VERSION = "0.01"`). Don't use a "1.3.2" style version. See *Exporter* for details.

    It may be handy to add a function or method to retrieve the number. Use the number in announcements and archive file names when releasing the module (ModuleName-1.02.tar.Z). See perldoc ExtUtils::MakeMaker.pm for details.

  - How to release and distribute a module.

    It's good idea to post an announcement of the availability of your module (or the module itself if small) to the comp.lang.perl.announce Usenet newsgroup. This will at least ensure very wide once-off distribution.

    If possible, register the module with CPAN. You should include details of its location in your announcement.

    Some notes about ftp archives: Please use a long descriptive file name that includes the version number. Most incoming directories will not be readable/listable, i.e., you won't be able to see your file after uploading it. Remember to send your email notification message as soon as possible after uploading else your file may get deleted automatically. Allow time for the file to be processed and/or check the file has been processed before announcing its location.

FTP Archives for Perl Modules:

Follow the instructions and links on:

```
http://www.cpan.org/modules/00modlist.long.html
http://www.cpan.org/modules/04pause.html
```

or upload to one of these sites:

```
https://pause.kbx.de/pause/
http://pause.perl.org/pause/
```

and notify <modules@perl.org>.

By using the WWW interface you can ask the Upload Server to mirror your modules from your ftp or WWW site into your own directory on CPAN!

Please remember to send me an updated entry for the Module list!

- Take care when changing a released module.

  Always strive to remain compatible with previous released versions. Otherwise try to add a mechanism to revert to the old behavior if people rely on it. Document incompatible changes.

58.3.2 Guidelines for Converting Perl 4 Library Scripts into Modules

- There is no requirement to convert anything.

  If it ain't broke, don't fix it! Perl 4 library scripts should continue to work with no problems. You may need to make some minor changes (like escaping non-array @'s in double quoted strings) but there is no need to convert a .pl file into a Module for just that.

- Consider the implications.

  All Perl applications that make use of the script will need to be changed (slightly) if the script is converted into a module. Is it worth it unless you plan to make other changes at the same time?

- Make the most of the opportunity.

  If you are going to convert the script to a module you can use the opportunity to redesign the interface. The guidelines for module creation above include many of the issues you should consider.

- The pl2pm utility will get you started.

  This utility will read *.pl files (given as parameters) and write corresponding *.pm files. The pl2pm utilities does the following:

  - Adds the standard Module prologue lines
  - Converts package specifiers from ' to ::
  - Converts die(...) to croak(...)
  - Several other minor changes

  Being a mechanical process pl2pm is not bullet proof. The converted code will need careful checking, especially any package statements. Don't delete the original .pl file till the new .pm one works!

58.3.3 Guidelines for Reusing Application Code

- Complete applications rarely belong in the Perl Module Library.
- Many applications contain some Perl code that could be reused.
  Help save the world! Share your code in a form that makes it easy to reuse.
- Break-out the reusable code into one or more separate module files.
- Take the opportunity to reconsider and redesign the interfaces.
- In some cases the 'application' can then be reduced to a small
  fragment of code built on top of the reusable modules. In these cases the application could invoked as:

```
     % perl -e 'use Module::Name; method(@ARGV)' ...
or
     % perl -mModule::Name ...    (in perl5.002 or higher)
```

58.4 NOTE

Perl does not enforce private and public parts of its modules as you may have been used to in other languages like C++, Ada, or Modula-17. Perl doesn't have an infatuation with enforced privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun.

The module and its user have a contract, part of which is common law, and part of which is "written". Part of the common law contract is that a module doesn't pollute any namespace it wasn't asked to. The written contract for the module (A.K.A. documentation) may make other provisions. But then you know when you `use RedefineTheWorld` that you're redefining the world and willing to take the consequences.

Chapter 59

perlmodstyle

Perl module style guide

59.1 INTRODUCTION

This document attempts to describe the Perl Community's "best practice" for writing Perl modules. It extends the recommendations found in *perlstyle* , 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 *perlnewmod* 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.

59.2 QUICK CHECKLIST

For more detail on each item in this checklist, see below.

59.2.1 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

59.2.2 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

59.2.3 Stability

- Ensure your module works under `use strict` and `-w`
- Stable modules should maintain backwards compatibility

59.2.4 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)

59.2.5 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)

59.3 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.

59.3.1 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 asking on modules@perl.org

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.

59.3.2 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.

59.3.3 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.

You should contact modules@perl.org to ask them about your module name before publishing your module. You should also try to ask 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.

59.4 DESIGNING AND WRITING YOUR MODULE

Considerations for module design and coding:

59.4.1 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.

According to Damian Conway, you should consider using OO:

- When the system is large or likely to become so
- When the data is aggregated in obvious structures that will become objects
- When the types of data form a natural hierarchy that can make use of inheritance
- When operations on data vary according to data type (making polymorphic invocation of methods feasible)
- When it is likely that new data types may be later introduced into the system, and will need to be handled by existing code

- When interactions between data are best represented by overloaded operators
- When the implementation of system components is likely to change over time (and hence should be encapsulated)
- When the system design is itself object-oriented
- When large amounts of client code will use the software (and should be insulated from changes in its implementation)
- When many separate operations will need to be applied to the same set of data

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.

59.4.2 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, 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!

59.4.3 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.

59.4.4 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.

59.4.5 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.

59.5 DOCUMENTING YOUR MODULE

59.5.1 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).

59.5.2 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.

59.6 RELEASE CONSIDERATIONS

59.6.1 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:

```
  $VERSION = "1.12_01";
  $XS_VERSION = $VERSION; # only needed if you have XS code
  $VERSION = eval $VERSION;
```

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.

59.6.2 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 require in *perlfunc* for details.

59.6.3 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.

59.6.4 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 *perlnewmod*.

59.6.5 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 *perlgpl* and *perlartistic*.

59.7 COMMON PITFALLS

59.7.1 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.

59.7.2 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.

59.7.3 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.

59.8 SEE ALSO

perlstyle

General Perl style guide

perlnewmod

How to create a new module

perlpod

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

59.9 AUTHOR

Kirrily "Skud" Robert <skud@cpan.org>

Chapter 60

perlmodinstall

Installing CPAN Modules

60.1 DESCRIPTION

You can think of a module as the fundamental unit of reusable Perl code; see *perlmod* 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.

60.1.1 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 Linux 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/EN-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,**

  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://www.aladdinsys.com/expander/) in combination with **DropStuff with Expander Enhancer** (http://www.aladdinsys.com/dropstuff/) 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 §60.2.)

  If a module does not work on MacPerl but should, or needs to be compiled, see if the module exists already as a port on the MacPerl Module Porters site (http://pudge.net/mmp/). For more information on doing XS with MacPerl yourself, see Arved Sandstrom's XS tutorial (http://macperl.com/depts/Tutorials/), and then consider uploading your binary to the CPAN and registering it on the MMP site.

  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.simtel.net/pub/simtelnet/gnu/djgpp/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.openvms.digital.com/freeware/
http://www.crinoid.com/utils/
```

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/

60.2 PORTABILITY

Note that not all modules will work with on all platforms. See *perlport* 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/
```

60.3 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.

60.4 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.

60.5 COPYRIGHT

Chapter 61

perlnewmod

Preparing a new module for distribution

61.1 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 *perlmod*, *perlmodlib* and *perlmodinstall* 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.

61.1.1 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.

61.1.2 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.

61.1.3 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 like Time::Zone, File::Copy and then some of the `Mail::*` modules if you're planning on writing object oriented code.

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 modules list and the *by-module* directories, 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.

61.1.4 Step-by-step: Making the module

Start with *h2xs*

Originally a utility to convert C header files into XS modules, h2xs has become a useful utility for churning out skeletons for Perl-only modules as well. If you don't want to use the Autoloader which splits up big modules into smaller subroutine-sized chunks, you'll say something like this:

```
h2xs -AX -n Net::Acme
```

The `-A` omits the Autoloader code, `-X` omits XS elements, 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!

`h2xs` provides stubs for Exporter, which 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 plain old documentation

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. `h2xs` will provide a stub for you to fill in; if you're not sure about the format, look at *perlpod* 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, `h2xs` provides a test framework which you can extend - you should do something more than just checking your module will compile.

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.

61.1.5 Step-by-step: Distributing your module

Get a CPAN user ID

Every developer publishing modules on CPAN needs a CPAN ID. See the instructions at `http://www.cpan.org/modules/04pause.html` (or equivalent on your nearest mirror) to find out how to do this.

`perl Makefile.PL; make test; make dist`

Once again, `h2xs` has done all the work for you. It produces the standard `Makefile.PL` you'll have seen when you downloaded and installs 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 tell the modules list about it. The best way to do this is to email them a line in the style of the modules list, like this:

```
Net::Acme bdpOP   Interface to Acme Frobnicator servers      FOOBAR
^         ^^^^^   ^                                          ^
|         |||||   Module description                         Your ID
|         |||||
|         ||||\-Public Licence: (p)standard Perl, (g)GPL, (b)BSD,
|         ||||                  (l)LGPL, (a)rtistic, (o)ther
|         ||||
|         |||\- Interface: (O)OP, (r)eferences, (h)ybrid, (f)unctions
|         |||
|         ||\-- Language: (p)ure Perl, C(+)+, (h)ybrid, (C), (o)ther
|         ||
Module    |\--- Support: (d)eveloper, (m)ailing list, (u)senet, (n)one
Name      |
          \---- Development: (i)dea, (c)onstructions, (a)lpha, (b)eta,
                             (R)eleased, (M)ature, (S)tandard
```

plus a description of the module and why you think it should be included. If you hear nothing back, that means your module will probably appear on the modules list at the next update. Don't try subscribing to `modules@perl.org`; it's not another mailing list. Just have patience.

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...

61.2 AUTHOR

Simon Cozens, `simon@cpan.org`

61.3 SEE ALSO

perlmod, *perlmodlib*, *perlmodinstall*, *h2xs*, *strict*, *Carp*, *Exporter*, *perlpod*, *Test*, *ExtUtils::MakeMaker*, http://www.cpan.org/ , Ken Williams' tutorial on building your own module at http://mathforum.org/˜ken/perl_modules.html

Chapter 62

perlutil

Utilities packaged with the Perl distribution

62.1 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.

62.1.1 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 *perlpod* 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 and pod2latex

As well as these two, there are two other converters: *pod2html* will produce HTML pages from POD, and *pod2latex*, which produces LaTeX files.

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 *perldiag* - 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.

62.1.2 CONVERTORS

To help you convert legacy programs to Perl, we've included three conversion filters:

a2p

a2p converts *awk* scripts to Perl programs; for example, `a2p -F:` on the simple *awk* script `{print $2}` will produce a Perl program based around this code:

```
while (<>) {
    ($Fld1,$Fld2) = split(/[:\n]/, $_, 9999);
    print $Fld2;
}
```

s2p

Similarly, *s2p* converts *sed* scripts to Perl programs. *s2p* run on `s/foo/bar` will produce a Perl program based around this:

```
while (<>) {
    chomp;
    s/foo/bar/g;
    print if $printit;
}
```

find2perl

Finally, *find2perl* translates `find` commands to Perl equivalents which use the File::Find module. As an example, `find2perl . -user root -perm 4000 -print` produces the following callback subroutine for `File::Find`:

```
sub wanted {
    my ($dev,$ino,$mode,$nlink,$uid,$gid);
    (($dev,$ino,$mode,$nlink,$uid,$gid) = lstat($_)) &&
    $uid == $uid{'root'}) &&
    (($mode & 0777) == 04000);
    print("$name\n");
}
```

As well as these filters for converting other languages, the pl2pm utility will help you convert old-style Perl 4 libraries to new-style Perl5 modules.

62.1.3 Administration

libnetcfg

To display and change the libnet configuration run the libnetcfg command.

62.1.4 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.

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.

dprofpp

Perl comes with a profiler, the *Devel::DProf* module. The *dprofpp* utility analyzes the output of this profiler and tells you which subroutines are taking up the most run time. See *Devel::DProf* for more information.

perlcc

perlcc is the interface to the experimental Perl compiler suite.

62.1.5 SEE ALSO

perldoc, pod2man, *perlpod*, pod2html, pod2usage, podselect, podchecker, splain, *perldiag*, roffitall, a2p, s2p, find2perl, File::Find, pl2pm, perlbug, h2ph, c2ph, h2xs, dprofpp, *Devel::DProf*, perlcc

Chapter 63

perlcompile

Introduction to the Perl Compiler-Translator

63.1 DESCRIPTION

Perl has always had a compiler: your source is compiled into an internal form (a parse tree) which is then optimized before being run. Since version 5.005, Perl has shipped with a module capable of inspecting the optimized parse tree (`B`), and this has been used to write many useful utilities, including a module that lets you turn your Perl into C source code that can be compiled into a native executable.

The `B` module provides access to the parse tree, and other modules ("back ends") do things with the tree. Some write it out as bytecode, C source code, or a semi-human-readable text. Another traverses the parse tree to build a cross-reference of which subroutines, formats, and variables are used where. Another checks your code for dubious constructs. Yet another back end dumps the parse tree back out as Perl source, acting as a source code beautifier or deobfuscator.

Because its original purpose was to be a way to produce C code corresponding to a Perl program, and in turn a native executable, the `B` module and its associated back ends are known as "the compiler", even though they don't really compile anything. Different parts of the compiler are more accurately a "translator", or an "inspector", but people want Perl to have a "compiler option" not an "inspector gadget". What can you do?

This document covers the use of the Perl compiler: which modules it comprises, how to use the most important of the back end modules, what problems there are, and how to work around them.

63.1.1 Layout

The compiler back ends are in the `B::` hierarchy, and the front-end (the module that you, the user of the compiler, will sometimes interact with) is the O module. Some back ends (e.g., `B::C`) have programs (e.g., *perlcc*) to hide the modules' complexity.

Here are the important back ends to know about, with their status expressed as a number from 0 (outline for later implementation) to 10 (if there's a bug in it, we're very surprised):

B::Bytecode

Stores the parse tree in a machine-independent format, suitable for later reloading through the ByteLoader module. Status: 5 (some things work, some things don't, some things are untested).

B::C

Creates a C source file containing code to rebuild the parse tree and resume the interpreter. Status: 6 (many things work adequately, including programs using Tk).

B::CC

Creates a C source file corresponding to the run time code path in the parse tree. This is the closest to a Perl-to-C translator there is, but the code it generates is almost incomprehensible because it translates the parse tree into a giant switch structure that manipulates Perl structures. Eventual goal is to reduce (given sufficient type information in the Perl program) some of the Perl data structure manipulations into manipulations of C-level ints, floats, etc. Status: 5 (some things work, including uncomplicated Tk examples).

B::Lint

Complains if it finds dubious constructs in your source code. Status: 6 (it works adequately, but only has a very limited number of areas that it checks).

B::Deparse

Recreates the Perl source, making an attempt to format it coherently. Status: 8 (it works nicely, but a few obscure things are missing).

B::Xref

Reports on the declaration and use of subroutines and variables. Status: 8 (it works nicely, but still has a few lingering bugs).

63.2 Using The Back Ends

The following sections describe how to use the various compiler back ends. They're presented roughly in order of maturity, so that the most stable and proven back ends are described first, and the most experimental and incomplete back ends are described last.

The O module automatically enabled the **-c** flag to Perl, which prevents Perl from executing your code once it has been compiled. This is why all the back ends print:

```
  myperlprogram syntax OK
```

before producing any other output.

63.2.1 The Cross Referencing Back End

The cross referencing back end (B::Xref) produces a report on your program, breaking down declarations and uses of subroutines and variables (and formats) by file and subroutine. For instance, here's part of the report from the *pod2man* program that comes with Perl:

```
  Subroutine clear_noremap
    Package (lexical)
      $ready_to_print   i1069, 1079
    Package main
      $&                1086
      $.                1086
      $0                1086
      $1                1087
      $2                1085, 1085
      $3                1085, 1085
      $ARGV             1086
      %HTML_Escapes     1085, 1085
```

This shows the variables used in the subroutine `clear_noremap`. The variable `$ready_to_print` is a my() (lexical) variable, **i**ntroduced (first declared with my()) on line 1069, and used on line 1079. The variable `$&` from the main package is used on 1086, and so on.

A line number may be prefixed by a single letter:

i

Lexical variable introduced (declared with my()) for the first time.

&

Subroutine or method call.

s

Subroutine defined.

r

Format defined.

The most useful option the cross referencer has is to save the report to a separate file. For instance, to save the report on *myperlprogram* to the file *report*:

```
$ perl -MO=Xref,-oreport myperlprogram
```

63.2.2 The Decompiling Back End

The Deparse back end turns your Perl source back into Perl source. It can reformat along the way, making it useful as a de-obfuscator. The most basic way to use it is:

```
$ perl -MO=Deparse myperlprogram
```

You'll notice immediately that Perl has no idea of how to paragraph your code. You'll have to separate chunks of code from each other with newlines by hand. However, watch what it will do with one-liners:

```
$ perl -MO=Deparse -e '$op=shift||die "usage: $0
code [...]";chomp(@ARGV=<>)unless@ARGV; for(@ARGV){$was=$_;eval$op;
die$@ if$@; rename$was,$_ unless$was eq $_}'
-e syntax OK
$op = shift @ARGV || die("usage: $0 code [...]");
chomp(@ARGV = <ARGV>) unless @ARGV;
foreach $_ (@ARGV) {
    $was = $_;
    eval $op;
    die $@ if $@;
    rename $was, $_ unless $was eq $_;
}
```

The decompiler has several options for the code it generates. For instance, you can set the size of each indent from 4 (as above) to 2 with:

```
$ perl -MO=Deparse,-si2 myperlprogram
```

The **-p** option adds parentheses where normally they are omitted:

```
$ perl -MO=Deparse -e 'print "Hello, world\n"'
-e syntax OK
print "Hello, world\n";
$ perl -MO=Deparse,-p -e 'print "Hello, world\n"'
-e syntax OK
print("Hello, world\n");
```

See *B::Deparse* for more information on the formatting options.

63.2.3 The Lint Back End

The lint back end (B::Lint) inspects programs for poor style. One programmer's bad style is another programmer's useful tool, so options let you select what is complained about.

To run the style checker across your source code:

```
  $ perl -MO=Lint myperlprogram
```

To disable context checks and undefined subroutines:

```
  $ perl -MO=Lint,-context,-undefined-subs myperlprogram
```

See *B::Lint* for information on the options.

63.2.4 The Simple C Back End

This module saves the internal compiled state of your Perl program to a C source file, which can be turned into a native executable for that particular platform using a C compiler. The resulting program links against the Perl interpreter library, so it will not save you disk space (unless you build Perl with a shared library) or program size. It may, however, save you startup time.

The `perlcc` tool generates such executables by default.

```
  perlcc myperlprogram.pl
```

63.2.5 The Bytecode Back End

This back end is only useful if you also have a way to load and execute the bytecode that it produces. The ByteLoader module provides this functionality.

To turn a Perl program into executable byte code, you can use `perlcc` with the `-B` switch:

```
  perlcc -B myperlprogram.pl
```

The byte code is machine independent, so once you have a compiled module or program, it is as portable as Perl source (assuming that the user of the module or program has a modern-enough Perl interpreter to decode the byte code).

See **B::Bytecode** for information on options to control the optimization and nature of the code generated by the Bytecode module.

63.2.6 The Optimized C Back End

The optimized C back end will turn your Perl program's run time code-path into an equivalent (but optimized) C program that manipulates the Perl data structures directly. The program will still link against the Perl interpreter library, to allow for eval(), `s///e`, `require`, etc.

The `perlcc` tool generates such executables when using the -O switch. To compile a Perl program (ending in `.pl` or `.p`):

```
  perlcc -O myperlprogram.pl
```

To produce a shared library from a Perl module (ending in `.pm`):

```
  perlcc -O Myperlmodule.pm
```

For more information, see *perlcc* and *B::CC*.

63.3 Module List for the Compiler Suite

B

This module is the introspective ("reflective" in Java terms) module, which allows a Perl program to inspect its innards. The back end modules all use this module to gain access to the compiled parse tree. You, the user of a back end module, will not need to interact with B.

O

This module is the front-end to the compiler's back ends. Normally called something like this:

```
$ perl -MO=Deparse myperlprogram
```

This is like saying `use O 'Deparse'` in your Perl program.

B::Asmdata

This module is used by the B::Assembler module, which is in turn used by the B::Bytecode module, which stores a parse-tree as bytecode for later loading. It's not a back end itself, but rather a component of a back end.

B::Assembler

This module turns a parse-tree into data suitable for storing and later decoding back into a parse-tree. It's not a back end itself, but rather a component of a back end. It's used by the *assemble* program that produces bytecode.

B::Bblock

This module is used by the B::CC back end. It walks "basic blocks". A basic block is a series of operations which is known to execute from start to finish, with no possibility of branching or halting.

B::Bytecode

This module is a back end that generates bytecode from a program's parse tree. This bytecode is written to a file, from where it can later be reconstructed back into a parse tree. The goal is to do the expensive program compilation once, save the interpreter's state into a file, and then restore the state from the file when the program is to be executed. See §63.2.5 for details about usage.

B::C

This module writes out C code corresponding to the parse tree and other interpreter internal structures. You compile the corresponding C file, and get an executable file that will restore the internal structures and the Perl interpreter will begin running the program. See §63.2.4 for details about usage.

B::CC

This module writes out C code corresponding to your program's operations. Unlike the B::C module, which merely stores the interpreter and its state in a C program, the B::CC module makes a C program that does not involve the interpreter. As a consequence, programs translated into C by B::CC can execute faster than normal interpreted programs. See §63.2.6 for details about usage.

B::Concise

This module prints a concise (but complete) version of the Perl parse tree. Its output is more customizable than the one of B::Terse or B::Debug (and it can emulate them). This module useful for people who are writing their own back end, or who are learning about the Perl internals. It's not useful to the average programmer.

B::Debug

This module dumps the Perl parse tree in verbose detail to STDOUT. It's useful for people who are writing their own back end, or who are learning about the Perl internals. It's not useful to the average programmer.

B::Deparse

This module produces Perl source code from the compiled parse tree. It is useful in debugging and deconstructing other people's code, also as a pretty-printer for your own source. See §63.2.2 for details about usage.

B::Disassembler

This module turns bytecode back into a parse tree. It's not a back end itself, but rather a component of a back end. It's used by the *disassemble* program that comes with the bytecode.

B::Lint

This module inspects the compiled form of your source code for things which, while some people frown on them, aren't necessarily bad enough to justify a warning. For instance, use of an array in scalar context without explicitly saying `scalar(@array)` is something that Lint can identify. See §63.2.3 for details about usage.

B::Showlex

This module prints out the my() variables used in a function or a file. To get a list of the my() variables used in the subroutine mysub() defined in the file myperlprogram:

```
$ perl -MO=Showlex,mysub myperlprogram
```

To get a list of the my() variables used in the file myperlprogram:

```
$ perl -MO=Showlex myperlprogram
```

[BROKEN]

B::Stackobj

This module is used by the B::CC module. It's not a back end itself, but rather a component of a back end.

B::Stash

This module is used by the *perlcc* program, which compiles a module into an executable. B::Stash prints the symbol tables in use by a program, and is used to prevent B::CC from producing C code for the B::* and O modules. It's not a back end itself, but rather a component of a back end.

B::Terse

This module prints the contents of the parse tree, but without as much information as B::Debug. For comparison, `print "Hello, world."` produced 96 lines of output from B::Debug, but only 6 from B::Terse.

This module is useful for people who are writing their own back end, or who are learning about the Perl internals. It's not useful to the average programmer.

B::Xref

This module prints a report on where the variables, subroutines, and formats are defined and used within a program and the modules it loads. See §63.2.1 for details about usage.

63.4 KNOWN PROBLEMS

The simple C backend currently only saves typeglobs with alphanumeric names.

The optimized C backend outputs code for more modules than it should (e.g., DirHandle). It also has little hope of properly handling `goto LABEL` outside the running subroutine (`goto &sub` is okay). `goto LABEL` currently does not work at all in this backend. It also creates a huge initialization function that gives C compilers headaches. Splitting the initialization function gives better results. Other problems include: unsigned math does not work correctly; some opcodes are handled incorrectly by default opcode handling mechanism.

BEGIN{} blocks are executed while compiling your code. Any external state that is initialized in BEGIN{}, such as opening files, initiating database connections etc., do not behave properly. To work around this, Perl has an INIT{} block that corresponds to code being executed before your program begins running but after your program has finished being compiled. Execution order: BEGIN{}, (possible save of state through compiler back-end), INIT{}, program runs, END{}.

63.5 AUTHOR

This document was originally written by Nathan Torkington, and is now maintained by the perl5-porters mailing list *perl5-porters@perl.org*.

Chapter 64

perlfilter

Source Filters

64.1 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.

64.2 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
```

64.3 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. You could use the existing `-P` command line option to do this, but as it happens, the source filters distribution comes with a C preprocessor filter module called Filter::cpp. Let's use that instead.

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"0[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.

64.4 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.

64.5 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, 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 module.

64.6 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.

64.7 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
```

64.8 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
```

When the `DEBUG` environment variable exists, the filter ensures that Perl parses only the code between the `DEBUG_BEGIN` and `DEBUG_END` markers. 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.

64.9 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.

64.10 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.

64.11 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.

64.12 AUTHOR

Paul Marquess <Paul.Marquess@btinternet.com>

64.13 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.

Part IV

Internals and C Language Interface

Chapter 65

perlembed

How to embed perl in your C program

65.1 DESCRIPTION

65.1.1 PREAMBLE

Do you want to:

Use C from Perl?
: Read *perlxstut*, *perlxs*, *h2xs*, *perlguts*, and *perlapi*.

Use a Unix program from Perl?
: Read about back-quotes and about `system` and `exec` in *perlfunc*.

Use Perl from Perl?
: Read about do in *perlfunc* and eval in *perlfunc* and require in *perlfunc* and use in *perlfunc*.

Use C from C?
: Rethink your design.

Use Perl from C?
: Read on...

65.1.2 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

65.1.3 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, Adding a Perl interpreter to your C program, 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.

65.1.4 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, nonportable 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; since PERL_SYS_INIT3() may change `env`, it may be more appropriate to provide `env` as an argument to perl_parse().

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*.

65.1.5 Calling a Perl subroutine from your C program

To call individual Perl subroutines, you can use any of the **call_*** functions documented in *perlcall*. 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 *perlcall*.

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 Fiddling with the Perl stack from your C program.

65.1.6 Evaluating a Perl statement from your C program

Perl provides two API functions to evaluate pieces of Perl code. These are eval_sv in *perlapi* and 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 use in *perlfunc*, require in *perlfunc*, and do in *perlfunc* 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)
{
    STRLEN n_a;
    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", FALSE)));

    /** Treat $a as a float **/
    eval_pv("$a = 3.14; $a **= 2", TRUE);
    printf("a = %f\n", SvNV(get_sv("a", FALSE)));

    /** Treat $a as a string **/
    eval_pv("$a = 'rekcaH lreP rehtonA tsuJ'; $a = reverse($a);", TRUE);
    printf("a = %s\n", SvPV(get_sv("a", FALSE), n_a));

    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 *perlguts* 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'd expression. It is also possible and in most cases a better strategy to fetch the return value from *eval_pv()* instead. Example:

```
...
STRLEN n_a;
SV *val = eval_pv("reverse 'rekcaH lreP rehtonA tsuJ'", TRUE);
printf("%s\n", SvPV(val,n_a));
...
```

This way, we avoid namespace pollution by not creating global variables and we've simplified our code as well.

65.1.7 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.

```
int 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;
    STRLEN n_a;

    PUSHMARK(SP);
    eval_sv(sv, G_SCALAR);

    SPAGAIN;
    retval = POPs;
    PUTBACK;
    if (croak_on_error && SvTRUE(ERRSV))
       croak(SvPVx(ERRSV, n_a));

    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(1099, 0), *retval;
    STRLEN n_a;

    sv_setpvf(command, "my $string = '%s'; $string =~ %s",
             SvPV(string,n_a), 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(1099, 0), *retval;
    STRLEN n_a;

    sv_setpvf(command, "$string = '%s'; ($string =~ %s)",
             SvPV(*string,n_a), pattern);

    retval = my_eval_sv(command, TRUE);
    SvREFCNT_dec(command);

    *string = get_sv("string", FALSE);
    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
**/
```

```
I32 matches(SV *string, char *pattern, AV **match_list)
{
    SV *command = NEWSV(1099, 0);
    I32 num_matches;
    STRLEN n_a;

    sv_setpvf(command, "my $string = '%s'; @array = ($string =~ %s)",
             SvPV(string,n_a), pattern);

    my_eval_sv(command, TRUE);
    SvREFCNT_dec(command);

    *match_list = get_av("array", FALSE);
    num_matches = av_len(*match_list) + 1; /** assume $[ is 0 **/

    return num_matches;
}

main (int argc, char **argv, char **env)
{
    char *embedding[] = { "", "-e", "0" };
    AV *match_list;
    I32 num_matches, i;
    SV *text;
    STRLEN n_a;

    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(1099,0);
    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(*av_fetch(match_list, i, FALSE),n_a));
    printf("\n");

    /** Remove all vowels from text **/
    num_matches = substitute(&text, "s/[aeiou]//gi");
    if (num_matches) {
       printf("substitute: s/[aeiou]//gi...%d substitutions made.\n",
              num_matches);
       printf("Now text is: %s\n\n", SvPV(text,n_a));
    }

    /** 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.
```

65.1.8 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 *perlguts* and *perlapi*.

Then you'll need to know how to manipulate the Perl stack. That's described in *perlcall*.

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.
```

65.1.9 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 eval in *perlfunc*. 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 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 `AUTOLOAD`ed 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 my in *perlfunc* 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;
    STRLEN n_a;
    PERL_SYS_INIT3(&argc,&argv,&env);
    if((my_perl = perl_alloc()) == NULL) {
       fprintf(stderr, "no memory!");
       exit(1);
    }
    perl_construct(my_perl);

    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(ERRSV,n_a));
       }
    }

    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
```

65.1.10 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.

65.1.11 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.

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 */
/* 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
```

65.1.12 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*, *perlguts*, and *perlapi* for more details.

65.2 Embedding Perl under Win32

In general, all of the source code shown here should work unmodified under Windows.

However, there are some caveats about the command-line examples shown. For starters, backticks won't work under the Win32 native command shell. The ExtUtils::Embed kit on CPAN ships with a script called **genmake**, which generates a simple makefile to build a program from a single C source file. It can be used like this:

```
C:\ExtUtils-Embed\eg> perl genmake interp.c
C:\ExtUtils-Embed\eg> nmake
C:\ExtUtils-Embed\eg> interp -e "print qq{I'm embedded in Win32!\n}"
```

You may wish to use a more robust environment such as the Microsoft Developer Studio. In this case, run this to generate perlxsi.c:

```
perl -MExtUtils::Embed -e xsinit
```

Create a new project and Insert -> Files into Project: perlxsi.c, perl.lib, and your own source files, e.g. interp.c. Typically you'll find perl.lib in **C:\perl\lib\CORE**, if not, you should see the **CORE** directory relative to `perl -V:archlib`. The studio will also need this path so it knows where to find Perl include files. This path can be added via the Tools -> Options -> Directories menu. Finally, select Build -> Build interp.exe and you're ready to go.

65.3 Hiding Perl_

If you completely hide the short forms 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 Background and PERL_IMPLICIT_CONTEXT for the explanation of the `aTHX_`. in *perlguts*) 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.)

65.4 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.

65.5 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.

July 22, 1998

65.6 COPYRIGHT

Chapter 66

perldebguts

Guts of Perl debugging

66.1 DESCRIPTION

This is not the perldebug(1) manpage, 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.

66.2 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 *perlrun*, 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. `$^P` in *perlvar*):

- 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)`. Code assertions in regexes look like `(re_eval 19)`.

  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 $filename for `eval`ed strings looks like `(eval 34)` or `(re_eval 19)`.
- Each scalar `${"_<$filename"}` contains `"_<$filename"`. This is also the case for evaluated strings that contain subroutines, or which are currently being executed. The $filename for `eval`ed strings looks like `(eval 34)` or `(re_eval 19)`.
- After each `required` file is compiled, but before it is executed, `DB::postponed(*{"_<$filename"})` is called if the subroutine `DB::postponed` exists. Here, the $filename is the expanded name of the `required` file, as found in the values of %INC.

- After each subroutine `subname` is compiled, the existence of `$DB::postponed{subname}` is checked. If this key exists, `DB::postponed(subname)` is called if the `DB::postponed` subroutine also exists.
- A hash `%DB::sub` is maintained, whose keys are subroutine names and whose values have the form `filename:startline-endline`. `filename` has the form `(eval 34)` for subroutines defined inside `eval`s, or `(re_eval 19)` for those within regex code assertions.
- When the execution of your program reaches a point that can hold a breakpoint, the `DB::DB()` subroutine is called if any of the variables `$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.)

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.

66.2.1 Writing Your Own Debugger

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 Options in *perldebug*).

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"`.

Debugger customization functions

Some functions are provided to simplify customization.

- See Options in *perldebug* for description of options parsed by `DB::parse_options(string)` parses debugger options; see Options in *pperldebug* for a description of options recognized.
- `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 *perldebug*) are considered for internal use only, and as such are subject to change without notice.

66.3 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 use various values of `n` are shown to give you a feel for the difference between settings. Long those 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/Exporter.pm:171
         out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/Exporter.pm:171
         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.

66.4 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. It is not lexically scoped.

66.4.1 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 `BOL`, `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*)

66.4.2 Types of nodes

Here are the 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.

# Anchors:
BOL         no      Match "" at beginning of line.
MBOL        no      Same, assuming multiline.
SBOL        no      Same, assuming singleline.
EOS         no      Match "" at end of string.
EOL         no      Match "" at end of line.
MEOL        no      Same, assuming multiline.
SEOL        no      Same, assuming singleline.
BOUND       no      Match "" at any word boundary
BOUNDL      no      Match "" at any word boundary
NBOUND      no      Match "" at any word non-boundary
NBOUNDL     no      Match "" at any word non-boundary
GPOS        no      Matches where last m//g left off.

# [Special] alternatives
ANY         no      Match any one character (except newline).
SANY        no      Match any one character.
ANYOF       sv      Match character in (or not in) this class.
ALNUM       no      Match any alphanumeric character
ALNUML      no      Match any alphanumeric char in locale
NALNUM      no      Match any non-alphanumeric character
NALNUML     no      Match any non-alphanumeric char in locale
SPACE       no      Match any whitespace character
SPACEL      no      Match any whitespace char in locale
NSPACE      no      Match any non-whitespace character
NSPACEL     no      Match any non-whitespace char in locale
DIGIT       no      Match any numeric character
NDIGIT      no      Match any non-numeric character

# 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...

# BACK      Normal "next" pointers all implicitly point forward; BACK
#           exists to make loop structures possible.
# not used
BACK        no      Match "", "next" ptr points backward.

# Literals
EXACT       sv      Match this string (preceded by length).
EXACTF      sv      Match this string, folded (prec. by length).
EXACTFL     sv      Match this string, folded in locale (w/len).

# Do nothing
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.

# STAR,PLUS '?', and complex '*' and '+', are implemented as circular
#           BRANCH structures using BACK.  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    Match next-after-this simple thing
#                   {n,m} times, set parens.
CURLYM      no 2    Match 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.

# 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
REFFL       num 1   Match already matched string, folded in loc.

# grouping assertions
IFMATCH     off 1 2 Succeeds if the following matches.
UNLESSM     off 1 2 Fails if the following matches.
SUSPEND     off 1 1 "Independent" sub-regex.
IFTHEN      off 1 1 Switch, should be preceded by switcher .
GROUPP      num 1   Whether the group matched.

# Support for long regex
LONGJMP     off 1 1 Jump far away.
BRANCHJ     off 1 1 BRANCH with long offset.
```

```
    # The heavy worker
    EVAL        evl 1   Execute some Perl code.

    # 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.

    # 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.
```

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.

66.4.3 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.

66.5 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.

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 the $ENV{PERL_DEBUG_MSTATS}.

66.5.1 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, case 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 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.

66.5.2 Example of using -DL switch

(Note that -DL is obsolete since circa 5.6.0, and even before that Perl needed to be compiled with -DDEBUGGING.)

Below we show how to analyse memory usage by

```
do 'lib/auto/POSIX/autosplit.ix';
```

The file in question contains a header and 146 lines similar to

```
sub getcwd;
```

WARNING: The discussion below supposes 32-bit architecture. In newer releases of Perl, memory usage of the constructs discussed here is greatly improved, but the story discussed below is a real-life story. This story is mercilessly terse, and assumes rather more than cursory knowledge of Perl internals. Type space to continue, 'q' to quit. (Actually, you just want to skip to the next section.)

Here is the itemized list of Perl allocations performed during parsing of this file:

```
 !!! "after" at test.pl line 3.
    Id  subtot   4   8  12  16  20  24  28  32  36  40  48  56  64  72  80 80+
  0 02   13752   .   .   .   . 294   .   .   .   .   .   .   .   .   .   .   4
  0 54    5545   .   .   8 124  16   .   .   .   1   1   .   .   .   .   .   3
  5 05      32   .   .   .   .   .   .   .   1   .   .   .   .   .   .   .   .
  6 02    7152   .   .   .   .   .   .   .   .   .   . 149   .   .   .   .   .
  7 02    3600   .   .   .   .   . 150   .   .   .   .   .   .   .   .   .   .
  7 03      64   .  -1   .   1   .   .   2   .   .   .   .   .   .   .   .   .
  7 04    7056   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   7
  7 17   38404   .   .   .   .   .   .   .   1   .   . 442 149   .   . 147   .
  9 03    2078  17 249  32   .   .   .   .   2   .   .   .   .   .   .   .   .
```

To see this list, insert two `warn('!...')` statements around the call:

```
warn('!');
do 'lib/auto/POSIX/autosplit.ix';
warn('!!! "after"');
```

and run it with Perl's **-DL** option. The first warn() will print memory allocation info before parsing the file and will memorize the statistics at this point (we ignore what it prints). The second warn() prints increments with respect to these memorized data. This is the printout shown above.

Different *Id*s on the left correspond to different subsystems of the perl interpreter. They are just the first argument given to the perl memory allocation API named New(). To find what `9 03` means, just **grep** the perl source for `903`. You'll find it in *util.c*, function savepvn(). (I know, you wonder why we told you to **grep** and then gave away the answer. That's because grepping the source is good for the soul.) This function is used to store a copy of an existing chunk of memory. Using a C debugger, one can see that the function was called either directly from gv_init() or via sv_magic(), and that gv_init() is called from gv_fetchpv()–which was itself called from newSUB(). Please stop to catch your breath now.

NOTE: To reach this point in the debugger and skip the calls to savepvn() during the compilation of the main program, you should set a C breakpoint in Perl_warn(), continue until this point is reached, and *then* set a C breakpoint in Perl_savepvn(). Note that you may need to skip a handful of Perl_savepvn() calls that do not correspond to mass production of CVs (there are more `903` allocations than 146 similar lines of *lib/auto/POSIX/autosplit.ix*). Note also that `Perl_` prefixes are added by macroization code in perl header files to avoid conflicts with external libraries.

Anyway, we see that `903` ids correspond to creation of globs, twice per glob - for glob name, and glob stringification magic.

Here are explanations for other *Id*s above:

`717`

Creates bigger `XPV*` structures. In the case above, it creates 3 `AV`s per subroutine, one for a list of lexical variable names, one for a scratchpad (which contains lexical variables and `targets`), and one for the array of scratchpads needed for recursion.

It also creates a `GV` and a `CV` per subroutine, all called from start_subparse().

`002`

Creates a C array corresponding to the `AV` of scratchpads and the scratchpad itself. The first fake entry of this scratchpad is created though the subroutine itself is not defined yet.

It also creates C arrays to keep data for the stash. This is one HV, but it grows; thus, there are 4 big allocations: the big chunks are not freed, but are kept as additional arenas for `SV` allocations.

`054`

Creates a `HEK` for the name of the glob for the subroutine. This name is a key in a *stash*.

Big allocations with this *Id* correspond to allocations of new arenas to keep `HE`.

`602`

Creates a `GP` for the glob for the subroutine.

`702`

Creates the `MAGIC` for the glob for the subroutine.

`704`

Creates *arenas* which keep SVs.

66.5.3 -DL details

If Perl is run with **-DL** option, then warn()s that start with '!' behave specially. They print a list of *categories* of memory allocations, and statistics of allocations of different sizes for these categories.

If warn() string starts with

`!!!`

print changed categories only, print the differences in counts of allocations.

`!!`

print grown categories only; print the absolute values of counts, and totals.

`!`

print nonempty categories, print the absolute values of counts and totals.

66.5.4 Limitations of -DL statistics

If an extension or external library does not use the Perl API to allocate memory, such allocations are not counted.

66.6 SEE ALSO

perldebug, *perlguts*, *perlrun re*, and *Devel::DProf*.

Chapter 67

perlXStut

Tutorial for writing XSUBs

67.1 DESCRIPTION

This tutorial will educate the reader on the steps involved in creating a Perl extension. The reader is assumed to have access to *perlguts*, *perlapi* and *perlxs*.

This tutorial starts with very simple examples and becomes more complex, with each new example adding new features. Certain concepts may not be completely explained until later in the tutorial in order to slowly ease the reader into building extensions.

This tutorial was written from a Unix point of view. Where I know them to be otherwise different for other platforms (e.g. Win32), I will list them. If you find something that was missed, please let me know.

67.2 SPECIAL NOTES

67.2.1 make

This tutorial assumes that the make program that Perl is configured to use is called `make`. Instead of running "make" in the examples that follow, you may have to substitute whatever make program Perl has been configured to use. Running **perl -V:make** should tell you what it is.

67.2.2 Version caveat

When writing a Perl extension for general consumption, one should expect that the extension will be used with versions of Perl different from the version available on your machine. Since you are reading this document, the version of Perl on your machine is probably 5.005 or later, but the users of your extension may have more ancient versions.

To understand what kinds of incompatibilities one may expect, and in the rare case that the version of Perl on your machine is older than this document, see the section on "Troubleshooting these Examples" for more information.

If your extension uses some features of Perl which are not available on older releases of Perl, your users would appreciate an early meaningful warning. You would probably put this information into the *README* file, but nowadays installation of extensions may be performed automatically, guided by *CPAN.pm* module or other tools.

In MakeMaker-based installations, *Makefile.PL* provides the earliest opportunity to perform version checks. One can put something like this in *Makefile.PL* for this purpose:

```
eval { require 5.007 }
    or die <<EOD;
############
### This module uses frobnication framework which is not available before
### version 5.007 of Perl.  Upgrade your Perl before installing Kara::Mba.
############
EOD
```

67.2.3 Dynamic Loading versus Static Loading

It is commonly thought that if a system does not have the capability to dynamically load a library, you cannot build XSUBs. This is incorrect. You *can* build them, but you must link the XSUBs subroutines with the rest of Perl, creating a new executable. This situation is similar to Perl 4.

This tutorial can still be used on such a system. The XSUB build mechanism will check the system and build a dynamically-loadable library if possible, or else a static library and then, optionally, a new statically-linked executable with that static library linked in.

Should you wish to build a statically-linked executable on a system which can dynamically load libraries, you may, in all the following examples, where the command "`make`" with no arguments is executed, run the command "`make perl`" instead.

If you have generated such a statically-linked executable by choice, then instead of saying "`make test`", you should say "`make test_static`". On systems that cannot build dynamically-loadable libraries at all, simply saying "`make test`" is sufficient.

67.3 TUTORIAL

Now let’s go on with the show!

67.3.1 EXAMPLE 1

Our first extension will be very simple. When we call the routine in the extension, it will print out a well-known message and return.

Run "`h2xs -A -n Mytest`". This creates a directory named Mytest, possibly under ext/ if that directory exists in the current working directory. Several files will be created in the Mytest dir, including MANIFEST, Makefile.PL, Mytest.pm, Mytest.xs, test.pl, and Changes.

The MANIFEST file contains the names of all the files just created in the Mytest directory.

The file Makefile.PL should look something like this:

```
use ExtUtils::MakeMaker;
# See lib/ExtUtils/MakeMaker.pm for details of how to influence
# the contents of the Makefile that is written.
WriteMakefile(
    NAME         => 'Mytest',
    VERSION_FROM => 'Mytest.pm', # finds $VERSION
    LIBS         => [''],   # e.g., '-lm'
    DEFINE       => '',     # e.g., '-DHAVE_SOMETHING'
    INC          => '',     # e.g., '-I/usr/include/other'
);
```

The file Mytest.pm should start with something like this:

```
package Mytest;

use strict;
use warnings;

require Exporter;
require DynaLoader;
```

```
our @ISA = qw(Exporter DynaLoader);
# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.
our @EXPORT = qw(

);
our $VERSION = '0.01';

bootstrap Mytest $VERSION;

# Preloaded methods go here.

# Autoload methods go after __END__, and are processed by the autosplit program.

1;
__END__
# Below is the stub of documentation for your module. You better edit it!
```

The rest of the .pm file contains sample code for providing documentation for the extension.

Finally, the Mytest.xs file should look something like this:

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

MODULE = Mytest         PACKAGE = Mytest
```

Let's edit the .xs file by adding this to the end of the file:

```
void
hello()
    CODE:
        printf("Hello, world!\n");
```

It is okay for the lines starting at the "CODE:" line to not be indented. However, for readability purposes, it is suggested that you indent CODE: one level and the lines following one more level.

Now we'll run "`perl Makefile.PL`". This will create a real Makefile, which make needs. Its output looks something like:

```
% perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Mytest
%
```

Now, running make will produce output that looks something like this (some long lines have been shortened for clarity and some extraneous lines have been deleted):

```
% make
umask 0 && cp Mytest.pm ./blib/Mytest.pm
perl xsubpp -typemap typemap Mytest.xs >Mytest.tc && mv Mytest.tc Mytest.c
Please specify prototyping behavior for Mytest.xs (see perlxs manual)
cc -c Mytest.c
Running Mkbootstrap for Mytest ()
```

```
chmod 644 Mytest.bs
LD_RUN_PATH="" ld -o ./blib/PA-RISC1.1/auto/Mytest/Mytest.sl -b Mytest.o
chmod 755 ./blib/PA-RISC1.1/auto/Mytest/Mytest.sl
cp Mytest.bs ./blib/PA-RISC1.1/auto/Mytest/Mytest.bs
chmod 644 ./blib/PA-RISC1.1/auto/Mytest/Mytest.bs
Manifying ./blib/man3/Mytest.3
%
```

You can safely ignore the line about "prototyping behavior" - it is explained in the section "The PROTOTYPES: Keyword" in *perlxs*.

If you are on a Win32 system, and the build process fails with linker errors for functions in the C library, check if your Perl is configured to use PerlCRT (running **perl -V:libc** should show you if this is the case). If Perl is configured to use PerlCRT, you have to make sure PerlCRT.lib is copied to the same location that msvcrt.lib lives in, so that the compiler can find it on its own. msvcrt.lib is usually found in the Visual C compiler's lib directory (e.g. C:/DevStudio/VC/lib).

Perl has its own special way of easily writing test scripts, but for this example only, we'll create our own test script. Create a file called hello that looks like this:

```
#! /opt/perl5/bin/perl

use ExtUtils::testlib;

use Mytest;

Mytest::hello();
```

Now we make the script executable (`chmod +x hello`), run the script and we should see the following output:

```
% ./hello
Hello, world!
%
```

67.3.2 EXAMPLE 2

Now let's add to our extension a subroutine that will take a single numeric argument as input and return 0 if the number is even or 1 if the number is odd.

Add the following to the end of Mytest.xs:

```
int
is_even(input)
        int     input
    CODE:
        RETVAL = (input % 2 == 0);
    OUTPUT:
        RETVAL
```

There does not need to be white space at the start of the "`int input`" line, but it is useful for improving readability. Placing a semi-colon at the end of that line is also optional. Any amount and kind of white space may be placed between the "`int`" and "`input`".

Now re-run make to rebuild our new shared library.

Now perform the same steps as before, generating a Makefile from the Makefile.PL file, and running make.

In order to test that our extension works, we now need to look at the file test.pl. This file is set up to imitate the same kind of testing structure that Perl itself has. Within the test script, you perform a number of tests to confirm the behavior of the extension, printing "ok" when the test is correct, "not ok" when it is not. Change the print statement in the BEGIN block to print "1..4", and add the following code to the end of the file:

```
        print &Mytest::is_even(0) == 1 ? "ok 2" : "not ok 2", "\n";
        print &Mytest::is_even(1) == 0 ? "ok 3" : "not ok 3", "\n";
        print &Mytest::is_even(2) == 1 ? "ok 4" : "not ok 4", "\n";
```

We will be calling the test script through the command "`make test`". You should see output that looks something like this:

```
        % make test
        PERL_DL_NONLAZY=1 /opt/perl5.004/bin/perl (lots of -I arguments) test.pl
        1..4
        ok 1
        ok 2
        ok 3
        ok 4
        %
```

67.3.3 What has gone on?

The program h2xs is the starting point for creating extensions. In later examples we'll see how we can use h2xs to read header files and generate templates to connect to C routines.

h2xs creates a number of files in the extension directory. The file Makefile.PL is a perl script which will generate a true Makefile to build the extension. We'll take a closer look at it later.

The .pm and .xs files contain the meat of the extension. The .xs file holds the C routines that make up the extension. The .pm file contains routines that tell Perl how to load your extension.

Generating the Makefile and running `make` created a directory called blib (which stands for "build library") in the current working directory. This directory will contain the shared library that we will build. Once we have tested it, we can install it into its final location.

Invoking the test script via "`make test`" did something very important. It invoked perl with all those `-I` arguments so that it could find the various files that are part of the extension. It is *very* important that while you are still testing extensions that you use "`make test`". If you try to run the test script all by itself, you will get a fatal error. Another reason it is important to use "`make test`" to run your test script is that if you are testing an upgrade to an already-existing version, using "`make test`" insures that you will test your new extension, not the already-existing version.

When Perl sees a `use extension;`, it searches for a file with the same name as the `use`'d extension that has a .pm suffix. If that file cannot be found, Perl dies with a fatal error. The default search path is contained in the `@INC` array.

In our case, Mytest.pm tells perl that it will need the Exporter and Dynamic Loader extensions. It then sets the `@ISA` and `@EXPORT` arrays and the `$VERSION` scalar; finally it tells perl to bootstrap the module. Perl will call its dynamic loader routine (if there is one) and load the shared library.

The two arrays `@ISA` and `@EXPORT` are very important. The `@ISA` array contains a list of other packages in which to search for methods (or subroutines) that do not exist in the current package. This is usually only important for object-oriented extensions (which we will talk about much later), and so usually doesn't need to be modified.

The `@EXPORT` array tells Perl which of the extension's variables and subroutines should be placed into the calling package's namespace. Because you don't know if the user has already used your variable and subroutine names, it's vitally important to carefully select what to export. Do *not* export method or variable names *by default* without a good reason.

As a general rule, if the module is trying to be object-oriented then don't export anything. If it's just a collection of functions and variables, then you can export them via another array, called `@EXPORT_OK`. This array does not automatically place its subroutine and variable names into the namespace unless the user specifically requests that this be done.

See *perlmod* for more information.

The `$VERSION` variable is used to ensure that the .pm file and the shared library are "in sync" with each other. Any time you make changes to the .pm or .xs files, you should increment the value of this variable.

67.3.4 Writing good test scripts

The importance of writing good test scripts cannot be overemphasized. You should closely follow the "ok/not ok" style that Perl itself uses, so that it is very easy and unambiguous to determine the outcome of each test case. When you find and fix a bug, make sure you add a test case for it.

By running "`make test`", you ensure that your test.pl script runs and uses the correct version of your extension. If you have many test cases, you might want to copy Perl's test style. Create a directory named "t" in the extension's directory and append the suffix ".t" to the names of your test files. When you run "`make test`", all of these test files will be executed.

67.3.5 EXAMPLE 3

Our third extension will take one argument as its input, round off that value, and set the *argument* to the rounded value.

Add the following to the end of Mytest.xs:

```
void
round(arg)
        double  arg
    CODE:
        if (arg > 0.0) {
                arg = floor(arg + 0.5);
        } else if (arg < 0.0) {
                arg = ceil(arg - 0.5);
        } else {
                arg = 0.0;
        }
    OUTPUT:
        arg
```

Edit the Makefile.PL file so that the corresponding line looks like this:

```
'LIBS'      => ['-lm'],   # e.g., '-lm'
```

Generate the Makefile and run make. Change the BEGIN block to print "1..9" and add the following to test.pl:

```
$i = -1.5; &Mytest::round($i); print $i == -2.0 ? "ok 5" : "not ok 5", "\n";
$i = -1.1; &Mytest::round($i); print $i == -1.0 ? "ok 6" : "not ok 6", "\n";
$i = 0.0; &Mytest::round($i); print $i == 0.0 ? "ok 7" : "not ok 7", "\n";
$i = 0.5; &Mytest::round($i); print $i == 1.0 ? "ok 8" : "not ok 8", "\n";
$i = 1.2; &Mytest::round($i); print $i == 1.0 ? "ok 9" : "not ok 9", "\n";
```

Running "`make test`" should now print out that all nine tests are okay.

Notice that in these new test cases, the argument passed to round was a scalar variable. You might be wondering if you can round a constant or literal. To see what happens, temporarily add the following line to test.pl:

```
&Mytest::round(3);
```

Run "`make test`" and notice that Perl dies with a fatal error. Perl won't let you change the value of constants!

67.3.6 What's new here?

- We've made some changes to Makefile.PL. In this case, we've specified an extra library to be linked into the extension's shared library, the math library libm in this case. We'll talk later about how to write XSUBs that can call every routine in a library.
- The value of the function is not being passed back as the function's return value, but by changing the value of the variable that was passed into the function. You might have guessed that when you saw that the return value of round is of type "void".

67.3.7 Input and Output Parameters

You specify the parameters that will be passed into the XSUB on the line(s) after you declare the function's return value and name. Each input parameter line starts with optional white space, and may have an optional terminating semicolon.

The list of output parameters occurs at the very end of the function, just before after the OUTPUT: directive. The use of RETVAL tells Perl that you wish to send this value back as the return value of the XSUB function. In Example 3, we wanted the "return value" placed in the original variable which we passed in, so we listed it (and not RETVAL) in the OUTPUT: section.

67.3.8 The XSUBPP Program

The **xsubpp** program takes the XS code in the .xs file and translates it into C code, placing it in a file whose suffix is .c. The C code created makes heavy use of the C functions within Perl.

67.3.9 The TYPEMAP file

The **xsubpp** program uses rules to convert from Perl's data types (scalar, array, etc.) to C's data types (int, char, etc.). These rules are stored in the typemap file ($PERLLIB/ExtUtils/typemap). This file is split into three parts.

The first section maps various C data types to a name, which corresponds somewhat with the various Perl types. The second section contains C code which **xsubpp** uses to handle input parameters. The third section contains C code which **xsubpp** uses to handle output parameters.

Let's take a look at a portion of the .c file created for our extension. The file name is Mytest.c:

```
XS(XS_Mytest_round)
{
    dXSARGS;
    if (items != 1)
        croak("Usage: Mytest::round(arg)");
    {
        double  arg = (double)SvNV(ST(0));      /* XXXXX */
        if (arg > 0.0) {
                arg = floor(arg + 0.5);
        } else if (arg < 0.0) {
                arg = ceil(arg - 0.5);
        } else {
                arg = 0.0;
        }
        sv_setnv(ST(0), (double)arg);   /* XXXXX */
    }
    XSRETURN(1);
}
```

Notice the two lines commented with "XXXXX". If you check the first section of the typemap file, you'll see that doubles are of type T_DOUBLE. In the INPUT section, an argument that is T_DOUBLE is assigned to the variable arg by calling the routine SvNV on something, then casting it to double, then assigned to the variable arg. Similarly, in the OUTPUT section, once arg has its final value, it is passed to the sv_setnv function to be passed back to the calling subroutine. These two functions are explained in *perlguts*; we'll talk more later about what that "ST(0)" means in the section on the argument stack.

67.3.10 Warning about Output Arguments

In general, it's not a good idea to write extensions that modify their input parameters, as in Example 3. Instead, you should probably return multiple values in an array and let the caller handle them (we'll do this in a later example). However, in order to better accommodate calling pre-existing C routines, which often do modify their input parameters, this behavior is tolerated.

67.3.11 EXAMPLE 4

In this example, we'll now begin to write XSUBs that will interact with pre-defined C libraries. To begin with, we will build a small library of our own, then let h2xs write our .pm and .xs files for us.

Create a new directory called Mytest2 at the same level as the directory Mytest. In the Mytest2 directory, create another directory called mylib, and cd into that directory.

Here we'll create some files that will generate a test library. These will include a C source file and a header file. We'll also create a Makefile.PL in this directory. Then we'll make sure that running make at the Mytest2 level will automatically run this Makefile.PL file and the resulting Makefile.

In the mylib directory, create a file mylib.h that looks like this:

```
        #define TESTVAL 4

        extern double   foo(int, long, const char*);
```

Also create a file mylib.c that looks like this:

```
        #include <stdlib.h>
        #include "./mylib.h"

        double
        foo(int a, long b, const char *c)
        {
                return (a + b + atof(c) + TESTVAL);
        }
```

And finally create a file Makefile.PL that looks like this:

```
        use ExtUtils::MakeMaker;
        $Verbose = 1;
        WriteMakefile(
            NAME   => 'Mytest2::mylib',
            SKIP   => [qw(all static static_lib dynamic dynamic_lib)],
            clean  => {'FILES' => 'libmylib$(LIB_EXT)'},
        );

        sub MY::top_targets {
                '
        all :: static

        pure_all :: static

        static ::       libmylib$(LIB_EXT)

        libmylib$(LIB_EXT): $(O_FILES)
                $(AR) cr libmylib$(LIB_EXT) $(O_FILES)
                $(RANLIB) libmylib$(LIB_EXT)

        ';
        }
```

Make sure you use a tab and not spaces on the lines beginning with "$(AR)" and "$(RANLIB)". Make will not function properly if you use spaces. It has also been reported that the "cr" argument to $(AR) is unnecessary on Win32 systems.

We will now create the main top-level Mytest2 files. Change to the directory above Mytest2 and run the following command:

```
% h2xs -O -n Mytest2 ./Mytest2/mylib/mylib.h
```

This will print out a warning about overwriting Mytest2, but that’s okay. Our files are stored in Mytest2/mylib, and will be untouched.

The normal Makefile.PL that h2xs generates doesn’t know about the mylib directory. We need to tell it that there is a subdirectory and that we will be generating a library in it. Let’s add the argument MYEXTLIB to the WriteMakefile call so that it looks like this:

```
WriteMakefile(
    'NAME'      => 'Mytest2',
    'VERSION_FROM' => 'Mytest2.pm', # finds $VERSION
    'LIBS'      => [''],   # e.g., '-lm'
    'DEFINE'    => '',     # e.g., '-DHAVE_SOMETHING'
    'INC'       => '',     # e.g., '-I/usr/include/other'
    'MYEXTLIB' => 'mylib/libmylib$(LIB_EXT)',
);
```

and then at the end add a subroutine (which will override the pre-existing subroutine). Remember to use a tab character to indent the line beginning with "cd"!

```
sub MY::postamble {
'
$(MYEXTLIB): mylib/Makefile
        cd mylib && $(MAKE) $(PASSTHRU)
';
}
```

Let’s also fix the MANIFEST file so that it accurately reflects the contents of our extension. The single line that says "mylib" should be replaced by the following three lines:

```
mylib/Makefile.PL
mylib/mylib.c
mylib/mylib.h
```

To keep our namespace nice and unpolluted, edit the .pm file and change the variable `@EXPORT` to `@EXPORT_OK`. Finally, in the .xs file, edit the #include line to read:

```
#include "mylib/mylib.h"
```

And also add the following function definition to the end of the .xs file:

```
double
foo(a,b,c)
        int             a
        long            b
        const char *    c
    OUTPUT:
        RETVAL
```

Now we also need to create a typemap file because the default Perl doesn’t currently support the const char * type. Create a file called typemap in the Mytest2 directory and place the following in it:

```
const char *    T_PV
```

Now run perl on the top-level Makefile.PL. Notice that it also created a Makefile in the mylib directory. Run make and watch that it does cd into the mylib directory and run make in there as well.

Now edit the test.pl script and change the BEGIN block to print "1..4", and add the following lines to the end of the script:

```
print &Mytest2::foo(1, 2, "Hello, world!") == 7 ? "ok 2\n" : "not ok 2\n";
print &Mytest2::foo(1, 2, "0.0") == 7 ? "ok 3\n" : "not ok 3\n";
print abs(&Mytest2::foo(0, 0, "-3.4") - 0.6) <= 0.01 ? "ok 4\n" : "not ok 4\n";
```

(When dealing with floating-point comparisons, it is best to not check for equality, but rather that the difference between the expected and actual result is below a certain amount (called epsilon) which is 0.01 in this case)

Run "`make test`" and all should be well.

67.3.12 What has happened here?

Unlike previous examples, we've now run h2xs on a real include file. This has caused some extra goodies to appear in both the .pm and .xs files.

- In the .xs file, there's now a #include directive with the absolute path to the mylib.h header file. We changed this to a relative path so that we could move the extension directory if we wanted to.
- There's now some new C code that's been added to the .xs file. The purpose of the `constant` routine is to make the values that are #define'd in the header file accessible by the Perl script (by calling either `TESTVAL` or `&Mytest2::TESTVAL`). There's also some XS code to allow calls to the `constant` routine.
- The .pm file originally exported the name `TESTVAL` in the `@EXPORT` array. This could lead to name clashes. A good rule of thumb is that if the #define is only going to be used by the C routines themselves, and not by the user, they should be removed from the `@EXPORT` array. Alternately, if you don't mind using the "fully qualified name" of a variable, you could move most or all of the items from the `@EXPORT` array into the `@EXPORT_OK` array.
- If our include file had contained #include directives, these would not have been processed by h2xs. There is no good solution to this right now.
- We've also told Perl about the library that we built in the mylib subdirectory. That required only the addition of the `MYEXTLIB` variable to the WriteMakefile call and the replacement of the postamble subroutine to cd into the subdirectory and run make. The Makefile.PL for the library is a bit more complicated, but not excessively so. Again we replaced the postamble subroutine to insert our own code. This code simply specified that the library to be created here was a static archive library (as opposed to a dynamically loadable library) and provided the commands to build it.

67.3.13 Anatomy of .xs file

The .xs file of §67.3.11 contained some new elements. To understand the meaning of these elements, pay attention to the line which reads

```
MODULE = Mytest2                PACKAGE = Mytest2
```

Anything before this line is plain C code which describes which headers to include, and defines some convenience functions. No translations are performed on this part, apart from having embedded POD documentation skipped over (see *perlpod*) it goes into the generated output C file as is.

Anything after this line is the description of XSUB functions. These descriptions are translated by **xsubpp** into C code which implements these functions using Perl calling conventions, and which makes these functions visible from Perl interpreter.

Pay a special attention to the function `constant`. This name appears twice in the generated .xs file: once in the first part, as a static C function, then another time in the second part, when an XSUB interface to this static C function is defined.

This is quite typical for .xs files: usually the .xs file provides an interface to an existing C function. Then this C function is defined somewhere (either in an external library, or in the first part of .xs file), and a Perl interface to this function (i.e. "Perl glue") is described in the second part of .xs file. The situation in §67.3.1, §67.3.2, and §67.3.5, when all the work is done inside the "Perl glue", is somewhat of an exception rather than the rule.

67.3.14 Getting the fat out of XSUBs

In §67.3.11 the second part of .xs file contained the following description of an XSUB:

```
double
foo(a,b,c)
        int             a
        long            b
        const char *    c
    OUTPUT:
        RETVAL
```

Note that in contrast with §67.3.1, §67.3.2 and §67.3.5, this description does not contain the actual *code* for what is done is done during a call to Perl function foo(). To understand what is going on here, one can add a CODE section to this XSUB:

```
double
foo(a,b,c)
        int             a
        long            b
        const char *    c
    CODE:
        RETVAL = foo(a,b,c);
    OUTPUT:
        RETVAL
```

However, these two XSUBs provide almost identical generated C code: **xsubpp** compiler is smart enough to figure out the `CODE:` section from the first two lines of the description of XSUB. What about `OUTPUT:` section? In fact, that is absolutely the same! The `OUTPUT:` section can be removed as well, *as far as `CODE:` section or `PPCODE:` section* is not specified: **xsubpp** can see that it needs to generate a function call section, and will autogenerate the OUTPUT section too. Thus one can shortcut the XSUB to become:

```
double
foo(a,b,c)
        int             a
        long            b
        const char *    c
```

Can we do the same with an XSUB

```
int
is_even(input)
        int     input
    CODE:
        RETVAL = (input % 2 == 0);
    OUTPUT:
        RETVAL
```

of §67.3.2? To do this, one needs to define a C function `int is_even(int input)`. As we saw in Anatomy of .xs file, a proper place for this definition is in the first part of .xs file. In fact a C function

```
int
is_even(int arg)
{
        return (arg % 2 == 0);
}
```

is probably overkill for this. Something as simple as a `#define` will do too:

```
        #define is_even(arg)    ((arg) % 2 == 0)
```

After having this in the first part of .xs file, the "Perl glue" part becomes as simple as

```
        int
        is_even(input)
                int     input
```

This technique of separation of the glue part from the workhorse part has obvious tradeoffs: if you want to change a Perl interface, you need to change two places in your code. However, it removes a lot of clutter, and makes the workhorse part independent from idiosyncrasies of Perl calling convention. (In fact, there is nothing Perl-specific in the above description, a different version of **xsubpp** might have translated this to TCL glue or Python glue as well.)

67.3.15 More about XSUB arguments

With the completion of Example 4, we now have an easy way to simulate some real-life libraries whose interfaces may not be the cleanest in the world. We shall now continue with a discussion of the arguments passed to the **xsubpp** compiler.

When you specify arguments to routines in the .xs file, you are really passing three pieces of information for each argument listed. The first piece is the order of that argument relative to the others (first, second, etc). The second is the type of argument, and consists of the type declaration of the argument (e.g., int, char*, etc). The third piece is the calling convention for the argument in the call to the library function.

While Perl passes arguments to functions by reference, C passes arguments by value; to implement a C function which modifies data of one of the "arguments", the actual argument of this C function would be a pointer to the data. Thus two C functions with declarations

```
        int string_length(char *s);
        int upper_case_char(char *cp);
```

may have completely different semantics: the first one may inspect an array of chars pointed by s, and the second one may immediately dereference `cp` and manipulate `*cp` only (using the return value as, say, a success indicator). From Perl one would use these functions in a completely different manner.

One conveys this info to **xsubpp** by replacing * before the argument by `&`. `&` means that the argument should be passed to a library function by its address. The above two function may be XSUB-ified as

```
        int
        string_length(s)
                char *  s

        int
        upper_case_char(cp)
                char    &cp
```

For example, consider:

```
        int
        foo(a,b)
                char    &a
                char *  b
```

The first Perl argument to this function would be treated as a char and assigned to the variable a, and its address would be passed into the function foo. The second Perl argument would be treated as a string pointer and assigned to the variable b. The *value* of b would be passed into the function foo. The actual call to the function foo that **xsubpp** generates would look like this:

```
foo(&a, b);
```

xsubpp will parse the following function argument lists identically:

```
char    &a
char&a
char    & a
```

However, to help ease understanding, it is suggested that you place a "&" next to the variable name and away from the variable type), and place a "*" near the variable type, but away from the variable name (as in the call to foo above). By doing so, it is easy to understand exactly what will be passed to the C function – it will be whatever is in the "last column".

You should take great pains to try to pass the function the type of variable it wants, when possible. It will save you a lot of trouble in the long run.

67.3.16 The Argument Stack

If we look at any of the C code generated by any of the examples except example 1, you will notice a number of references to ST(n), where n is usually 0. "ST" is actually a macro that points to the n'th argument on the argument stack. ST(0) is thus the first argument on the stack and therefore the first argument passed to the XSUB, ST(1) is the second argument, and so on.

When you list the arguments to the XSUB in the .xs file, that tells **xsubpp** which argument corresponds to which of the argument stack (i.e., the first one listed is the first argument, and so on). You invite disaster if you do not list them in the same order as the function expects them.

The actual values on the argument stack are pointers to the values passed in. When an argument is listed as being an OUTPUT value, its corresponding value on the stack (i.e., ST(0) if it was the first argument) is changed. You can verify this by looking at the C code generated for Example 3. The code for the round() XSUB routine contains lines that look like this:

```
double  arg = (double)SvNV(ST(0));
/* Round the contents of the variable arg */
sv_setnv(ST(0), (double)arg);
```

The arg variable is initially set by taking the value from ST(0), then is stored back into ST(0) at the end of the routine.

XSUBs are also allowed to return lists, not just scalars. This must be done by manipulating stack values ST(0), ST(1), etc, in a subtly different way. See *perlxs* for details.

XSUBs are also allowed to avoid automatic conversion of Perl function arguments to C function arguments. See *perlxs* for details. Some people prefer manual conversion by inspecting `ST(i)` even in the cases when automatic conversion will do, arguing that this makes the logic of an XSUB call clearer. Compare with §67.3.14 for a similar tradeoff of a complete separation of "Perl glue" and "workhorse" parts of an XSUB.

While experts may argue about these idioms, a novice to Perl guts may prefer a way which is as little Perl-guts-specific as possible, meaning automatic conversion and automatic call generation, as in §67.3.14. This approach has the additional benefit of protecting the XSUB writer from future changes to the Perl API.

67.3.17 Extending your Extension

Sometimes you might want to provide some extra methods or subroutines to assist in making the interface between Perl and your extension simpler or easier to understand. These routines should live in the .pm file. Whether they are automatically loaded when the extension itself is loaded or only loaded when called depends on where in the .pm file the subroutine definition is placed. You can also consult *AutoLoader* for an alternate way to store and load your extra subroutines.

67.3.18 Documenting your Extension

There is absolutely no excuse for not documenting your extension. Documentation belongs in the .pm file. This file will be fed to pod2man, and the embedded documentation will be converted to the manpage format, then placed in the blib directory. It will be copied to Perl's manpage directory when the extension is installed.

You may intersperse documentation and Perl code within the .pm file. In fact, if you want to use method autoloading, you must do this, as the comment inside the .pm file explains.

See *perlpod* for more information about the pod format.

67.3.19 Installing your Extension

Once your extension is complete and passes all its tests, installing it is quite simple: you simply run "make install". You will either need to have write permission into the directories where Perl is installed, or ask your system administrator to run the make for you.

Alternately, you can specify the exact directory to place the extension's files by placing a "PREFIX=/destination/directory" after the make install. (or in between the make and install if you have a brain-dead version of make). This can be very useful if you are building an extension that will eventually be distributed to multiple systems. You can then just archive the files in the destination directory and distribute them to your destination systems.

67.3.20 EXAMPLE 5

In this example, we'll do some more work with the argument stack. The previous examples have all returned only a single value. We'll now create an extension that returns an array.

This extension is very Unix-oriented (struct statfs and the statfs system call). If you are not running on a Unix system, you can substitute for statfs any other function that returns multiple values, you can hard-code values to be returned to the caller (although this will be a bit harder to test the error case), or you can simply not do this example. If you change the XSUB, be sure to fix the test cases to match the changes.

Return to the Mytest directory and add the following code to the end of Mytest.xs:

```
void
statfs(path)
        char *  path
    INIT:
        int i;
        struct statfs buf;

    PPCODE:
        i = statfs(path, &buf);
        if (i == 0) {
                XPUSHs(sv_2mortal(newSVnv(buf.f_bavail)));
                XPUSHs(sv_2mortal(newSVnv(buf.f_bfree)));
                XPUSHs(sv_2mortal(newSVnv(buf.f_blocks)));
                XPUSHs(sv_2mortal(newSVnv(buf.f_bsize)));
                XPUSHs(sv_2mortal(newSVnv(buf.f_ffree)));
                XPUSHs(sv_2mortal(newSVnv(buf.f_files)));
                XPUSHs(sv_2mortal(newSVnv(buf.f_type)));
                XPUSHs(sv_2mortal(newSVnv(buf.f_fsid[0])));
                XPUSHs(sv_2mortal(newSVnv(buf.f_fsid[1])));
        } else {
                XPUSHs(sv_2mortal(newSVnv(errno)));
        }
```

You'll also need to add the following code to the top of the .xs file, just after the include of "XSUB.h":

```
        #include <sys/vfs.h>
```

Also add the following code segment to test.pl while incrementing the "1..9" string in the BEGIN block to "1..11":

```
        @a = &Mytest::statfs("/blech");
        print ((scalar(@a) == 1 && $a[0] == 2) ? "ok 10\n" : "not ok 10\n");
        @a = &Mytest::statfs("/");
        print scalar(@a) == 9 ? "ok 11\n" : "not ok 11\n";
```

67.3.21 New Things in this Example

This example added quite a few new concepts. We'll take them one at a time.

- The INIT: directive contains code that will be placed immediately after the argument stack is decoded. C does not allow variable declarations at arbitrary locations inside a function, so this is usually the best way to declare local variables needed by the XSUB. (Alternatively, one could put the whole `PPCODE:` section into braces, and put these declarations on top.)
- This routine also returns a different number of arguments depending on the success or failure of the call to statfs. If there is an error, the error number is returned as a single-element array. If the call is successful, then a 9-element array is returned. Since only one argument is passed into this function, we need room on the stack to hold the 9 values which may be returned.

  We do this by using the PPCODE: directive, rather than the CODE: directive. This tells **xsubpp** that we will be managing the return values that will be put on the argument stack by ourselves.
- When we want to place values to be returned to the caller onto the stack, we use the series of macros that begin with "XPUSH". There are five different versions, for placing integers, unsigned integers, doubles, strings, and Perl scalars on the stack. In our example, we placed a Perl scalar onto the stack. (In fact this is the only macro which can be used to return multiple values.)

  The XPUSH* macros will automatically extend the return stack to prevent it from being overrun. You push values onto the stack in the order you want them seen by the calling program.
- The values pushed onto the return stack of the XSUB are actually mortal SV's. They are made mortal so that once the values are copied by the calling program, the SV's that held the returned values can be deallocated. If they were not mortal, then they would continue to exist after the XSUB routine returned, but would not be accessible. This is a memory leak.
- If we were interested in performance, not in code compactness, in the success branch we would not use `XPUSHs` macros, but `PUSHs` macros, and would pre-extend the stack before pushing the return values:

  ```
          EXTEND(SP, 9);
  ```

  The tradeoff is that one needs to calculate the number of return values in advance (though overextending the stack will not typically hurt anything but memory consumption).

  Similarly, in the failure branch we could use `PUSHs` *without* extending the stack: the Perl function reference comes to an XSUB on the stack, thus the stack is *always* large enough to take one return value.

67.3.22 EXAMPLE 6

In this example, we will accept a reference to an array as an input parameter, and return a reference to an array of hashes. This will demonstrate manipulation of complex Perl data types from an XSUB.

This extension is somewhat contrived. It is based on the code in the previous example. It calls the statfs function multiple times, accepting a reference to an array of filenames as input, and returning a reference to an array of hashes containing the data for each of the filesystems.

Return to the Mytest directory and add the following code to the end of Mytest.xs:

```
SV *
multi_statfs(paths)
        SV * paths
    INIT:
        AV * results;
        I32 numpaths = 0;
        int i, n;
        struct statfs buf;

        if ((!SvROK(paths))
            || (SvTYPE(SvRV(paths)) != SVt_PVAV)
            || ((numpaths = av_len((AV *)SvRV(paths))) < 0))
        {
            XSRETURN_UNDEF;
        }
        results = (AV *)sv_2mortal((SV *)newAV());
    CODE:
        for (n = 0; n <= numpaths; n++) {
            HV * rh;
            STRLEN l;
            char * fn = SvPV(*av_fetch((AV *)SvRV(paths), n, 0), l);

            i = statfs(fn, &buf);
            if (i != 0) {
                av_push(results, newSVnv(errno));
                continue;
            }

            rh = (HV *)sv_2mortal((SV *)newHV());

            hv_store(rh, "f_bavail", 8, newSVnv(buf.f_bavail), 0);
            hv_store(rh, "f_bfree",  7, newSVnv(buf.f_bfree),  0);
            hv_store(rh, "f_blocks", 8, newSVnv(buf.f_blocks), 0);
            hv_store(rh, "f_bsize",  7, newSVnv(buf.f_bsize),  0);
            hv_store(rh, "f_ffree",  7, newSVnv(buf.f_ffree),  0);
            hv_store(rh, "f_files",  7, newSVnv(buf.f_files),  0);
            hv_store(rh, "f_type",   6, newSVnv(buf.f_type),   0);

            av_push(results, newRV((SV *)rh));
        }
        RETVAL = newRV((SV *)results);
    OUTPUT:
        RETVAL
```

And add the following code to test.pl, while incrementing the "1..11" string in the BEGIN block to "1..13":

```
$results = Mytest::multi_statfs([ '/', '/blech' ]);
print ((ref $results->[0]) ? "ok 12\n" : "not ok 12\n");
print ((! ref $results->[1]) ? "ok 13\n" : "not ok 13\n");
```

67.3.23 New Things in this Example

There are a number of new concepts introduced here, described below:

- This function does not use a typemap. Instead, we declare it as accepting one SV* (scalar) parameter, and returning an SV* value, and we take care of populating these scalars within the code. Because we are only returning one value, we don't need a `PPCODE:` directive - instead, we use `CODE:` and `OUTPUT:` directives.
- When dealing with references, it is important to handle them with caution. The `INIT:` block first checks that `SvROK` returns true, which indicates that paths is a valid reference. It then verifies that the object referenced by paths is an array, using `SvRV` to dereference paths, and `SvTYPE` to discover its type. As an added test, it checks that the array referenced by paths is non-empty, using the `av_len` function (which returns -1 if the array is empty). The XSRETURN_UNDEF macro is used to abort the XSUB and return the undefined value whenever all three of these conditions are not met.
- We manipulate several arrays in this XSUB. Note that an array is represented internally by an AV* pointer. The functions and macros for manipulating arrays are similar to the functions in Perl: `av_len` returns the highest index in an AV*, much like $#array; `av_fetch` fetches a single scalar value from an array, given its index; `av_push` pushes a scalar value onto the end of the array, automatically extending the array as necessary.

  Specifically, we read pathnames one at a time from the input array, and store the results in an output array (results) in the same order. If statfs fails, the element pushed onto the return array is the value of errno after the failure. If statfs succeeds, though, the value pushed onto the return array is a reference to a hash containing some of the information in the statfs structure.

  As with the return stack, it would be possible (and a small performance win) to pre-extend the return array before pushing data into it, since we know how many elements we will return:

  ```
          av_extend(results, numpaths);
  ```
- We are performing only one hash operation in this function, which is storing a new scalar under a key using `hv_store`. A hash is represented by an HV* pointer. Like arrays, the functions for manipulating hashes from an XSUB mirror the functionality available from Perl. See *perlguts* and *perlapi* for details.
- To create a reference, we use the `newRV` function. Note that you can cast an AV* or an HV* to type SV* in this case (and many others). This allows you to take references to arrays, hashes and scalars with the same function. Conversely, the `SvRV` function always returns an SV*, which may need to be cast to the appropriate type if it is something other than a scalar (check with `SvTYPE`).
- At this point, xsubpp is doing very little work - the differences between Mytest.xs and Mytest.c are minimal.

67.3.24 EXAMPLE 7 (Coming Soon)

XPUSH args AND set RETVAL AND assign return value to array

67.3.25 EXAMPLE 8 (Coming Soon)

Setting $!

67.3.26 EXAMPLE 9 Passing open files to XSes

You would think passing files to an XS is difficult, with all the typeglobs and stuff. Well, it isn't.

Suppose that for some strange reason we need a wrapper around the standard C library function `fputs()`. This is all we need:

```
        #define PERLIO_NOT_STDIO 0
        #include "EXTERN.h"
        #include "perl.h"
        #include "XSUB.h"

        #include <stdio.h>
```

```
        int
        fputs(s, stream)
                char *          s
                FILE *          stream
```

The real work is done in the standard typemap.

But you loose all the fine stuff done by the perlio layers. This calls the stdio function `fputs()`, which knows nothing about them.

The standard typemap offers three variants of PerlIO *: `InputStream` (T_IN), `InOutStream` (T_INOUT) and `OutputStream` (T_OUT). A bare `PerlIO *` is considered a T_INOUT. If it matters in your code (see below for why it might) #define or typedef one of the specific names and use that as the argument or result type in your XS file.

The standard typemap does not contain PerlIO * before perl 5.7, but it has the three stream variants. Using a PerlIO * directly is not backwards compatible unless you provide your own typemap.

For streams coming *from* perl the main difference is that `OutputStream` will get the output PerlIO * - which may make a difference on a socket. Like in our example...

For streams being handed *to* perl a new file handle is created (i.e. a reference to a new glob) and associated with the PerlIO * provided. If the read/write state of the PerlIO * is not correct then you may get errors or warnings from when the file handle is used. So if you opened the PerlIO * as "w" it should really be an `OutputStream` if open as "r" it should be an `InputStream`.

Now, suppose you want to use perlio layers in your XS. We'll use the perlio `PerlIO_puts()` function as an example.

In the C part of the XS file (above the first MODULE line) you have

```
        #define OutputStream    PerlIO *
    or
        typedef PerlIO *        OutputStream;
```

And this is the XS code:

```
        int
        perlioputs(s, stream)
                char *          s
                OutputStream    stream
        CODE:
                RETVAL = PerlIO_puts(stream, s);
        OUTPUT:
                RETVAL
```

We have to use a `CODE` section because `PerlIO_puts()` has the arguments reversed compared to `fputs()`, and we want to keep the arguments the same.

Wanting to explore this thoroughly, we want to use the stdio `fputs()` on a PerlIO *. This means we have to ask the perlio system for a stdio `FILE *`:

```
        int
        perliofputs(s, stream)
                char *          s
                OutputStream    stream
        PREINIT:
                FILE *fp = PerlIO_findFILE(stream);
        CODE:
                if (fp != (FILE*) 0) {
                        RETVAL = fputs(s, fp);
                } else {
                        RETVAL = -1;
                }
        OUTPUT:
                RETVAL
```

Note: `PerlIO_findFILE()` will search the layers for a stdio layer. If it can't find one, it will call `PerlIO_exportFILE()` to generate a new stdio `FILE`. Please only call `PerlIO_exportFILE()` if you want a *new* `FILE`. It will generate one on each call and push a new stdio layer. So don't call it repeatedly on the same file. `PerlIO()`_findFILE will retrieve the stdio layer once it has been generated by `PerlIO_exportFILE()`.

This applies to the perlio system only. For versions before 5.7, `PerlIO_exportFILE()` is equivalent to `PerlIO_findFILE()`.

67.3.27 Troubleshooting these Examples

As mentioned at the top of this document, if you are having problems with these example extensions, you might see if any of these help you.

- In versions of 5.002 prior to the gamma version, the test script in Example 1 will not function properly. You need to change the "use lib" line to read:

```
        use lib './blib';
```

- In versions of 5.002 prior to version 5.002b1h, the test.pl file was not automatically created by h2xs. This means that you cannot say "make test" to run the test script. You will need to add the following line before the "use extension" statement:

```
        use lib './blib';
```

- In versions 5.000 and 5.001, instead of using the above line, you will need to use the following line:

```
        BEGIN { unshift(@INC, "./blib") }
```

- This document assumes that the executable named "perl" is Perl version 5. Some systems may have installed Perl version 5 as "perl5".

67.4 See also

For more information, consult *perlguts*, *perlapi*, *perlxs*, *perlmod*, and *perlpod*.

67.5 Author

Jeff Okamoto <*okamoto@corp.hp.com*>

Reviewed and assisted by Dean Roehrich, Ilya Zakharevich, Andreas Koenig, and Tim Bunce.

PerlIO material contributed by Lupe Christoph, with some clarification by Nick Ing-Simmons.

67.5.1 Last Changed

2002/05/08

Chapter 68

perlxs

XS language reference manual

68.1 DESCRIPTION

68.1.1 Introduction

XS is an interface description file format used to create an extension interface between Perl and C code (or a C library) which one wishes to use with Perl. The XS interface is combined with the library to create a new library which can then be either dynamically loaded or statically linked into perl. The XS interface description is written in the XS language and is the core component of the Perl extension interface.

An **XSUB** forms the basic unit of the XS interface. After compilation by the **xsubpp** compiler, each XSUB amounts to a C function definition which will provide the glue between Perl calling conventions and C calling conventions.

The glue code pulls the arguments from the Perl stack, converts these Perl values to the formats expected by a C function, call this C function, transfers the return values of the C function back to Perl. Return values here may be a conventional C return value or any C function arguments that may serve as output parameters. These return values may be passed back to Perl either by putting them on the Perl stack, or by modifying the arguments supplied from the Perl side.

The above is a somewhat simplified view of what really happens. Since Perl allows more flexible calling conventions than C, XSUBs may do much more in practice, such as checking input parameters for validity, throwing exceptions (or returning undef/empty list) if the return value from the C function indicates failure, calling different C functions based on numbers and types of the arguments, providing an object-oriented interface, etc.

Of course, one could write such glue code directly in C. However, this would be a tedious task, especially if one needs to write glue for multiple C functions, and/or one is not familiar enough with the Perl stack discipline and other such arcana. XS comes to the rescue here: instead of writing this glue C code in long-hand, one can write a more concise short-hand *description* of what should be done by the glue, and let the XS compiler **xsubpp** handle the rest.

The XS language allows one to describe the mapping between how the C routine is used, and how the corresponding Perl routine is used. It also allows creation of Perl routines which are directly translated to C code and which are not related to a pre-existing C function. In cases when the C interface coincides with the Perl interface, the XSUB declaration is almost identical to a declaration of a C function (in K&R style). In such circumstances, there is another tool called `h2xs` that is able to translate an entire C header file into a corresponding XS file that will provide glue to the functions/macros described in the header file.

The XS compiler is called **xsubpp**. This compiler creates the constructs necessary to let an XSUB manipulate Perl values, and creates the glue necessary to let Perl call the XSUB. The compiler uses **typemaps** to determine how to map C function parameters and output values to Perl values and back. The default typemap (which comes with Perl) handles many common C types. A supplementary typemap may also be needed to handle any special structures and types for the library being linked.

A file in XS format starts with a C language section which goes until the first `MODULE =` directive. Other XS directives and XSUB definitions may follow this line. The "language" used in this part of the file is usually referred to as the XS language. **xsubpp** recognizes and skips POD (see *perlpod*) in both the C and XS language sections, which allows the XS file to contain embedded documentation.

See *perlxstut* for a tutorial on the whole extension creation process.

Note: For some extensions, Dave Beazley's SWIG system may provide a significantly more convenient mechanism for creating the extension glue code. See http://www.swig.org/ for more information.

68.1.2 On The Road

Many of the examples which follow will concentrate on creating an interface between Perl and the ONC+ RPC bind library functions. The rpcb_gettime() function is used to demonstrate many features of the XS language. This function has two parameters; the first is an input parameter and the second is an output parameter. The function also returns a status value.

```
        bool_t rpcb_gettime(const char *host, time_t *timep);
```

From C this function will be called with the following statements.

```
     #include <rpc/rpc.h>
     bool_t status;
     time_t timep;
     status = rpcb_gettime( "localhost", &timep );
```

If an XSUB is created to offer a direct translation between this function and Perl, then this XSUB will be used from Perl with the following code. The $status and $timep variables will contain the output of the function.

```
     use RPC;
     $status = rpcb_gettime( "localhost", $timep );
```

The following XS file shows an XS subroutine, or XSUB, which demonstrates one possible interface to the rpcb_gettime() function. This XSUB represents a direct translation between C and Perl and so preserves the interface even from Perl. This XSUB will be invoked from Perl with the usage shown above. Note that the first three #include statements, for `EXTERN.h`, `perl.h`, and `XSUB.h`, will always be present at the beginning of an XS file. This approach and others will be expanded later in this document.

```
     #include "EXTERN.h"
     #include "perl.h"
     #include "XSUB.h"
     #include <rpc/rpc.h>

     MODULE = RPC  PACKAGE = RPC

     bool_t
     rpcb_gettime(host,timep)
          char *host
          time_t &timep
        OUTPUT:
          timep
```

Any extension to Perl, including those containing XSUBs, should have a Perl module to serve as the bootstrap which pulls the extension into Perl. This module will export the extension's functions and variables to the Perl program and will cause the extension's XSUBs to be linked into Perl. The following module will be used for most of the examples in this document and should be used from Perl with the `use` command as shown earlier. Perl modules are explained in more detail later in this document.

```
package RPC;

require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
@EXPORT = qw( rpcb_gettime );

bootstrap RPC;
1;
```

Throughout this document a variety of interfaces to the rpcb_gettime() XSUB will be explored. The XSUBs will take their parameters in different orders or will take different numbers of parameters. In each case the XSUB is an abstraction between Perl and the real C rpcb_gettime() function, and the XSUB must always ensure that the real rpcb_gettime() function is called with the correct parameters. This abstraction will allow the programmer to create a more Perl-like interface to the C function.

68.1.3 The Anatomy of an XSUB

The simplest XSUBs consist of 3 parts: a description of the return value, the name of the XSUB routine and the names of its arguments, and a description of types or formats of the arguments.

The following XSUB allows a Perl program to access a C library function called sin(). The XSUB will imitate the C function which takes a single argument and returns a single value.

```
double
sin(x)
  double x
```

Optionally, one can merge the description of types and the list of argument names, rewriting this as

```
double
sin(double x)
```

This makes this XSUB look similar to an ANSI C declaration. An optional semicolon is allowed after the argument list, as in

```
double
sin(double x);
```

Parameters with C pointer types can have different semantic: C functions with similar declarations

```
bool string_looks_as_a_number(char *s);
bool make_char_uppercase(char *c);
```

are used in absolutely incompatible manner. Parameters to these functions could be described **xsubpp** like this:

```
char *  s
char    &c
```

Both these XS declarations correspond to the `char*` C type, but they have different semantics, see §68.1.40.

It is convenient to think that the indirection operator * should be considered as a part of the type and the address operator & should be considered part of the variable. See §68.1.45 for more info about handling qualifiers and unary operators in C types.

The function name and the return type must be placed on separate lines and should be flush left-adjusted.

```
  INCORRECT                         CORRECT

  double sin(x)                     double
    double x                        sin(x)
                                      double x
```

The rest of the function description may be indented or left-adjusted. The following example shows a function with its body left-adjusted. Most examples in this document will indent the body for better readability.

```
  CORRECT

  double
  sin(x)
  double x
```

More complicated XSUBs may contain many other sections. Each section of an XSUB starts with the corresponding keyword, such as INIT: or CLEANUP:. However, the first two lines of an XSUB always contain the same data: descriptions of the return type and the names of the function and its parameters. Whatever immediately follows these is considered to be an INPUT: section unless explicitly marked with another keyword. (See The INPUT: Keyword.)

An XSUB section continues until another section-start keyword is found.

68.1.4 The Argument Stack

The Perl argument stack is used to store the values which are sent as parameters to the XSUB and to store the XSUB's return value(s). In reality all Perl functions (including non-XSUB ones) keep their values on this stack all the same time, each limited to its own range of positions on the stack. In this document the first position on that stack which belongs to the active function will be referred to as position 0 for that function.

XSUBs refer to their stack arguments with the macro **ST(x)**, where *x* refers to a position in this XSUB's part of the stack. Position 0 for that function would be known to the XSUB as ST(0). The XSUB's incoming parameters and outgoing return values always begin at ST(0). For many simple cases the **xsubpp** compiler will generate the code necessary to handle the argument stack by embedding code fragments found in the typemaps. In more complex cases the programmer must supply the code.

68.1.5 The RETVAL Variable

The RETVAL variable is a special C variable that is declared automatically for you. The C type of RETVAL matches the return type of the C library function. The **xsubpp** compiler will declare this variable in each XSUB with non-`void` return type. By default the generated C function will use RETVAL to hold the return value of the C library function being called. In simple cases the value of RETVAL will be placed in ST(0) of the argument stack where it can be received by Perl as the return value of the XSUB.

If the XSUB has a return type of `void` then the compiler will not declare a RETVAL variable for that function. When using a PPCODE: section no manipulation of the RETVAL variable is required, the section may use direct stack manipulation to place output values on the stack.

If PPCODE: directive is not used, `void` return value should be used only for subroutines which do not return a value, *even if* CODE: directive is used which sets ST(0) explicitly.

Older versions of this document recommended to use `void` return value in such cases. It was discovered that this could lead to segfaults in cases when XSUB was *truly* `void`. This practice is now deprecated, and may be not supported at some future version. Use the return value `SV *` in such cases. (Currently `xsubpp` contains some heuristic code which tries to disambiguate between "truly-void" and "old-practice-declared-as-void" functions. Hence your code is at mercy of this heuristics unless you use `SV *` as return value.)

68.1.6 Returning SVs, AVs and HVs through RETVAL

When you're using RETVAL to return an `SV *`, there's some magic going on behind the scenes that should be mentioned. When you're manipulating the argument stack using the ST(x) macro, for example, you usually have to pay special attention to reference counts. (For more about reference counts, see *perlguts*.) To make your life easier, the typemap file automatically makes `RETVAL` mortal when you're returning an `SV *`. Thus, the following two XSUBs are more or less equivalent:

```
void
alpha()
    PPCODE:
        ST(0) = newSVpv("Hello World",0);
        sv_2mortal(ST(0));
        XSRETURN(1);

SV *
beta()
    CODE:
        RETVAL = newSVpv("Hello World",0);
    OUTPUT:
        RETVAL
```

This is quite useful as it usually improves readability. While this works fine for an `SV *`, it's unfortunately not as easy to have `AV *` or `HV *` as a return value. You *should* be able to write:

```
AV *
array()
    CODE:
        RETVAL = newAV();
        /* do something with RETVAL */
    OUTPUT:
        RETVAL
```

But due to an unfixable bug (fixing it would break lots of existing CPAN modules) in the typemap file, the reference count of the `AV *` is not properly decremented. Thus, the above XSUB would leak memory whenever it is being called. The same problem exists for `HV *`.

When you're returning an `AV *` or a `HV *`, you have make sure their reference count is decremented by making the AV or HV mortal:

```
AV *
array()
    CODE:
        RETVAL = newAV();
        sv_2mortal((SV*)RETVAL);
        /* do something with RETVAL */
    OUTPUT:
        RETVAL
```

And also remember that you don't have to do this for an `SV *`.

68.1.7 The MODULE Keyword

The MODULE keyword is used to start the XS code and to specify the package of the functions which are being defined. All text preceding the first MODULE keyword is considered C code and is passed through to the output with POD stripped, but otherwise untouched. Every XS module will have a bootstrap function which is used to hook the XSUBs into Perl. The package name of this bootstrap function will match the value of the last MODULE statement in the XS source files. The value of MODULE should always remain constant within the same XS file, though this is not required.

The following example will start the XS code and will place all functions in a package named RPC.

```
MODULE = RPC
```

68.1.8 The PACKAGE Keyword

When functions within an XS source file must be separated into packages the PACKAGE keyword should be used. This keyword is used with the MODULE keyword and must follow immediately after it when used.

```
MODULE = RPC  PACKAGE = RPC

[ XS code in package RPC ]

MODULE = RPC  PACKAGE = RPCB

[ XS code in package RPCB ]

MODULE = RPC  PACKAGE = RPC

[ XS code in package RPC ]
```

The same package name can be used more than once, allowing for non-contiguous code. This is useful if you have a stronger ordering principle than package names.

Although this keyword is optional and in some cases provides redundant information it should always be used. This keyword will ensure that the XSUBs appear in the desired package.

68.1.9 The PREFIX Keyword

The PREFIX keyword designates prefixes which should be removed from the Perl function names. If the C function is `rpcb_gettime()` and the PREFIX value is `rpcb_` then Perl will see this function as `gettime()`.

This keyword should follow the PACKAGE keyword when used. If PACKAGE is not used then PREFIX should follow the MODULE keyword.

```
MODULE = RPC  PREFIX = rpc_

MODULE = RPC  PACKAGE = RPCB  PREFIX = rpcb_
```

68.1.10 The OUTPUT: Keyword

The OUTPUT: keyword indicates that certain function parameters should be updated (new values made visible to Perl) when the XSUB terminates or that certain values should be returned to the calling Perl function. For simple functions which have no CODE: or PPCODE: section, such as the sin() function above, the RETVAL variable is automatically designated as an output value. For more complex functions the **xsubpp** compiler will need help to determine which variables are output variables.

This keyword will normally be used to complement the CODE: keyword. The RETVAL variable is not recognized as an output variable when the CODE: keyword is present. The OUTPUT: keyword is used in this situation to tell the compiler that RETVAL really is an output variable.

The OUTPUT: keyword can also be used to indicate that function parameters are output variables. This may be necessary when a parameter has been modified within the function and the programmer would like the update to be seen by Perl.

```
bool_t
rpcb_gettime(host,timep)
     char *host
     time_t &timep
   OUTPUT:
     timep
```

The OUTPUT: keyword will also allow an output parameter to be mapped to a matching piece of code rather than to a typemap.

```
     bool_t
     rpcb_gettime(host,timep)
          char *host
          time_t &timep
        OUTPUT:
          timep sv_setnv(ST(1), (double)timep);
```

xsubpp emits an automatic `SvSETMAGIC()` for all parameters in the OUTPUT section of the XSUB, except RETVAL. This is the usually desired behavior, as it takes care of properly invoking 'set' magic on output parameters (needed for hash or array element parameters that must be created if they didn't exist). If for some reason, this behavior is not desired, the OUTPUT section may contain a `SETMAGIC: DISABLE` line to disable it for the remainder of the parameters in the OUTPUT section. Likewise, `SETMAGIC: ENABLE` can be used to reenable it for the remainder of the OUTPUT section. See *perlguts* for more details about 'set' magic.

68.1.11 The NO_OUTPUT Keyword

The NO_OUTPUT can be placed as the first token of the XSUB. This keyword indicates that while the C subroutine we provide an interface to has a non-`void` return type, the return value of this C subroutine should not be returned from the generated Perl subroutine.

With this keyword present The RETVAL Variable is created, and in the generated call to the subroutine this variable is assigned to, but the value of this variable is not going to be used in the auto-generated code.

This keyword makes sense only if `RETVAL` is going to be accessed by the user-supplied code. It is especially useful to make a function interface more Perl-like, especially when the C return value is just an error condition indicator. For example,

```
  NO_OUTPUT int
  delete_file(char *name)
    POSTCALL:
      if (RETVAL != 0)
          croak("Error %d while deleting file '%s'", RETVAL, name);
```

Here the generated XS function returns nothing on success, and will die() with a meaningful error message on error.

68.1.12 The CODE: Keyword

This keyword is used in more complicated XSUBs which require special handling for the C function. The RETVAL variable is still declared, but it will not be returned unless it is specified in the OUTPUT: section.

The following XSUB is for a C function which requires special handling of its parameters. The Perl usage is given first.

```
     $status = rpcb_gettime( "localhost", $timep );
```

The XSUB follows.

```
     bool_t
     rpcb_gettime(host,timep)
          char *host
          time_t timep
        CODE:
               RETVAL = rpcb_gettime( host, &timep );
        OUTPUT:
          timep
          RETVAL
```

68.1.13 The INIT: Keyword

The INIT: keyword allows initialization to be inserted into the XSUB before the compiler generates the call to the C function. Unlike the CODE: keyword above, this keyword does not affect the way the compiler handles RETVAL.

```
    bool_t
    rpcb_gettime(host,timep)
          char *host
          time_t &timep
        INIT:
          printf("# Host is %s\n", host );
        OUTPUT:
          timep
```

Another use for the INIT: section is to check for preconditions before making a call to the C function:

```
    long long
    lldiv(a,b)
        long long a
        long long b
      INIT:
        if (a == 0 && b == 0)
            XSRETURN_UNDEF;
        if (b == 0)
            croak("lldiv: cannot divide by 0");
```

68.1.14 The NO_INIT Keyword

The NO_INIT keyword is used to indicate that a function parameter is being used only as an output value. The **xsubpp** compiler will normally generate code to read the values of all function parameters from the argument stack and assign them to C variables upon entry to the function. NO_INIT will tell the compiler that some parameters will be used for output rather than for input and that they will be handled before the function terminates.

The following example shows a variation of the rpcb_gettime() function. This function uses the timep variable only as an output variable and does not care about its initial contents.

```
     bool_t
     rpcb_gettime(host,timep)
          char *host
          time_t &timep = NO_INIT
        OUTPUT:
          timep
```

68.1.15 Initializing Function Parameters

C function parameters are normally initialized with their values from the argument stack (which in turn contains the parameters that were passed to the XSUB from Perl). The typemaps contain the code segments which are used to translate the Perl values to the C parameters. The programmer, however, is allowed to override the typemaps and supply alternate (or additional) initialization code. Initialization code starts with the first =, ; or + on a line in the INPUT: section. The only exception happens if this ; terminates the line, then this ; is quietly ignored.

The following code demonstrates how to supply initialization code for function parameters. The initialization code is eval'd within double quotes by the compiler before it is added to the output so anything which should be interpreted literally [mainly $, @, or \\] must be protected with backslashes. The variables $var, $arg, and $type can be used as in typemaps.

```
bool_t
rpcb_gettime(host,timep)
     char *host = (char *)SvPV($arg,PL_na);
     time_t &timep = 0;
   OUTPUT:
     timep
```

This should not be used to supply default values for parameters. One would normally use this when a function parameter must be processed by another library function before it can be used. Default parameters are covered in the next section.

If the initialization begins with =, then it is output in the declaration for the input variable, replacing the initialization supplied by the typemap. If the initialization begins with `;` or +, then it is performed after all of the input variables have been declared. In the `;` case the initialization normally supplied by the typemap is not performed. For the + case, the declaration for the variable will include the initialization from the typemap. A global variable, `%v`, is available for the truly rare case where information from one initialization is needed in another initialization.

Here's a truly obscure example:

```
bool_t
rpcb_gettime(host,timep)
     time_t &timep ; /* \$v{timep}=@{[$v{timep}=$arg]} */
     char *host + SvOK($v{timep}) ? SvPV($arg,PL_na) : NULL;
   OUTPUT:
     timep
```

The construct `\$v{timep}=@{[$v{timep}=$arg]}` used in the above example has a two-fold purpose: first, when this line is processed by **xsubpp**, the Perl snippet `$v{timep}=$arg` is evaluated. Second, the text of the evaluated snippet is output into the generated C file (inside a C comment)! During the processing of `char *host` line, $arg will evaluate to `ST(0)`, and `$v{timep}` will evaluate to `ST(1)`.

68.1.16 Default Parameter Values

Default values for XSUB arguments can be specified by placing an assignment statement in the parameter list. The default value may be a number, a string or the special string `NO_INIT`. Defaults should always be used on the right-most parameters only.

To allow the XSUB for rpcb_gettime() to have a default host value the parameters to the XSUB could be rearranged. The XSUB will then call the real rpcb_gettime() function with the parameters in the correct order. This XSUB can be called from Perl with either of the following statements:

```
$status = rpcb_gettime( $timep, $host );

$status = rpcb_gettime( $timep );
```

The XSUB will look like the code which follows. A CODE: block is used to call the real rpcb_gettime() function with the parameters in the correct order for that function.

```
bool_t
rpcb_gettime(timep,host="localhost")
     char *host
     time_t timep = NO_INIT
   CODE:
          RETVAL = rpcb_gettime( host, &timep );
   OUTPUT:
     timep
     RETVAL
```

68.1.17 The PREINIT: Keyword

The PREINIT: keyword allows extra variables to be declared immediately before or after the declarations of the parameters from the INPUT: section are emitted.

If a variable is declared inside a CODE: section it will follow any typemap code that is emitted for the input parameters. This may result in the declaration ending up after C code, which is C syntax error. Similar errors may happen with an explicit `;`-type or `+`-type initialization of parameters is used (see §68.1.15). Declaring these variables in an INIT: section will not help.

In such cases, to force an additional variable to be declared together with declarations of other variables, place the declaration into a PREINIT: section. The PREINIT: keyword may be used one or more times within an XSUB.

The following examples are equivalent, but if the code is using complex typemaps then the first example is safer.

```
bool_t
rpcb_gettime(timep)
     time_t timep = NO_INIT
   PREINIT:
     char *host = "localhost";
   CODE:
     RETVAL = rpcb_gettime( host, &timep );
   OUTPUT:
     timep
     RETVAL
```

For this particular case an INIT: keyword would generate the same C code as the PREINIT: keyword. Another correct, but error-prone example:

```
bool_t
rpcb_gettime(timep)
     time_t timep = NO_INIT
   CODE:
     char *host = "localhost";
     RETVAL = rpcb_gettime( host, &timep );
   OUTPUT:
     timep
     RETVAL
```

Another way to declare `host` is to use a C block in the CODE: section:

```
bool_t
rpcb_gettime(timep)
     time_t timep = NO_INIT
   CODE:
     {
       char *host = "localhost";
       RETVAL = rpcb_gettime( host, &timep );
     }
   OUTPUT:
     timep
     RETVAL
```

The ability to put additional declarations before the typemap entries are processed is very handy in the cases when typemap conversions manipulate some global state:

```
MyObject
mutate(o)
    PREINIT:
        MyState st = global_state;
    INPUT:
        MyObject o;
    CLEANUP:
        reset_to(global_state, st);
```

Here we suppose that conversion to `MyObject` in the INPUT: section and from MyObject when processing RETVAL will modify a global variable `global_state`. After these conversions are performed, we restore the old value of `global_state` (to avoid memory leaks, for example).

There is another way to trade clarity for compactness: INPUT sections allow declaration of C variables which do not appear in the parameter list of a subroutine. Thus the above code for mutate() can be rewritten as

```
MyObject
mutate(o)
      MyState st = global_state;
      MyObject o;
    CLEANUP:
      reset_to(global_state, st);
```

and the code for rpcb_gettime() can be rewritten as

```
 bool_t
 rpcb_gettime(timep)
      time_t timep = NO_INIT
      char *host = "localhost";
    C_ARGS:
      host, &timep
    OUTPUT:
      timep
      RETVAL
```

68.1.18 The SCOPE: Keyword

The SCOPE: keyword allows scoping to be enabled for a particular XSUB. If enabled, the XSUB will invoke ENTER and LEAVE automatically.

To support potentially complex type mappings, if a typemap entry used by an XSUB contains a comment like `/*scope*/` then scoping will be automatically enabled for that XSUB.

To enable scoping:

```
SCOPE: ENABLE
```

To disable scoping:

```
SCOPE: DISABLE
```

68.1.19 The INPUT: Keyword

The XSUB's parameters are usually evaluated immediately after entering the XSUB. The INPUT: keyword can be used to force those parameters to be evaluated a little later. The INPUT: keyword can be used multiple times within an XSUB and can be used to list one or more input variables. This keyword is used with the PREINIT: keyword.

The following example shows how the input parameter `timep` can be evaluated late, after a PREINIT.

```
bool_t
rpcb_gettime(host,timep)
      char *host
    PREINIT:
      time_t tt;
    INPUT:
      time_t timep
    CODE:
           RETVAL = rpcb_gettime( host, &tt );
           timep = tt;
    OUTPUT:
      timep
      RETVAL
```

The next example shows each input parameter evaluated late.

```
bool_t
rpcb_gettime(host,timep)
    PREINIT:
      time_t tt;
    INPUT:
      char *host
    PREINIT:
      char *h;
    INPUT:
      time_t timep
    CODE:
           h = host;
           RETVAL = rpcb_gettime( h, &tt );
           timep = tt;
    OUTPUT:
      timep
      RETVAL
```

Since INPUT sections allow declaration of C variables which do not appear in the parameter list of a subroutine, this may be shortened to:

```
bool_t
rpcb_gettime(host,timep)
      time_t tt;
      char *host;
      char *h = host;
      time_t timep;
    CODE:
      RETVAL = rpcb_gettime( h, &tt );
      timep = tt;
    OUTPUT:
      timep
      RETVAL
```

(We used our knowledge that input conversion for `char *` is a "simple" one, thus `host` is initialized on the declaration line, and our assignment `h = host` is not performed too early. Otherwise one would need to have the assignment `h = host` in a CODE: or INIT: section.)

68.1.20 The IN/OUTLIST/IN_OUTLIST/OUT/IN_OUT Keywords

In the list of parameters for an XSUB, one can precede parameter names by the `IN`/`OUTLIST`/`IN_OUTLIST`/`OUT`/`IN_OUT` keywords. `IN` keyword is the default, the other keywords indicate how the Perl interface should differ from the C interface.

Parameters preceded by `OUTLIST`/`IN_OUTLIST`/`OUT`/`IN_OUT` keywords are considered to be used by the C subroutine *via pointers*. `OUTLIST`/`OUT` keywords indicate that the C subroutine does not inspect the memory pointed by this parameter, but will write through this pointer to provide additional return values.

Parameters preceded by `OUTLIST` keyword do not appear in the usage signature of the generated Perl function.

Parameters preceded by `IN_OUTLIST`/`IN_OUT`/`OUT` *do* appear as parameters to the Perl function. With the exception of `OUT`-parameters, these parameters are converted to the corresponding C type, then pointers to these data are given as arguments to the C function. It is expected that the C function will write through these pointers.

The return list of the generated Perl function consists of the C return value from the function (unless the XSUB is of `void` return type or `The NO_OUTPUT Keyword` was used) followed by all the `OUTLIST` and `IN_OUTLIST` parameters (in the order of appearance). On the return from the XSUB the `IN_OUT`/`OUT` Perl parameter will be modified to have the values written by the C function.

For example, an XSUB

```
void
day_month(OUTLIST day, IN unix_time, OUTLIST month)
  int day
  int unix_time
  int month
```

should be used from Perl as

```
my ($day, $month) = day_month(time);
```

The C signature of the corresponding function should be

```
void day_month(int *day, int unix_time, int *month);
```

The `IN`/`OUTLIST`/`IN_OUTLIST`/`IN_OUT`/`OUT` keywords can be mixed with ANSI-style declarations, as in

```
void
day_month(OUTLIST int day, int unix_time, OUTLIST int month)
```

(here the optional `IN` keyword is omitted).

The `IN_OUT` parameters are identical with parameters introduced with The & Unary Operator and put into the `OUTPUT:` section (see The OUTPUT: Keyword). The `IN_OUTLIST` parameters are very similar, the only difference being that the value C function writes through the pointer would not modify the Perl parameter, but is put in the output list.

The `OUTLIST`/`OUT` parameter differ from `IN_OUTLIST`/`IN_OUT` parameters only by the initial value of the Perl parameter not being read (and not being given to the C function - which gets some garbage instead). For example, the same C function as above can be interfaced with as

```
void day_month(OUT int day, int unix_time, OUT int month);
```

or

```
void
day_month(day, unix_time, month)
    int &day = NO_INIT
    int  unix_time
    int &month = NO_INIT
  OUTPUT:
    day
    month
```

However, the generated Perl function is called in very C-ish style:

```
my ($day, $month);
day_month($day, time, $month);
```

68.1.21 The `length(NAME)` Keyword

If one of the input arguments to the C function is the length of a string argument `NAME`, one can substitute the name of the length-argument by `length(NAME)` in the XSUB declaration. This argument must be omited when the generated Perl function is called. E.g.,

```
void
dump_chars(char *s, short l)
{
  short n = 0;
  while (n < l) {
      printf("s[%d] = \"\\%#03o\"\n", n, (int)s[n]);
      n++;
  }
}

MODULE = x            PACKAGE = x

void dump_chars(char *s, short length(s))
```

should be called as `dump_chars($string)`.

This directive is supported with ANSI-type function declarations only.

68.1.22 Variable-length Parameter Lists

XSUBs can have variable-length parameter lists by specifying an ellipsis `(...)` in the parameter list. This use of the ellipsis is similar to that found in ANSI C. The programmer is able to determine the number of arguments passed to the XSUB by examining the `items` variable which the **xsubpp** compiler supplies for all XSUBs. By using this mechanism one can create an XSUB which accepts a list of parameters of unknown length.

The *host* parameter for the rpcb_gettime() XSUB can be optional so the ellipsis can be used to indicate that the XSUB will take a variable number of parameters. Perl should be able to call this XSUB with either of the following statements.

```
   $status = rpcb_gettime( $timep, $host );

   $status = rpcb_gettime( $timep );
```

The XS code, with ellipsis, follows.

```
bool_t
rpcb_gettime(timep, ...)
     time_t timep = NO_INIT
   PREINIT:
     char *host = "localhost";
     STRLEN n_a;
   CODE:
     if( items > 1 )
          host = (char *)SvPV(ST(1), n_a);
     RETVAL = rpcb_gettime( host, &timep );
   OUTPUT:
     timep
     RETVAL
```

68.1.23 The C_ARGS: Keyword

The C_ARGS: keyword allows creating of XSUBS which have different calling sequence from Perl than from C, without a need to write CODE: or PPCODE: section. The contents of the C_ARGS: paragraph is put as the argument to the called C function without any change.

For example, suppose that a C function is declared as

```
symbolic nth_derivative(int n, symbolic function, int flags);
```

and that the default flags are kept in a global C variable `default_flags`. Suppose that you want to create an interface which is called as

```
$second_deriv = $function->nth_derivative(2);
```

To do this, declare the XSUB as

```
symbolic
nth_derivative(function, n)
    symbolic        function
    int             n
  C_ARGS:
    n, function, default_flags
```

68.1.24 The PPCODE: Keyword

The PPCODE: keyword is an alternate form of the CODE: keyword and is used to tell the **xsubpp** compiler that the programmer is supplying the code to control the argument stack for the XSUBs return values. Occasionally one will want an XSUB to return a list of values rather than a single value. In these cases one must use PPCODE: and then explicitly push the list of values on the stack. The PPCODE: and CODE: keywords should not be used together within the same XSUB.

The actual difference between PPCODE: and CODE: sections is in the initialization of `SP` macro (which stands for the *current* Perl stack pointer), and in the handling of data on the stack when returning from an XSUB. In CODE: sections SP preserves the value which was on entry to the XSUB: SP is on the function pointer (which follows the last parameter). In PPCODE: sections SP is moved backward to the beginning of the parameter list, which allows `PUSH*()` macros to place output values in the place Perl expects them to be when the XSUB returns back to Perl.

The generated trailer for a CODE: section ensures that the number of return values Perl will see is either 0 or 1 (depending on the `void`ness of the return value of the C function, and heuristics mentioned in §68.1.5). The trailer generated for a PPCODE: section is based on the number of return values and on the number of times `SP` was updated by `[X]PUSH*()` macros.

Note that macros `ST(i)`, `XST_m*()` and `XSRETURN*()` work equally well in CODE: sections and PPCODE: sections.

The following XSUB will call the C rpcb_gettime() function and will return its two output values, timep and status, to Perl as a single list.

```
void
rpcb_gettime(host)
     char *host
   PREINIT:
     time_t  timep;
     bool_t  status;
   PPCODE:
     status = rpcb_gettime( host, &timep );
     EXTEND(SP, 2);
     PUSHs(sv_2mortal(newSViv(status)));
     PUSHs(sv_2mortal(newSViv(timep)));
```

Notice that the programmer must supply the C code necessary to have the real rpcb_gettime() function called and to have the return values properly placed on the argument stack.

The `void` return type for this function tells the **xsubpp** compiler that the RETVAL variable is not needed or used and that it should not be created. In most scenarios the void return type should be used with the PPCODE: directive.

The EXTEND() macro is used to make room on the argument stack for 2 return values. The PPCODE: directive causes the **xsubpp** compiler to create a stack pointer available as `SP`, and it is this pointer which is being used in the EXTEND() macro. The values are then pushed onto the stack with the PUSHs() macro.

Now the rpcb_gettime() function can be used from Perl with the following statement.

```
($status, $timep) = rpcb_gettime("localhost");
```

When handling output parameters with a PPCODE section, be sure to handle 'set' magic properly. See *perlguts* for details about 'set' magic.

68.1.25 Returning Undef And Empty Lists

Occasionally the programmer will want to return simply `undef` or an empty list if a function fails rather than a separate status value. The rpcb_gettime() function offers just this situation. If the function succeeds we would like to have it return the time and if it fails we would like to have undef returned. In the following Perl code the value of $timep will either be undef or it will be a valid time.

```
$timep = rpcb_gettime( "localhost" );
```

The following XSUB uses the `SV *` return type as a mnemonic only, and uses a CODE: block to indicate to the compiler that the programmer has supplied all the necessary code. The sv_newmortal() call will initialize the return value to undef, making that the default return value.

```
SV *
rpcb_gettime(host)
     char *  host
   PREINIT:
     time_t  timep;
     bool_t x;
   CODE:
     ST(0) = sv_newmortal();
     if( rpcb_gettime( host, &timep ) )
          sv_setnv( ST(0), (double)timep);
```

The next example demonstrates how one would place an explicit undef in the return value, should the need arise.

```
SV *
rpcb_gettime(host)
     char *  host
   PREINIT:
     time_t  timep;
     bool_t x;
   CODE:
     ST(0) = sv_newmortal();
     if( rpcb_gettime( host, &timep ) ){
          sv_setnv( ST(0), (double)timep);
     }
     else{
          ST(0) = &PL_sv_undef;
     }
```

To return an empty list one must use a PPCODE: block and then not push return values on the stack.

```
void
rpcb_gettime(host)
     char *host
   PREINIT:
     time_t  timep;
   PPCODE:
     if( rpcb_gettime( host, &timep ) )
          PUSHs(sv_2mortal(newSViv(timep)));
     else{
         /* Nothing pushed on stack, so an empty
          * list is implicitly returned. */
     }
```

Some people may be inclined to include an explicit `return` in the above XSUB, rather than letting control fall through to the end. In those situations `XSRETURN_EMPTY` should be used, instead. This will ensure that the XSUB stack is properly adjusted. Consult *perlapi* for other `XSRETURN` macros.

Since `XSRETURN_*` macros can be used with CODE blocks as well, one can rewrite this example as:

```
int
rpcb_gettime(host)
     char *host
   PREINIT:
     time_t  timep;
   CODE:
     RETVAL = rpcb_gettime( host, &timep );
     if (RETVAL == 0)
           XSRETURN_UNDEF;
   OUTPUT:
     RETVAL
```

In fact, one can put this check into a POSTCALL: section as well. Together with PREINIT: simplifications, this leads to:

```
int
rpcb_gettime(host)
     char *host
     time_t  timep;
   POSTCALL:
     if (RETVAL == 0)
           XSRETURN_UNDEF;
```

68.1.26 The REQUIRE: Keyword

The REQUIRE: keyword is used to indicate the minimum version of the **xsubpp** compiler needed to compile the XS module. An XS module which contains the following statement will compile with only **xsubpp** version 1.922 or greater:

```
        REQUIRE: 1.922
```

68.1.27 The CLEANUP: Keyword

This keyword can be used when an XSUB requires special cleanup procedures before it terminates. When the CLEANUP: keyword is used it must follow any CODE:, PPCODE:, or OUTPUT: blocks which are present in the XSUB. The code specified for the cleanup block will be added as the last statements in the XSUB.

68.1.28 The POSTCALL: Keyword

This keyword can be used when an XSUB requires special procedures executed after the C subroutine call is performed. When the POSTCALL: keyword is used it must precede OUTPUT: and CLEANUP: blocks which are present in the XSUB.

See examples in §68.1.11 and §68.1.25.

The POSTCALL: block does not make a lot of sense when the C subroutine call is supplied by user by providing either CODE: or PPCODE: section.

68.1.29 The BOOT: Keyword

The BOOT: keyword is used to add code to the extension's bootstrap function. The bootstrap function is generated by the **xsubpp** compiler and normally holds the statements necessary to register any XSUBs with Perl. With the BOOT: keyword the programmer can tell the compiler to add extra statements to the bootstrap function.

This keyword may be used any time after the first MODULE keyword and should appear on a line by itself. The first blank line after the keyword will terminate the code block.

```
     BOOT:
     # The following message will be printed when the
     # bootstrap function executes.
     printf("Hello from the bootstrap!\n");
```

68.1.30 The VERSIONCHECK: Keyword

The VERSIONCHECK: keyword corresponds to **xsubpp**'s `-versioncheck` and `-noversioncheck` options. This keyword overrides the command line options. Version checking is enabled by default. When version checking is enabled the XS module will attempt to verify that its version matches the version of the PM module.

To enable version checking:

```
    VERSIONCHECK: ENABLE
```

To disable version checking:

```
    VERSIONCHECK: DISABLE
```

68.1.31 The PROTOTYPES: Keyword

The PROTOTYPES: keyword corresponds to **xsubpp**'s `-prototypes` and `-noprototypes` options. This keyword overrides the command line options. Prototypes are enabled by default. When prototypes are enabled XSUBs will be given Perl prototypes. This keyword may be used multiple times in an XS module to enable and disable prototypes for different parts of the module.

To enable prototypes:

```
PROTOTYPES: ENABLE
```

To disable prototypes:

```
PROTOTYPES: DISABLE
```

68.1.32 The PROTOTYPE: Keyword

This keyword is similar to the PROTOTYPES: keyword above but can be used to force **xsubpp** to use a specific prototype for the XSUB. This keyword overrides all other prototype options and keywords but affects only the current XSUB. Consult Prototypes in *perlsub* for information about Perl prototypes.

```
bool_t
rpcb_gettime(timep, ...)
      time_t timep = NO_INIT
    PROTOTYPE: $;$
    PREINIT:
      char *host = "localhost";
      STRLEN n_a;
    CODE:
              if( items > 1 )
                   host = (char *)SvPV(ST(1), n_a);
              RETVAL = rpcb_gettime( host, &timep );
    OUTPUT:
      timep
      RETVAL
```

If the prototypes are enabled, you can disable it locally for a given XSUB as in the following example:

```
void
rpcb_gettime_noproto()
    PROTOTYPE: DISABLE
...
```

68.1.33 The ALIAS: Keyword

The ALIAS: keyword allows an XSUB to have two or more unique Perl names and to know which of those names was used when it was invoked. The Perl names may be fully-qualified with package names. Each alias is given an index. The compiler will setup a variable called `ix` which contain the index of the alias which was used. When the XSUB is called with its declared name `ix` will be 0.

The following example will create aliases `FOO::gettime()` and `BAR::getit()` for this function.

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t &timep
```

```
        ALIAS:
            FOO::gettime = 1
            BAR::getit = 2
        INIT:
          printf("# ix = %d\n", ix );
        OUTPUT:
          timep
```

68.1.34 The OVERLOAD: Keyword

Instead of writing an overloaded interface using pure Perl, you can also use the OVERLOAD keyword to define additional Perl names for your functions (like the ALIAS: keyword above). However, the overloaded functions must be defined with three parameters (except for the nomethod() function which needs four parameters). If any function has the OVERLOAD: keyword, several additional lines will be defined in the c file generated by xsubpp in order to register with the overload magic.

Since blessed objects are actually stored as RV's, it is useful to use the typemap features to preprocess parameters and extract the actual SV stored within the blessed RV. See the sample for T_PTROBJ_SPECIAL below.

To use the OVERLOAD: keyword, create an XS function which takes three input parameters (or use the c style '...' definition) like this:

```
    SV *
    cmp (lobj, robj, swap)
    My_Module_obj    lobj
    My_Module_obj    robj
    IV               swap
    OVERLOAD: cmp <=>
    { /* function defined here */}
```

In this case, the function will overload both of the three way comparison operators. For all overload operations using non-alpha characters, you must type the parameter without quoting, seperating multiple overloads with whitespace. Note that "" (the stringify overload) should be entered as \"\" (i.e. escaped).

68.1.35 The FALLBACK: Keyword

In addition to the OVERLOAD keyword, if you need to control how Perl autogenerates missing overloaded operators, you can set the FALLBACK keyword in the module header section, like this:

```
    MODULE = RPC  PACKAGE = RPC

    FALLBACK: TRUE
    ...
```

where FALLBACK can take any of the three values TRUE, FALSE, or UNDEF. If you do not set any FALLBACK value when using OVERLOAD, it defaults to UNDEF. FALLBACK is not used except when one or more functions using OVERLOAD have been defined. Please see Fallback in *overload* for more details.

68.1.36 The INTERFACE: Keyword

This keyword declares the current XSUB as a keeper of the given calling signature. If some text follows this keyword, it is considered as a list of functions which have this signature, and should be attached to the current XSUB.

For example, if you have 4 C functions multiply(), divide(), add(), subtract() all having the signature:

```
    symbolic f(symbolic, symbolic);
```

you can make them all to use the same XSUB using this:

```
    symbolic
    interface_s_ss(arg1, arg2)
        symbolic        arg1
        symbolic        arg2
    INTERFACE:
        multiply divide
        add subtract
```

(This is the complete XSUB code for 4 Perl functions!) Four generated Perl function share names with corresponding C functions.

The advantage of this approach comparing to ALIAS: keyword is that there is no need to code a switch statement, each Perl function (which shares the same XSUB) knows which C function it should call. Additionally, one can attach an extra function remainder() at runtime by using

```
    CV *mycv = newXSproto("Symbolic::remainder",
                          XS_Symbolic_interface_s_ss, __FILE__, "$$");
    XSINTERFACE_FUNC_SET(mycv, remainder);
```

say, from another XSUB. (This example supposes that there was no INTERFACE_MACRO: section, otherwise one needs to use something else instead of `XSINTERFACE_FUNC_SET`, see the next section.)

68.1.37 The INTERFACE_MACRO: Keyword

This keyword allows one to define an INTERFACE using a different way to extract a function pointer from an XSUB. The text which follows this keyword should give the name of macros which would extract/set a function pointer. The extractor macro is given return type, `CV*`, and `XSANY.any_dptr` for this `CV*`. The setter macro is given cv, and the function pointer.

The default value is `XSINTERFACE_FUNC` and `XSINTERFACE_FUNC_SET`. An INTERFACE keyword with an empty list of functions can be omitted if INTERFACE_MACRO keyword is used.

Suppose that in the previous example functions pointers for multiply(), divide(), add(), subtract() are kept in a global C array `fp[]` with offsets being `multiply_off`, `divide_off`, `add_off`, `subtract_off`. Then one can use

```
    #define XSINTERFACE_FUNC_BYOFFSET(ret,cv,f) \
        ((XSINTERFACE_CVT(ret,))fp[CvXSUBANY(cv).any_i32])
    #define XSINTERFACE_FUNC_BYOFFSET_set(cv,f) \
        CvXSUBANY(cv).any_i32 = CAT2( f, _off )
```

in C section,

```
    symbolic
    interface_s_ss(arg1, arg2)
        symbolic        arg1
        symbolic        arg2
      INTERFACE_MACRO:
        XSINTERFACE_FUNC_BYOFFSET
        XSINTERFACE_FUNC_BYOFFSET_set
      INTERFACE:
        multiply divide
        add subtract
```

in XSUB section.

68.1.38 The INCLUDE: Keyword

This keyword can be used to pull other files into the XS module. The other files may have XS code. INCLUDE: can also be used to run a command to generate the XS code to be pulled into the module.
The file *Rpcb1.xsh* contains our `rpcb_gettime()` function:

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t &timep
    OUTPUT:
      timep
```

The XS module can use INCLUDE: to pull that file into it.

```
INCLUDE: Rpcb1.xsh
```

If the parameters to the INCLUDE: keyword are followed by a pipe (|) then the compiler will interpret the parameters as a command.

```
INCLUDE: cat Rpcb1.xsh |
```

68.1.39 The CASE: Keyword

The CASE: keyword allows an XSUB to have multiple distinct parts with each part acting as a virtual XSUB. CASE: is greedy and if it is used then all other XS keywords must be contained within a CASE:. This means nothing may precede the first CASE: in the XSUB and anything following the last CASE: is included in that case.
A CASE: might switch via a parameter of the XSUB, via the `ix` ALIAS: variable (see §68.1.33), or maybe via the `items` variable (see §68.1.22). The last CASE: becomes the **default** case if it is not associated with a conditional. The following example shows CASE switched via `ix` with a function `rpcb_gettime()` having an alias `x_gettime()`. When the function is called as `rpcb_gettime()` its parameters are the usual `(char *host, time_t *timep)`, but when the function is called as `x_gettime()` its parameters are reversed, `(time_t *timep, char *host)`.

```
long
rpcb_gettime(a,b)
  CASE: ix == 1
    ALIAS:
      x_gettime = 1
    INPUT:
      # 'a' is timep, 'b' is host
      char *b
      time_t a = NO_INIT
    CODE:
           RETVAL = rpcb_gettime( b, &a );
    OUTPUT:
      a
      RETVAL
  CASE:
      # 'a' is host, 'b' is timep
      char *a
      time_t &b = NO_INIT
    OUTPUT:
      b
      RETVAL
```

That function can be called with either of the following statements. Note the different argument lists.

```
$status = rpcb_gettime( $host, $timep );

$status = x_gettime( $timep, $host );
```

68.1.40 The & Unary Operator

The & unary operator in the INPUT: section is used to tell **xsubpp** that it should convert a Perl value to/from C using the C type to the left of &, but provide a pointer to this value when the C function is called.

This is useful to avoid a CODE: block for a C function which takes a parameter by reference. Typically, the parameter should be not a pointer type (an `int` or `long` but not an `int*` or `long*`).

The following XSUB will generate incorrect C code. The **xsubpp** compiler will turn this into code which calls `rpcb_gettime()` with parameters `(char *host, time_t timep)`, but the real `rpcb_gettime()` wants the `timep` parameter to be of type `time_t*` rather than `time_t`.

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t timep
    OUTPUT:
      timep
```

That problem is corrected by using the & operator. The **xsubpp** compiler will now turn this into code which calls `rpcb_gettime()` correctly with parameters `(char *host, time_t *timep)`. It does this by carrying the & through, so the function call looks like `rpcb_gettime(host, &timep)`.

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t &timep
    OUTPUT:
      timep
```

68.1.41 Inserting POD, Comments and C Preprocessor Directives

C preprocessor directives are allowed within BOOT:, PREINIT: INIT:, CODE:, PPCODE:, POSTCALL:, and CLEANUP: blocks, as well as outside the functions. Comments are allowed anywhere after the MODULE keyword. The compiler will pass the preprocessor directives through untouched and will remove the commented lines. POD documentation is allowed at any point, both in the C and XS language sections. POD must be terminated with a `=cut` command; `xsubpp` will exit with an error if it does not. It is very unlikely that human generated C code will be mistaken for POD, as most indenting styles result in whitespace in front of any line starting with =. Machine generated XS files may fall into this trap unless care is taken to ensure that a space breaks the sequence "\n=".

Comments can be added to XSUBs by placing a # as the first non-whitespace of a line. Care should be taken to avoid making the comment look like a C preprocessor directive, lest it be interpreted as such. The simplest way to prevent this is to put whitespace in front of the #.

If you use preprocessor directives to choose one of two versions of a function, use

```
#if ... version1
#else /* ... version2  */
#endif
```

and not

```
#if ... version1
#endif
#if ... version2
#endif
```

because otherwise **xsubpp** will believe that you made a duplicate definition of the function. Also, put a blank line before the #else/#endif so it will not be seen as part of the function body.

68.1.42 Using XS With C++

If an XSUB name contains `::`, it is considered to be a C++ method. The generated Perl function will assume that its first argument is an object pointer. The object pointer will be stored in a variable called THIS. The object should have been created by C++ with the new() function and should be blessed by Perl with the sv_setref_pv() macro. The blessing of the object by Perl can be handled by a typemap. An example typemap is shown at the end of this section.

If the return type of the XSUB includes `static`, the method is considered to be a static method. It will call the C++ function using the class::method() syntax. If the method is not static the function will be called using the THIS->method() syntax.

The next examples will use the following C++ class.

```
class color {
     public:
     color();
     ~color();
     int blue();
     void set_blue( int );

     private:
     int c_blue;
};
```

The XSUBs for the blue() and set_blue() methods are defined with the class name but the parameter for the object (THIS, or "self") is implicit and is not listed.

```
int
color::blue()

void
color::set_blue( val )
     int val
```

Both Perl functions will expect an object as the first parameter. In the generated C++ code the object is called `THIS`, and the method call will be performed on this object. So in the C++ code the blue() and set_blue() methods will be called as this:

```
RETVAL = THIS->blue();

THIS->set_blue( val );
```

You could also write a single get/set method using an optional argument:

```
int
color::blue( val = NO_INIT )
    int val
    PROTOTYPE $;$
    CODE:
        if (items > 1)
            THIS->set_blue( val );
        RETVAL = THIS->blue();
    OUTPUT:
        RETVAL
```

If the function's name is **DESTROY** then the C++ `delete` function will be called and `THIS` will be given as its parameter. The generated C++ code for

```
void
color::DESTROY()
```

will look like this:

```
color *THIS = ...; // Initialized as in typemap

delete THIS;
```

If the function's name is **new** then the C++ `new` function will be called to create a dynamic C++ object. The XSUB will expect the class name, which will be kept in a variable called `CLASS`, to be given as the first argument.

```
color *
color::new()
```

The generated C++ code will call `new`.

```
RETVAL = new color();
```

The following is an example of a typemap that could be used for this C++ example.

```
TYPEMAP
color *             O_OBJECT

OUTPUT
# The Perl object is blessed into 'CLASS', which should be a
# char* having the name of the package for the blessing.
O_OBJECT
    sv_setref_pv( $arg, CLASS, (void*)$var );

INPUT
O_OBJECT
    if( sv_isobject($arg) && (SvTYPE(SvRV($arg)) == SVt_PVMG) )
            $var = ($type)SvIV((SV*)SvRV( $arg ));
    else{
            warn( \"${Package}::$func_name() -- $var is not a blessed SV reference\" );
            XSRETURN_UNDEF;
    }
```

68.1.43 Interface Strategy

When designing an interface between Perl and a C library a straight translation from C to XS (such as created by `h2xs -x`) is often sufficient. However, sometimes the interface will look very C-like and occasionally nonintuitive, especially when the C function modifies one of its parameters, or returns failure inband (as in "negative return values mean failure"). In cases where the programmer wishes to create a more Perl-like interface the following strategy may help to identify the more critical parts of the interface.

Identify the C functions with input/output or output parameters. The XSUBs for these functions may be able to return lists to Perl.

Identify the C functions which use some inband info as an indication of failure. They may be candidates to return undef or an empty list in case of failure. If the failure may be detected without a call to the C function, you may want to use an INIT: section to report the failure. For failures detectable after the C function returns one may want to use a POSTCALL: section to process the failure. In more complicated cases use CODE: or PPCODE: sections.

If many functions use the same failure indication based on the return value, you may want to create a special typedef to handle this situation. Put

```
  typedef int negative_is_failure;
```

near the beginning of XS file, and create an OUTPUT typemap entry for `negative_is_failure` which converts negative values to `undef`, or maybe croak()s. After this the return value of type `negative_is_failure` will create more Perl-like interface.

Identify which values are used by only the C and XSUB functions themselves, say, when a parameter to a function should be a contents of a global variable. If Perl does not need to access the contents of the value then it may not be necessary to provide a translation for that value from C to Perl.

Identify the pointers in the C function parameter lists and return values. Some pointers may be used to implement input/output or output parameters, they can be handled in XS with the `&` unary operator, and, possibly, using the NO_INIT keyword. Some others will require handling of types like `int *`, and one needs to decide what a useful Perl translation will do in such a case. When the semantic is clear, it is advisable to put the translation into a typemap file.

Identify the structures used by the C functions. In many cases it may be helpful to use the T_PTROBJ typemap for these structures so they can be manipulated by Perl as blessed objects. (This is handled automatically by `h2xs -x`.)

If the same C type is used in several different contexts which require different translations, `typedef` several new types mapped to this C type, and create separate *typemap* entries for these new types. Use these types in declarations of return type and parameters to XSUBs.

68.1.44 Perl Objects And C Structures

When dealing with C structures one should select either **T_PTROBJ** or **T_PTRREF** for the XS type. Both types are designed to handle pointers to complex objects. The T_PTRREF type will allow the Perl object to be unblessed while the T_PTROBJ type requires that the object be blessed. By using T_PTROBJ one can achieve a form of type-checking because the XSUB will attempt to verify that the Perl object is of the expected type.

The following XS code shows the getnetconfigent() function which is used with ONC+ TIRPC. The getnetconfigent() function will return a pointer to a C structure and has the C prototype shown below. The example will demonstrate how the C pointer will become a Perl reference. Perl will consider this reference to be a pointer to a blessed object and will attempt to call a destructor for the object. A destructor will be provided in the XS source to free the memory used by getnetconfigent(). Destructors in XS can be created by specifying an XSUB function whose name ends with the word **DESTROY**. XS destructors can be used to free memory which may have been malloc'd by another XSUB.

```
     struct netconfig *getnetconfigent(const char *netid);
```

A `typedef` will be created for `struct netconfig`. The Perl object will be blessed in a class matching the name of the C type, with the tag `Ptr` appended, and the name should not have embedded spaces if it will be a Perl package name. The destructor will be placed in a class corresponding to the class of the object and the PREFIX keyword will be used to trim the name to the word DESTROY as Perl will expect.

```
     typedef struct netconfig Netconfig;

     MODULE = RPC  PACKAGE = RPC

     Netconfig *
     getnetconfigent(netid)
          char *netid

     MODULE = RPC  PACKAGE = NetconfigPtr  PREFIX = rpcb_

     void
     rpcb_DESTROY(netconf)
          Netconfig *netconf
        CODE:
          printf("Now in NetconfigPtr::DESTROY\n");
          free( netconf );
```

This example requires the following typemap entry. Consult the typemap section for more information about adding new typemaps for an extension.

```
TYPEMAP
Netconfig *  T_PTROBJ
```

This example will be used with the following Perl statements.

```
use RPC;
$netconf = getnetconfigent("udp");
```

When Perl destroys the object referenced by $netconf it will send the object to the supplied XSUB DESTROY function. Perl cannot determine, and does not care, that this object is a C struct and not a Perl object. In this sense, there is no difference between the object created by the getnetconfigent() XSUB and an object created by a normal Perl subroutine.

68.1.45 The Typemap

The typemap is a collection of code fragments which are used by the **xsubpp** compiler to map C function parameters and values to Perl values. The typemap file may consist of three sections labelled `TYPEMAP`, `INPUT`, and `OUTPUT`. An unlabelled initial section is assumed to be a `TYPEMAP` section. The INPUT section tells the compiler how to translate Perl values into variables of certain C types. The OUTPUT section tells the compiler how to translate the values from certain C types into values Perl can understand. The TYPEMAP section tells the compiler which of the INPUT and OUTPUT code fragments should be used to map a given C type to a Perl value. The section labels `TYPEMAP`, `INPUT`, or `OUTPUT` must begin in the first column on a line by themselves, and must be in uppercase.

The default typemap in the `lib/ExtUtils` directory of the Perl source contains many useful types which can be used by Perl extensions. Some extensions define additional typemaps which they keep in their own directory. These additional typemaps may reference INPUT and OUTPUT maps in the main typemap. The **xsubpp** compiler will allow the extension's own typemap to override any mappings which are in the default typemap.

Most extensions which require a custom typemap will need only the TYPEMAP section of the typemap file. The custom typemap used in the getnetconfigent() example shown earlier demonstrates what may be the typical use of extension typemaps. That typemap is used to equate a C structure with the T_PTROBJ typemap. The typemap used by getnetconfigent() is shown here. Note that the C type is separated from the XS type with a tab and that the C unary operator * is considered to be a part of the C type name.

```
TYPEMAP
Netconfig *<tab>T_PTROBJ
```

Here's a more complicated example: suppose that you wanted `struct netconfig` to be blessed into the class `Net::Config`. One way to do this is to use underscores (_) to separate package names, as follows:

```
typedef struct netconfig * Net_Config;
```

And then provide a typemap entry `T_PTROBJ_SPECIAL` that maps underscores to double-colons (::), and declare `Net_Config` to be of that type:

```
TYPEMAP
Net_Config      T_PTROBJ_SPECIAL

INPUT
T_PTROBJ_SPECIAL
        if (sv_derived_from($arg, \"${(my $ntt=$ntype)=~s/_/::/g;\$ntt}\")) {
                IV tmp = SvIV((SV*)SvRV($arg));
        $var = ($type) tmp;
        }
        else
                croak(\"$var is not of type ${(my $ntt=$ntype)=~s/_/::/g;\$ntt}\")
```

```
    OUTPUT
    T_PTROBJ_SPECIAL
            sv_setref_pv($arg, \"${(my $ntt=$ntype)=~s/_/::/g;\$ntt}\",
            (void*)$var);
```

The INPUT and OUTPUT sections substitute underscores for double-colons on the fly, giving the desired effect. This example demonstrates some of the power and versatility of the typemap facility.

68.1.46 Safely Storing Static Data in XS

Starting with Perl 5.8, a macro framework has been defined to allow static data to be safely stored in XS modules that will be accessed from a multi-threaded Perl.

Although primarily designed for use with multi-threaded Perl, the macros have been designed so that they will work with non-threaded Perl as well.

It is therefore strongly recommended that these macros be used by all XS modules that make use of static data.

The easiest way to get a template set of macros to use is by specifying the `-g` (`-global`) option with h2xs (see *h2xs*).

Below is an example module that makes use of the macros.

```
    #include "EXTERN.h"
    #include "perl.h"
    #include "XSUB.h"

    /* Global Data */

    #define MY_CXT_KEY "BlindMice::_guts" XS_VERSION

    typedef struct {
        int count;
        char name[3][100];
    } my_cxt_t;

    START_MY_CXT

    MODULE = BlindMice           PACKAGE = BlindMice

    BOOT:
    {
        MY_CXT_INIT;
        MY_CXT.count = 0;
        strcpy(MY_CXT.name[0], "None");
        strcpy(MY_CXT.name[1], "None");
        strcpy(MY_CXT.name[2], "None");
    }

    int
    newMouse(char * name)
        char * name;
        PREINIT:
          dMY_CXT;
        CODE:
          if (MY_CXT.count >= 3) {
              warn("Already have 3 blind mice") ;
              RETVAL = 0;
          }
          else {
              RETVAL = ++ MY_CXT.count;
              strcpy(MY_CXT.name[MY_CXT.count - 1], name);
          }
```

```
char *
get_mouse_name(index)
  int index
  CODE:
    dMY_CXT;
    RETVAL = MY_CXT.lives ++;
    if (index > MY_CXT.count)
      croak("There are only 3 blind mice.");
    else
      RETVAL = newSVpv(MY_CXT.name[index - 1]);
```

REFERENCE

MY_CXT_KEY

This macro is used to define a unique key to refer to the static data for an XS module. The suggested naming scheme, as used by h2xs, is to use a string that consists of the module name, the string "::_guts" and the module version number.

```
#define MY_CXT_KEY "MyModule::_guts" XS_VERSION
```

typedef my_cxt_t

This struct typedef *must* always be called `my_cxt_t` – the other `CXT*` macros assume the existence of the `my_cxt_t` typedef name.

Declare a typedef named `my_cxt_t` that is a structure that contains all the data that needs to be interpreter-local.

```
typedef struct {
    int some_value;
} my_cxt_t;
```

START_MY_CXT

Always place the START_MY_CXT macro directly after the declaration of `my_cxt_t`.

MY_CXT_INIT

The MY_CXT_INIT macro initialises storage for the `my_cxt_t` struct.

It *must* be called exactly once – typically in a BOOT: section.

dMY_CXT

Use the dMY_CXT macro (a declaration) in all the functions that access MY_CXT.

MY_CXT

Use the MY_CXT macro to access members of the `my_cxt_t` struct. For example, if `my_cxt_t` is

```
typedef struct {
    int index;
} my_cxt_t;
```

then use this to access the `index` member

```
dMY_CXT;
MY_CXT.index = 2;
```

68.2 EXAMPLES

File `RPC.xs`: Interface to some ONC+ RPC bind library functions.

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include <rpc/rpc.h>

typedef struct netconfig Netconfig;

MODULE = RPC  PACKAGE = RPC

SV *
rpcb_gettime(host="localhost")
     char *host
   PREINIT:
     time_t  timep;
   CODE:
     ST(0) = sv_newmortal();
     if( rpcb_gettime( host, &timep ) )
          sv_setnv( ST(0), (double)timep );

Netconfig *
getnetconfigent(netid="udp")
     char *netid

MODULE = RPC  PACKAGE = NetconfigPtr  PREFIX = rpcb_

void
rpcb_DESTROY(netconf)
     Netconfig *netconf
   CODE:
     printf("NetconfigPtr::DESTROY\n");
     free( netconf );
```

File `typemap`: Custom typemap for RPC.xs.

```
TYPEMAP
Netconfig *  T_PTROBJ
```

File `RPC.pm`: Perl module for the RPC extension.

```
package RPC;

require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
@EXPORT = qw(rpcb_gettime getnetconfigent);

bootstrap RPC;
1;
```

File `rpctest.pl`: Perl test program for the RPC extension.

```
use RPC;

$netconf = getnetconfigent();
$a = rpcb_gettime();
print "time = $a\n";
print "netconf = $netconf\n";

$netconf = getnetconfigent("tcp");
$a = rpcb_gettime("poplar");
print "time = $a\n";
print "netconf = $netconf\n";
```

68.3 XS VERSION

This document covers features supported by `xsubpp` 1.935.

68.4 AUTHOR

Originally written by Dean Roehrich <*roehrich@cray.com*>.

Maintained since 1996 by The Perl Porters <*perlbug@perl.org*>.

Chapter 69

perlclib

Internal replacements for standard C library functions

69.1 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.

69.1.1 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.

69.1.2 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)
```

69.1.3 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)
```

69.1.4 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)
```

69.1.5 Memory Management and String Handling

```
    Instead Of:                         Use:

    t* p = malloc(n)                    New(id, p, n, t)
    t* p = calloc(n, s)                 Newz(id, 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/*(struct foo *)              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)
```

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 Newz() 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() macro, which has similar arguments as Zero():

```
    Poison(dst, n, t)
```

69.1.6 Character Class Tests

There are two types of character class tests that Perl implements: one type deals in `char`s and are thus **not** Unicode aware (and hence deprecated unless you **know** you should use them) and the other type deal in `UV`s and know about Unicode properties. In the following table, `c` is a `char`, and `u` is a Unicode codepoint.

```
    Instead Of:                 Use:            But better use:

    isalnum(c)                  isALNUM(c)      isALNUM_uni(u)
    isalpha(c)                  isALPHA(c)      isALPHA_uni(u)
    iscntrl(c)                  isCNTRL(c)      isCNTRL_uni(u)
    isdigit(c)                  isDIGIT(c)      isDIGIT_uni(u)
    isgraph(c)                  isGRAPH(c)      isGRAPH_uni(u)
    islower(c)                  isLOWER(c)      isLOWER_uni(u)
    isprint(c)                  isPRINT(c)      isPRINT_uni(u)
    ispunct(c)                  isPUNCT(c)      isPUNCT_uni(u)
    isspace(c)                  isSPACE(c)      isSPACE_uni(u)
    isupper(c)                  isUPPER(c)      isUPPER_uni(u)
    isxdigit(c)                 isXDIGIT(c)     isXDIGIT_uni(u)

    tolower(c)                  toLOWER(c)      toLOWER_uni(u)
    toupper(c)                  toUPPER(c)      toUPPER_uni(u)
```

69.1.7 *stdlib.h* functions

```
Instead Of:                 Use:

atof(s)                     Atof(s)
atol(s)                     Atol(s)
strtod(s, *p)               Nothing.  Just don't use it.
strtol(s, *p, n)            Strtol(s, *p, n)
strtoul(s, *p, n)           Strtoul(s, *p, n)
```

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.

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_putenv(s, val)
```

69.1.8 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)`.

69.2 SEE ALSO

`perlapi`, `perlapio`, `perlguts`

Chapter 70

perlguts

Introduction to the Perl API

70.1 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.

70.2 Variables

70.2.1 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.

70.2.2 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.

70.2.3 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).

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 initialisation, 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 value undef.

```
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.

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 *perlsec*). 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 Magic Virtual Tables 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 a byte for the a trailing NUL (perl's own string functions typically do `SvGROW(sv, len + 1)`).

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 Magic Virtual Tables 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", FALSE);
```

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 AVs, HVs and undefined values).

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 Reference Counts and Mortality).

70.2.4 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 puts the number of bytes chopped off into the IV field of the SV. It then moves the PV pointer (called SvPVX) forward that many bytes, and adjusts SvCUR and SvLEN.

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:

```
% ./perl -Ilib -MDevel::Peek -le '$a="12345"; $a=~s/.//; Dump($a)'
SV = PVIV(0x8128450) at 0x81340f0
  REFCNT = 1
  FLAGS = (POK,OOK,pPOK)
  IV = 1  (OFFSET)
  PV = 0x8135781 ( "1" . ) "2345"\0
  CUR = 4
  LEN = 5
```

Here the number of bytes chopped off (1) is put into IV, and Devel::Peek::Dump helpfully reminds us that this is an 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.

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 AvLEN. 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*.

70.2.5 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.

The are various ways in which the private and public flags may differ. For example, 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. Another is when numeric conversion has occured 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.

70.2.6 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(I32 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 AVs:

```
    void  av_push(AV*, SV*);
    SV*   av_pop(AV*);
    SV*   av_shift(AV*);
    void  av_unshift(AV*, I32 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:

```
    I32   av_len(AV*);
    SV**  av_fetch(AV*, I32 key, I32 lval);
    SV**  av_store(AV*, I32 key, SV* val);
```

The `av_len` function returns the highest index value in 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.

```
    void  av_clear(AV*);
    void  av_undef(AV*);
    void  av_extend(AV*, I32 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", FALSE);
```

This returns NULL if the variable does not exist.

See Understanding the Magic of Tied Hashes and Arrays for more information on how to use the array access functions on tied arrays.

70.2.7 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 HVs:

```
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.

These two functions check if a hash table entry exists, and 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 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", FALSE);
```

This returns NULL if the variable does not exist.

The hash algorithm is defined in the `PERL_HASH(hash, key, klen)` macro:

```
    hash = 0;
    while (klen--)
        hash = (hash * 33) + *key++;
    hash = hash + (hash >> 5);                 /* after 5.6 */
```

The last step was added in version 5.6 to improve distribution of lower bits in the resulting hash value.

See Understanding the Magic of Tied Hashes and Arrays for more information on how to use the hash access functions on tied hashes.

70.2.8 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.

70.2.9 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. 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.

Other 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_true` or `&PL_sv_false` 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_true` and `&PL_sv_false` 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 );
```

70.2.10 References

References are a special type of scalar that point to other data types (including 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_IV    Scalar
    SVt_NV    Scalar
    SVt_PV    Scalar
    SVt_RV    Scalar
    SVt_PVAV  Array
    SVt_PVHV  Hash
    SVt_PVCV  Code
    SVt_PVGV  Glob (possible a file handle)
    SVt_PVMG  Blessed or Magical Scalar

    See the sv.h header file for more details.
```

70.2.11 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 Stashes and Globs for information on converting class names into stashes.

/* Still under construction */

Upgrades rv to reference if not already one. Creates 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);
```

Copies 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);
```

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, PV iv);
```

Copies 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, PV iv, STRLEN length);
```

Tests whether the SV is blessed into the specified class. It does not check inheritance relationships.

```
        int  sv_isa(SV* sv, const char* name);
```

Tests whether the SV is a reference to a blessed object.

```
        int  sv_isobject(SV* sv);
```

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)) { ... }
```

70.2.12 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", TRUE);
    AV*  get_av("package::varname", TRUE);
    HV*  get_hv("package::varname", TRUE);
```

Notice the use of TRUE 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 `TRUE` 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.

70.2.13 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 *perlcall* 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 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.

70.2.14 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 create)
    HV*  gv_stashsv(SV*, I32 create)
```

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 `create` flag will create a new package if it is set.

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 *perlref*.

70.2.15 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", TRUE);
    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`.

70.2.16 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;
        SV*         mg_obj;
        char*       mg_ptr;
        I32         mg_len;
    };
```

Note this is current as of patchlevel 0, and could change at any time.

70.2.17 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 and `namlen` >= 0 a malloc'd copy of the name is stored in `mg_ptr` field.

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 Magic Virtual Tables 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.

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:

```
    void sv_unmagic(SV *sv, int type);
```

The type argument should be equal to the how value when the SV was initially made magical.

70.2.18 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 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);
```

This MGVTBL structure is set at compile-time in *perl.h* and there are currently 19 types (or 21 with overloading turned on). 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.
```

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 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
A  PERL_MAGIC_overload       vtbl_amagic    %OVERLOAD hash
a  PERL_MAGIC_overload_elem  vtbl_amagicelem %OVERLOAD hash element
c  PERL_MAGIC_overload_table (none)         Holds overload table (AMT)
                                            on stash
B  PERL_MAGIC_bm             vtbl_bm        Boyer-Moore (fast string search)
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_fm        Formline ('compiled' format)
g  PERL_MAGIC_regex_global   vtbl_mglob     m//g target / study()ed string
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
m  PERL_MAGIC_mutex          vtbl_mutex     ???
o  PERL_MAGIC_collxfrm       vtbl_collxfrm  Locale collate 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_qr        precompiled qr// regex
S  PERL_MAGIC_sig            vtbl_sig       %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
v  PERL_MAGIC_vec            vtbl_vec       vec() lvalue
V  PERL_MAGIC_vstring        (none)         v-string scalars
w  PERL_MAGIC_utf8           vtbl_utf8      UTF-8 length+offset cache
x  PERL_MAGIC_substr         vtbl_substr    substr() lvalue
y  PERL_MAGIC_defelem        vtbl_defelem   Shadow "foreach" iterator
                                            variable / smart parameter
                                            vivification
*  PERL_MAGIC_glob           vtbl_glob      GV (typeglob)
#  PERL_MAGIC_arylen         vtbl_arylen    Array length ($#ary)
.  PERL_MAGIC_pos            vtbl_pos       pos() lvalue
<  PERL_MAGIC_backref        vtbl_backref   ???
~  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));
```

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 may also be appropriate to add an I32 'signature' at the top of the private data area and check that.

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.

70.2.19 Finding Magic

```
    MAGIC* mg_find(SV*, int type); /* Finds the magic pointer of that type */
```

This routine returns a pointer to the `MAGIC` structure stored in the SV. If the SV does not have that magical feature, `NULL` is returned. Also, if the SV 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.

70.2.20 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 Calling Perl Routines from within C Programs 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", TRUE);
        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.

70.2.21 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 Returning a Scalar in *perlcall*). 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.

`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.

70.3 Subroutines

70.3.1 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 Reference Counts and Mortality):

```
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)))
```

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 adjust 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 Putting a C value on Perl stack.

For more information, consult *perlxs* and *perlxstut*.

70.3.2 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, register 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 *perlcall*.

70.3.3 Memory Allocation

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 :

```
    New(x, pointer, number, type);
    Newc(x, pointer, number, type, cast);
    Newz(x, pointer, number, type);
```

The first argument `x` was a "magic cookie" that was used to keep track of who called the macro, to help when debugging memory problems. However, the current code makes no use of this feature (most Perl developers now use run-time memory checkers), so this argument can be any number.

The second argument `pointer` should be the name of a variable that will point to the newly allocated memory.

The third and fourth 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 `Newc`, `cast`, should be used if the `pointer` argument is different from the `type` argument.

Unlike the `New` and `Newc` macros, the `Newz` macro calls `memzero` to zero out all the newly allocated memory.

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.

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).

70.3.4 PerlIO

The most recent development releases of Perl has 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 *perlapio*.

70.3.5 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 Scratchpads and recursion 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 XSUBs and the Argument Stack, 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`.

70.3.6 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. 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.

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.

70.3.7 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.

70.4 Compiled code

70.4.1 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.

70.4.2 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, `UNOP`s, have one child, and this is pointed to by the `op_first` field. Binary operators (`BINOP`s) 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 `op_sibling` pointer from the first child to the last.

There are also two 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 Compile pass 2: context propagation) it will still have children in accordance with its former type.

Another way to examine the tree is to use a compiler back-end module, such as *B::Concise*.

70.4.3 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*).

70.4.4 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.

70.4.5 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).

70.4.6 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). These optimizations are done in the subroutine peep(). Optimizations performed at this stage are subject to the same restrictions as in the pass 2.

70.4.7 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.

70.5 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.

70.6 How multiple interpreters and concurrency are supported

70.6.1 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.

Two macros control the major Perl build flavors: MULTIPLICITY and USE_5005THREADS. The MULTIPLICITY build has a C structure that packages all the interpreter state, and there is a similar thread-specific data structure under USE_5005THREADS. In both cases, PERL_IMPLICIT_CONTEXT is also normally defined, and enables the support for passing in a "hidden" first argument that represents all three data structures.

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 Internal Functions.) 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 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.

70.6.2 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.

70.6.3 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 have to do nothing 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 my_private_function(int arg1, int arg2);

        static SV *
        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 my_private_function(pTHX_ int arg1, int arg2);

        static SV *
        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.

70.6.4 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 ...
```

70.6.5 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 and USE_5005THREADS on Windows (see inside iperlsys.h).

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.

70.7 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, you can get at the functions either with or without the `Perl_` prefix, thanks to a bunch of defines that live in *embed.h*. This header file 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.

p

This function has a `Perl_` prefix; ie, it is defined as `Perl_av_fetch`

d

This function has documentation using the `apidoc` feature which we'll look at in a second.

Other available flags are:

s

This is a static function and is defined as `S_whatever`, and usually called within the sources as `whatever(...)`.

n

This does not use `aTHX_` and `pTHX` to pass interpreter context. (See Background and PERL_IMPLICIT_CONTEXT in *perlguts*.)

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).

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.

70.7.1 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.

If you are printing addresses of pointers, use UVxf combined with PTR2UV(), do not use %lx or %p.

70.7.2 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);
```

70.7.3 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.

70.8 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.

70.8.1 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, instead of just one. You can learn more about Unicode at http://www.unicode.org/

70.8.2 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.

The API function `is_utf8_string` can help; it'll tell you if a string contains only valid UTF-8 characters. However, it can't do the work for you. On a character-by-character basis, `is_utf8_char` will tell you whether the current character in a string is valid UTF-8.

70.8.3 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 1...128 are stored in one byte, just like good ol' ASCII. Character 129 is stored as `v194.129`; 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; 192 is `v195.128`. And so it goes on, moving to three bytes at character 2048.

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 can be encoded as a single byte even in UTF-8):

```
U8 *utf;
UV uv;      /* Note: a UV, not a U8, not a char */

if (!UTF8_IS_INVARIANT(*utf))
    /* Must treat this as UTF-8 */
    uv = utf8_to_uv(utf);
else
    /* OK to treat this character as a byte */
    uv = *utf;
```

You can also see in that example that we use `utf8_to_uv` to get the value of the character; the inverse function `uv_to_utf8` is available for putting a UV into UTF-8:

```
    if (!UTF8_IS_INVARIANT(uv))
        /* Must treat this as UTF8 */
        utf8 = uv_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!

70.8.4 How does Perl store UTF-8 strings?

Currently, Perl deals with Unicode strings and non-Unicode strings slightly differently. If a string has been identified as being UTF-8 encoded, Perl will set a flag in the SV, `SVf_UTF8`. 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 Unicode data is not properly distinguished, regular expressions, `length`, `substr` and other string handling operations will have undesirable results.

The problem comes when you have, for instance, a string that isn't flagged is 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 to the PV value; you need 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 UTF-8 flag set, and act accordingly:

```
    p = SvPV(sv, len);
    frobnicate(p);
    nsv = newSVpvn(p, len);
    if (SvUTF8(sv))
        SvUTF8_on(nsv);
```

In fact, your `frobnicate` function should 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 UTF-8 flags, even less right is just passing a `char *` to an XS function.

70.8.5 How do I convert a string to UTF-8?

If you're mixing UTF-8 and non-UTF-8 strings, you might find it necessary to upgrade one of the 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.

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.

70.8.6 Is there anything else I need to know?

Not really. Just remember these things:

- There's no way to tell if a string is UTF-8 or not. You can tell if an SV is UTF-8 by looking at is `SvUTF8` flag. 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_uv` 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 `uv_to_utf8`, unless `UTF8_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. There are tricks you can use to delay deciding whether you need to use a UTF-8 string until you get to a high character - `HALF_UPGRADE` is one of those.

70.9 Custom Operators

Custom operator support is a new 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 `pp_addr` 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` as a key into the `PL_custom_op_descs` and `PL_custom_op_names` hashes. This means you need to enter a name and description for your op at the appropriate place in the `PL_custom_op_names` and `PL_custom_op_descs` hashes.

Forthcoming versions of `B::Generate` (version 1.0 and above) should directly support the creation of custom ops by name; `Opcodes::Custom` will provide functions which make it trivial to "register" custom ops to the Perl interpreter.

70.10 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.

70.11 SEE ALSO

perlapi(1), perlintern(1), perlxs(1), perlembed(1)

Chapter 71

perlcall

Perl calling conventions from C

71.1 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 like for an X windows 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 *perlembed*.

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 *perlguts*.

71.2 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, register 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 FLAG VALUES.

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 *perlobj* for more information on static and virtual methods and Using call_method 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.

71.3 FLAG VALUES

The `flags` parameter in all the *call_** functions is a bit mask which can consist of any combination of the symbols defined below, OR'ed together.

71.3.1 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.

71.3.2 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.

71.3.3 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.

71.3.4 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.

71.3.5 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`.

71.3.6 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.

71.3.7 G_KEEPERR

You may have noticed that using the G_EVAL flag described above will **always** clear the `$@` variable and set it to a string describing the error iff there was an error in the called code. This unqualified resetting of `$@` can be problematic in the reliable identification of errors using the `eval {}` mechanism, because the possibility exists that perl will call other code (end of block processing code, for example) between the time the error causes `$@` to be set within `eval {}`, and the subsequent statement which checks for the value of `$@` gets executed in the user's script.

This scenario will mostly be applicable to code that is meant to be called from within destructors, asynchronous callbacks, signal handlers, `__DIE__` or `__WARN__` hooks, and `tie` functions. In such situations, you will not want to clear `$@` at all, but simply to append any new errors to any existing value of `$@`.

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

When G_KEEPERR is used, any errors in the called code will be prefixed with the string "\t(in cleanup)", and appended to the current value of `$@`.

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.

71.3.8 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*.

71.4 KNOWN PROBLEMS

This section outlines all known problems that exist in the *call_** functions.

1. If you are intending to make use of both the G_EVAL and G_SCALAR flags in your code, use a version of Perl greater than 5.000. There is a bug in version 5.000 of Perl which means that the combination of these two flags will not work as described in the section *FLAG VALUES*.

   Specifically, if the two flags are used when calling a subroutine and that subroutine does not call *die*, the value returned by *call_** will be wrong.

2. In Perl 5.000 and 5.001 there is a problem with using *call_** if the Perl sub you are calling attempts to trap a *die*.

   The symptom of this problem is that the called Perl sub will continue to completion, but whenever it attempts to pass control back to the XSUB, the program will immediately terminate.

   For example, say you want to call this Perl sub

   ```
   sub fred
   {
       eval { die "Fatal Error" ; }
       print "Trapped error: $@\n"
           if $@ ;
   }
   ```

   via this XSUB

   ```
   void
   Call_fred()
       CODE:
       PUSHMARK(SP) ;
       call_pv("fred", G_DISCARD|G_NOARGS) ;
       fprintf(stderr, "back in Call_fred\n") ;
   ```

When `Call_fred` is executed it will print

```
Trapped error: Fatal Error
```

As control never returns to `Call_fred`, the `"back in Call_fred"` string will not get printed.

To work around this problem, you can either upgrade to Perl 5.002 or higher, or use the G_EVAL flag with *call_** as shown below

```
void
Call_fred()
    CODE:
    PUSHMARK(SP) ;
    call_pv("fred", G_EVAL|G_DISCARD|G_NOARGS) ;
    fprintf(stderr, "back in Call_fred\n") ;
```

71.5 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.

71.5.1 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.

71.5.2 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 XSUBs and the Argument Stack in *perlguts* 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 using `{` 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.

71.5.3 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 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 whilst in 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.

71.5.4 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.

71.5.5 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*.

71.5.6 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*.

71.5.7 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))
        {
            STRLEN n_a;
            printf ("Uh oh - %s\n", SvPV(ERRSV, n_a)) ;
            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))
   {
       STRLEN n_a;
       printf ("Uh oh - %s\n", SvPV(ERRSV, n_a)) ;
       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 is tidied up by removing the *undef*.

71.5.8 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;
eval { Foo->new->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 eval 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.

71.5.9 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. For Perl 4 this was adequate, but Perl 5 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 have 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` get 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`.

71.5.10 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.

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

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

```
    $a = new Mine ('red', 'green', 'blue') ;
    call_Method($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*.

71.5.12 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
```

71.5.13 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 disposed of 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.

71.5.14 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.

71.5.15 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 you.

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.

71.5.16 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 eval_pv in *perlapi*). Once this code reference is in hand, it can be mixed in with all the previous examples we’ve shown.

71.6 SEE ALSO

perlxs, *perlguts*, *perlembed*

71.7 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.

71.8 DATE

Version 1.3, 14th Apr 1997

Chapter 72

perlapi

Autogenerated documentation for the perl public API

72.1 DESCRIPTION

This file contains the documentation of the perl public API generated by embed.pl, specifically a listing of functions, macros, flags, and variables that may be used by extension writers. The interfaces of any functions that are not listed here are subject to change without notice. For this reason, blindly using functions listed in proto.h is to be avoided when writing extensions.

Note that all Perl API global variables must be referenced with the `PL_` prefix. Some macros are provided for compatibility with the older, unadorned names, but this support may be disabled in a future release.

The listing is alphabetical, case insensitive.

72.2 "Gimme" Values

GIMME

A backward-compatible version of `GIMME_V` which can only return `G_SCALAR` or `G_ARRAY`; in a void context, it returns `G_SCALAR`. Deprecated. Use `GIMME_V` instead.

```
        U32     GIMME
```

GIMME_V

The XSUB-writer's equivalent to Perl's `wantarray`. Returns `G_VOID`, `G_SCALAR` or `G_ARRAY` for void, scalar or list context, respectively.

```
        U32     GIMME_V
```

G_ARRAY

Used to indicate list context. See `GIMME_V`, `GIMME` and *perlcall*.

G_DISCARD

Indicates that arguments returned from a callback should be discarded. See *perlcall*.

G_EVAL

Used to force a Perl `eval` wrapper around a callback. See *perlcall*.

G_NOARGS

Indicates that no arguments are being sent to a callback. See *perlcall*.

G_SCALAR

Used to indicate scalar context. See `GIMME_V`, `GIMME`, and *perlcall*.

G_VOID

Used to indicate void context. See `GIMME_V` and *perlcall*.

72.3 Array Manipulation Functions

AvFILL

Same as `av_len()`. Deprecated, use `av_len()` instead.

```
        int     AvFILL(AV* av)
```

av_clear

Clears an array, making it empty. Does not free the memory used by the array itself.

```
        void    av_clear(AV* ar)
```

av_delete

Deletes the element indexed by `key` from the array. Returns the deleted element. `flags` is currently ignored.

```
        SV*     av_delete(AV* ar, I32 key, I32 flags)
```

av_exists

Returns true if the element indexed by `key` has been initialized.

This relies on the fact that uninitialized array elements are set to `&PL_sv_undef`.

```
        bool    av_exists(AV* ar, I32 key)
```

av_extend

Pre-extend an array. The `key` is the index to which the array should be extended.

```
        void    av_extend(AV* ar, I32 key)
```

av_fetch

Returns the SV at the specified index in the array. The `key` is the index. If `lval` is set then the fetch will be part of a store. Check that the return value is non-null before dereferencing it to a `SV*`.

See Understanding the Magic of Tied Hashes and Arrays in *perlguts* for more information on how to use this function on tied arrays.

```
        SV**    av_fetch(AV* ar, I32 key, I32 lval)
```

av_fill

Ensure than an array has a given number of elements, equivalent to Perl's `$#array = $fill;`.

```
        void    av_fill(AV* ar, I32 fill)
```

av_len

Returns the highest index in the array. Returns -1 if the array is empty.

```
        I32     av_len(AV* ar)
```

av_make

Creates a new AV and populates it with a list of SVs. The SVs are copied into the array, so they may be freed after the call to av_make. The new AV will have a reference count of 1.

```
        AV*     av_make(I32 size, SV** svp)
```

av_pop

Pops an SV off the end of the array. Returns `&PL_sv_undef` if the array is empty.

```
        SV*     av_pop(AV* ar)
```

av_push

Pushes an SV onto the end of the array. The array will grow automatically to accommodate the addition.

```
        void    av_push(AV* ar, SV* val)
```

av_shift

Shifts an SV off the beginning of the array.

```
        SV*     av_shift(AV* ar)
```

av_store

Stores an SV in an array. The array index is specified as `key`. The return value will be NULL if the operation failed or if the value did not need to be actually stored within the array (as in the case of tied arrays). Otherwise it can be dereferenced to get the original `SV*`. Note that the caller is responsible for suitably incrementing the reference count of `val` before the call, and decrementing it if the function returned NULL.

See Understanding the Magic of Tied Hashes and Arrays in *perlguts* for more information on how to use this function on tied arrays.

```
        SV**    av_store(AV* ar, I32 key, SV* val)
```

av_undef

Undefines the array. Frees the memory used by the array itself.

```
        void    av_undef(AV* ar)
```

av_unshift

Unshift the given number of `undef` values onto the beginning of the array. The array will grow automatically to accommodate the addition. You must then use `av_store` to assign values to these new elements.

```
        void    av_unshift(AV* ar, I32 num)
```

get_av

Returns the AV of the specified Perl array. If `create` is set and the Perl variable does not exist then it will be created. If `create` is not set and the variable does not exist then NULL is returned.

NOTE: the perl_ form of this function is deprecated.

```
        AV*     get_av(const char* name, I32 create)
```

newAV

Creates a new AV. The reference count is set to 1.

```
        AV*     newAV()
```

sortsv

Sort an array. Here is an example:

```
    sortsv(AvARRAY(av), av_len(av)+1, Perl_sv_cmp_locale);
```

See lib/sort.pm for details about controlling the sorting algorithm.

```
        void    sortsv(SV ** array, size_t num_elts, SVCOMPARE_t cmp)
```

72.4 Callback Functions

call_argv

Performs a callback to the specified Perl sub. See *perlcall*.

NOTE: the perl_ form of this function is deprecated.

```
        I32     call_argv(const char* sub_name, I32 flags, char** argv)
```

call_method

Performs a callback to the specified Perl method. The blessed object must be on the stack. See *perlcall*.

NOTE: the perl_ form of this function is deprecated.

```
        I32     call_method(const char* methname, I32 flags)
```

call_pv

Performs a callback to the specified Perl sub. See *perlcall*.

NOTE: the perl_ form of this function is deprecated.

```
        I32     call_pv(const char* sub_name, I32 flags)
```

call_sv

Performs a callback to the Perl sub whose name is in the SV. See *perlcall*.

NOTE: the perl_ form of this function is deprecated.

```
        I32     call_sv(SV* sv, I32 flags)
```

ENTER

Opening bracket on a callback. See `LEAVE` and *perlcall*.

```
                ENTER;
```

eval_pv

Tells Perl to `eval` the given string and return an SV* result.

NOTE: the perl_ form of this function is deprecated.

```
        SV*     eval_pv(const char* p, I32 croak_on_error)
```

eval_sv

Tells Perl to `eval` the string in the SV.

NOTE: the perl_ form of this function is deprecated.

```
        I32     eval_sv(SV* sv, I32 flags)
```

FREETMPS

Closing bracket for temporaries on a callback. See `SAVETMPS` and *perlcall*.

```
                FREETMPS;
```

LEAVE

Closing bracket on a callback. See `ENTER` and *perlcall*.

```
                LEAVE;
```

SAVETMPS

Opening bracket for temporaries on a callback. See `FREETMPS` and *perlcall*.

```
                SAVETMPS;
```

72.5 Character classes

isALNUM

Returns a boolean indicating whether the C `char` is an ASCII alphanumeric character (including underscore) or digit.

```
        bool    isALNUM(char ch)
```

isALPHA

Returns a boolean indicating whether the C `char` is an ASCII alphabetic character.

```
        bool    isALPHA(char ch)
```

isDIGIT

Returns a boolean indicating whether the C `char` is an ASCII digit.

```
        bool    isDIGIT(char ch)
```

isLOWER

Returns a boolean indicating whether the C `char` is a lowercase character.

```
        bool    isLOWER(char ch)
```

isSPACE

Returns a boolean indicating whether the C `char` is whitespace.

```
        bool    isSPACE(char ch)
```

isUPPER

Returns a boolean indicating whether the C `char` is an uppercase character.

```
        bool    isUPPER(char ch)
```

toLOWER

Converts the specified character to lowercase.

```
        char    toLOWER(char ch)
```

toUPPER

Converts the specified character to uppercase.

```
        char    toUPPER(char ch)
```

72.6 Cloning an interpreter

perl_clone

Create and return a new interpreter by cloning the current one.

perl_clone takes these flags as parameters:

CLONEf_COPY_STACKS - is used to, well, copy the stacks also, without it we only clone the data and zero the stacks, with it we copy the stacks and the new perl interpreter is ready to run at the exact same point as the previous one. The pseudo-fork code uses COPY_STACKS while the threads->new doesn't.

CLONEf_KEEP_PTR_TABLE perl_clone keeps a ptr_table with the pointer of the old variable as a key and the new variable as a value, this allows it to check if something has been cloned and not clone it again but rather just use the value and increase the refcount. If KEEP_PTR_TABLE is not set then perl_clone will kill the ptr_table using the function `ptr_table_free(PL_ptr_table); PL_ptr_table = NULL;`, reason to keep it around is if you want to dup some of your own variable who are outside the graph perl scans, example of this code is in threads.xs create

CLONEf_CLONE_HOST This is a win32 thing, it is ignored on unix, it tells perls win32host code (which is c++) to clone itself, this is needed on win32 if you want to run two threads at the same time, if you just want to do some stuff in a separate perl interpreter and then throw it away and return to the original one, you don't need to do anything.

```
        PerlInterpreter*        perl_clone(PerlInterpreter* interp, UV flags)
```

72.7 CV Manipulation Functions

CvSTASH

Returns the stash of the CV.

```
        HV*     CvSTASH(CV* cv)
```

get_cv

Returns the CV of the specified Perl subroutine. If `create` is set and the Perl subroutine does not exist then it will be declared (which has the same effect as saying `sub name;`). If `create` is not set and the subroutine does not exist then NULL is returned.

NOTE: the perl_ form of this function is deprecated.

```
        CV*     get_cv(const char* name, I32 create)
```

72.8 Embedding Functions

cv_undef

Clear out all the active components of a CV. This can happen either by an explicit `undef &foo`, or by the reference count going to zero. In the former case, we keep the CvOUTSIDE pointer, so that any anonymous children can still follow the full lexical scope chain.

```
        void    cv_undef(CV* cv)
```

load_module

Loads the module whose name is pointed to by the string part of name. Note that the actual module name, not its filename, should be given. Eg, "Foo::Bar" instead of "Foo/Bar.pm". flags can be any of PERL_LOADMOD_DENY, PERL_LOADMOD_NOIMPORT, or PERL_LOADMOD_IMPORT_OPS (or 0 for no flags). ver, if specified, provides version semantics similar to `use Foo::Bar VERSION`. The optional trailing SV* arguments can be used to specify arguments to the module's import() method, similar to `use Foo::Bar VERSION LIST`.

```
        void    load_module(U32 flags, SV* name, SV* ver, ...)
```

nothreadhook

Stub that provides thread hook for perl_destruct when there are no threads.

```
        int     nothreadhook()
```

perl_alloc

Allocates a new Perl interpreter. See *perlembed*.

```
        PerlInterpreter*        perl_alloc()
```

perl_construct

Initializes a new Perl interpreter. See *perlembed*.

```
        void    perl_construct(PerlInterpreter* interp)
```

perl_destruct

Shuts down a Perl interpreter. See *perlembed*.

```
        int     perl_destruct(PerlInterpreter* interp)
```

perl_free

Releases a Perl interpreter. See *perlembed*.

```
        void    perl_free(PerlInterpreter* interp)
```

perl_parse

Tells a Perl interpreter to parse a Perl script. See *perlembed*.

```
        int     perl_parse(PerlInterpreter* interp, XSINIT_t xsinit, int argc, char** argv, char**
```

perl_run

Tells a Perl interpreter to run. See *perlembed*.

```
        int     perl_run(PerlInterpreter* interp)
```

require_pv

Tells Perl to `require` the file named by the string argument. It is analogous to the Perl code `eval "require '$file'"`. It's even implemented that way; consider using load_module instead.

NOTE: the perl_ form of this function is deprecated.

```
        void    require_pv(const char* pv)
```

72.9 Functions in file pp_pack.c

packlist

The engine implementing pack() Perl function.

```
        void    packlist(SV *cat, char *pat, char *patend, SV **beglist, SV **endlist)
```

pack_cat

The engine implementing pack() Perl function. Note: parameters next_in_list and flags are not used. This call should not be used; use packlist instead.

```
        void    pack_cat(SV *cat, char *pat, char *patend, SV **beglist, SV **endlist, SV ***next_
```

unpackstring

The engine implementing unpack() Perl function. `unpackstring` puts the extracted list items on the stack and returns the number of elements. Issue `PUTBACK` before and `SPAGAIN` after the call to this function.

```
        I32     unpackstring(char *pat, char *patend, char *s, char *strend, U32 flags)
```

unpack_str

The engine implementing unpack() Perl function. Note: parameters strbeg, new_s and ocnt are not used. This call should not be used, use unpackstring instead.

```
        I32     unpack_str(char *pat, char *patend, char *s, char *strbeg, char *strend, char **ne
```

72.10 Global Variables

PL_modglobal

`PL_modglobal` is a general purpose, interpreter global HV for use by extensions that need to keep information on a per-interpreter basis. In a pinch, it can also be used as a symbol table for extensions to share data among each other. It is a good idea to use keys prefixed by the package name of the extension that owns the data.

```
        HV*     PL_modglobal
```

PL_na

A convenience variable which is typically used with `SvPV` when one doesn't care about the length of the string. It is usually more efficient to either declare a local variable and use that instead or to use the `SvPV_nolen` macro.

```
        STRLEN  PL_na
```

PL_sv_no

This is the `false` SV. See `PL_sv_yes`. Always refer to this as `&PL_sv_no`.

```
SV      PL_sv_no
```

PL_sv_undef

This is the `undef` SV. Always refer to this as `&PL_sv_undef`.

```
SV      PL_sv_undef
```

PL_sv_yes

This is the `true` SV. See `PL_sv_no`. Always refer to this as `&PL_sv_yes`.

```
SV      PL_sv_yes
```

72.11 GV Functions

GvSV

Return the SV from the GV.

```
SV*     GvSV(GV* gv)
```

gv_fetchmeth

Returns the glob with the given `name` and a defined subroutine or `NULL`. The glob lives in the given `stash`, or in the stashes accessible via @ISA and UNIVERSAL::.

The argument `level` should be either 0 or -1. If `level==0`, as a side-effect creates a glob with the given `name` in the given `stash` which in the case of success contains an alias for the subroutine, and sets up caching info for this glob. Similarly for all the searched stashes.

This function grants `"SUPER"` token as a postfix of the stash name. The GV returned from `gv_fetchmeth` may be a method cache entry, which is not visible to Perl code. So when calling `call_sv`, you should not use the GV directly; instead, you should use the method's CV, which can be obtained from the GV with the `GvCV` macro.

```
GV*     gv_fetchmeth(HV* stash, const char* name, STRLEN len, I32 level)
```

gv_fetchmethod

See gv_fetchmethod_autoload.

```
GV*     gv_fetchmethod(HV* stash, const char* name)
```

gv_fetchmethod_autoload

Returns the glob which contains the subroutine to call to invoke the method on the `stash`. In fact in the presence of autoloading this may be the glob for "AUTOLOAD". In this case the corresponding variable $AUTOLOAD is already setup.

The third parameter of `gv_fetchmethod_autoload` determines whether AUTOLOAD lookup is performed if the given method is not present: non-zero means yes, look for AUTOLOAD; zero means no, don't look for AUTOLOAD. Calling `gv_fetchmethod` is equivalent to calling `gv_fetchmethod_autoload` with a non-zero `autoload` parameter.

These functions grant `"SUPER"` token as a prefix of the method name. Note that if you want to keep the returned glob for a long time, you need to check for it being "AUTOLOAD", since at the later time the call may load a different subroutine due to $AUTOLOAD changing its value. Use the glob created via a side effect to do this.

These functions have the same side-effects and as `gv_fetchmeth` with `level==0`. `name` should be writable if contains `':'` or `' '`. The warning against passing the GV returned by `gv_fetchmeth` to `call_sv` apply equally to these functions.

```
        GV*     gv_fetchmethod_autoload(HV* stash, const char* name, I32 autoload)
```

gv_fetchmeth_autoload

Same as gv_fetchmeth(), but looks for autoloaded subroutines too. Returns a glob for the subroutine.

For an autoloaded subroutine without a GV, will create a GV even if `level < 0`. For an autoloaded subroutine without a stub, GvCV() of the result may be zero.

```
        GV*     gv_fetchmeth_autoload(HV* stash, const char* name, STRLEN len, I32 level)
```

gv_stashpv

Returns a pointer to the stash for a specified package. `name` should be a valid UTF-8 string. If `create` is set then the package will be created if it does not already exist. If `create` is not set and the package does not exist then NULL is returned.

```
        HV*     gv_stashpv(const char* name, I32 create)
```

gv_stashsv

Returns a pointer to the stash for a specified package, which must be a valid UTF-8 string. See `gv_stashpv`.

```
        HV*     gv_stashsv(SV* sv, I32 create)
```

72.12 Handy Values

Nullav

Null AV pointer.

Nullch

Null character pointer.

Nullcv

Null CV pointer.

Nullhv

Null HV pointer.

Nullsv

Null SV pointer.

72.13 Hash Manipulation Functions

get_hv

Returns the HV of the specified Perl hash. If `create` is set and the Perl variable does not exist then it will be created. If `create` is not set and the variable does not exist then NULL is returned.

NOTE: the perl_ form of this function is deprecated.

```
        HV*     get_hv(const char* name, I32 create)
```

HEf_SVKEY

This flag, used in the length slot of hash entries and magic structures, specifies the structure contains an `SV*` pointer where a `char*` pointer is to be expected. (For information only–not to be used).

HeHASH

Returns the computed hash stored in the hash entry.

```
U32     HeHASH(HE* he)
```

HeKEY

Returns the actual pointer stored in the key slot of the hash entry. The pointer may be either `char*` or `SV*`, depending on the value of `HeKLEN()`. Can be assigned to. The `HePV()` or `HeSVKEY()` macros are usually preferable for finding the value of a key.

```
void*   HeKEY(HE* he)
```

HeKLEN

If this is negative, and amounts to `HEf_SVKEY`, it indicates the entry holds an `SV*` key. Otherwise, holds the actual length of the key. Can be assigned to. The `HePV()` macro is usually preferable for finding key lengths.

```
STRLEN  HeKLEN(HE* he)
```

HePV

Returns the key slot of the hash entry as a `char*` value, doing any necessary dereferencing of possibly `SV*` keys. The length of the string is placed in `len` (this is a macro, so do *not* use `&len`). If you do not care about what the length of the key is, you may use the global variable `PL_na`, though this is rather less efficient than using a local variable. Remember though, that hash keys in perl are free to contain embedded nulls, so using `strlen()` or similar is not a good way to find the length of hash keys. This is very similar to the `SvPV()` macro described elsewhere in this document.

```
char*   HePV(HE* he, STRLEN len)
```

HeSVKEY

Returns the key as an `SV*`, or `Nullsv` if the hash entry does not contain an `SV*` key.

```
SV*     HeSVKEY(HE* he)
```

HeSVKEY_force

Returns the key as an `SV*`. Will create and return a temporary mortal `SV*` if the hash entry contains only a `char*` key.

```
SV*     HeSVKEY_force(HE* he)
```

HeSVKEY_set

Sets the key to a given `SV*`, taking care to set the appropriate flags to indicate the presence of an `SV*` key, and returns the same `SV*`.

```
SV*     HeSVKEY_set(HE* he, SV* sv)
```

HeVAL

Returns the value slot (type `SV*`) stored in the hash entry.

```
SV*     HeVAL(HE* he)
```

HvNAME

Returns the package name of a stash. See `SvSTASH`, `CvSTASH`.

```
        char*   HvNAME(HV* stash)
```

hv_clear

Clears a hash, making it empty.

```
        void    hv_clear(HV* tb)
```

hv_clear_placeholders

Clears any placeholders from a hash. If a restricted hash has any of its keys marked as readonly and the key is subsequently deleted, the key is not actually deleted but is marked by assigning it a value of &PL_sv_placeholder. This tags it so it will be ignored by future operations such as iterating over the hash, but will still allow the hash to have a value reaasigned to the key at some future point. This function clears any such placeholder keys from the hash. See Hash::Util::lock_keys() for an example of its use.

```
        void    hv_clear_placeholders(HV* hb)
```

hv_delete

Deletes a key/value pair in the hash. The value SV is removed from the hash and returned to the caller. The `klen` is the length of the key. The `flags` value will normally be zero; if set to G_DISCARD then NULL will be returned.

```
        SV*     hv_delete(HV* tb, const char* key, I32 klen, I32 flags)
```

hv_delete_ent

Deletes a key/value pair in the hash. The value SV is removed from the hash and returned to the caller. The `flags` value will normally be zero; if set to G_DISCARD then NULL will be returned. `hash` can be a valid precomputed hash value, or 0 to ask for it to be computed.

```
        SV*     hv_delete_ent(HV* tb, SV* key, I32 flags, U32 hash)
```

hv_exists

Returns a boolean indicating whether the specified hash key exists. The `klen` is the length of the key.

```
        bool    hv_exists(HV* tb, const char* key, I32 klen)
```

hv_exists_ent

Returns a boolean indicating whether the specified hash key exists. `hash` can be a valid precomputed hash value, or 0 to ask for it to be computed.

```
        bool    hv_exists_ent(HV* tb, SV* key, U32 hash)
```

hv_fetch

Returns the SV which corresponds to the specified key in the hash. The `klen` is the length of the key. If `lval` is set then the fetch will be part of a store. Check that the return value is non-null before dereferencing it to an `SV*`.

See Understanding the Magic of Tied Hashes and Arrays in *perlguts* for more information on how to use this function on tied hashes.

```
        SV**    hv_fetch(HV* tb, const char* key, I32 klen, I32 lval)
```

hv_fetch_ent

Returns the hash entry which corresponds to the specified key in the hash. `hash` must be a valid precomputed hash number for the given `key`, or 0 if you want the function to compute it. IF `lval` is set then the fetch will be part of a store. Make sure the return value is non-null before accessing it. The return value when `tb` is a tied hash is a pointer to a static location, so be sure to make a copy of the structure if you need to store it somewhere.

See Understanding the Magic of Tied Hashes and Arrays in *perlguts* for more information on how to use this function on tied hashes.

```
HE*     hv_fetch_ent(HV* tb, SV* key, I32 lval, U32 hash)
```

hv_iterinit

Prepares a starting point to traverse a hash table. Returns the number of keys in the hash (i.e. the same as `HvKEYS(tb)`). The return value is currently only meaningful for hashes without tie magic.

NOTE: Before version 5.004_65, `hv_iterinit` used to return the number of hash buckets that happen to be in use. If you still need that esoteric value, you can get it through the macro `HvFILL(tb)`.

```
I32     hv_iterinit(HV* tb)
```

hv_iterkey

Returns the key from the current position of the hash iterator. See `hv_iterinit`.

```
char*   hv_iterkey(HE* entry, I32* retlen)
```

hv_iterkeysv

Returns the key as an `SV*` from the current position of the hash iterator. The return value will always be a mortal copy of the key. Also see `hv_iterinit`.

```
SV*     hv_iterkeysv(HE* entry)
```

hv_iternext

Returns entries from a hash iterator. See `hv_iterinit`.

You may call `hv_delete` or `hv_delete_ent` on the hash entry that the iterator currently points to, without losing your place or invalidating your iterator. Note that in this case the current entry is deleted from the hash with your iterator holding the last reference to it. Your iterator is flagged to free the entry on the next call to `hv_iternext`, so you must not discard your iterator immediately else the entry will leak - call `hv_iternext` to trigger the resource deallocation.

```
HE*     hv_iternext(HV* tb)
```

hv_iternextsv

Performs an `hv_iternext`, `hv_iterkey`, and `hv_iterval` in one operation.

```
SV*     hv_iternextsv(HV* hv, char** key, I32* retlen)
```

hv_iternext_flags

Returns entries from a hash iterator. See `hv_iterinit` and `hv_iternext`. The `flags` value will normally be zero; if HV_ITERNEXT_WANTPLACEHOLDERS is set the placeholders keys (for restricted hashes) will be returned in addition to normal keys. By default placeholders are automatically skipped over. Currently a placeholder is implemented with a value that is `&Perl_sv_placeholder`. Note that the implementation of placeholders and restricted hashes may change, and the implementation currently is insufficiently abstracted for any change to be tidy.

NOTE: this function is experimental and may change or be removed without notice.

```
HE*     hv_iternext_flags(HV* tb, I32 flags)
```

hv_iterval

Returns the value from the current position of the hash iterator. See `hv_iterkey`.

```
SV*     hv_iterval(HV* tb, HE* entry)
```

hv_magic

Adds magic to a hash. See `sv_magic`.

```
        void    hv_magic(HV* hv, GV* gv, int how)
```

hv_scalar

Evaluates the hash in scalar context and returns the result. Handles magic when the hash is tied.

```
        SV*     hv_scalar(HV* hv)
```

hv_store

Stores an SV in a hash. The hash key is specified as `key` and `klen` is the length of the key. The `hash` parameter is the precomputed hash value; if it is zero then Perl will compute it. The return value will be NULL if the operation failed or if the value did not need to be actually stored within the hash (as in the case of tied hashes). Otherwise it can be dereferenced to get the original `SV*`. Note that the caller is responsible for suitably incrementing the reference count of `val` before the call, and decrementing it if the function returned NULL. Effectively a successful hv_store takes ownership of one reference to `val`. This is usually what you want; a newly created SV has a reference count of one, so if all your code does is create SVs then store them in a hash, hv_store will own the only reference to the new SV, and your code doesn't need to do anything further to tidy up. hv_store is not implemented as a call to hv_store_ent, and does not create a temporary SV for the key, so if your key data is not already in SV form then use hv_store in preference to hv_store_ent.

See Understanding the Magic of Tied Hashes and Arrays in *perlguts* for more information on how to use this function on tied hashes.

```
        SV**    hv_store(HV* tb, const char* key, I32 klen, SV* val, U32 hash)
```

hv_store_ent

Stores `val` in a hash. The hash key is specified as `key`. The `hash` parameter is the precomputed hash value; if it is zero then Perl will compute it. The return value is the new hash entry so created. It will be NULL if the operation failed or if the value did not need to be actually stored within the hash (as in the case of tied hashes). Otherwise the contents of the return value can be accessed using the `He?` macros described here. Note that the caller is responsible for suitably incrementing the reference count of `val` before the call, and decrementing it if the function returned NULL. Effectively a successful hv_store_ent takes ownership of one reference to `val`. This is usually what you want; a newly created SV has a reference count of one, so if all your code does is create SVs then store them in a hash, hv_store will own the only reference to the new SV, and your code doesn't need to do anything further to tidy up. Note that hv_store_ent only reads the `key`; unlike `val` it does not take ownership of it, so maintaining the correct reference count on `key` is entirely the caller's responsibility. hv_store is not implemented as a call to hv_store_ent, and does not create a temporary SV for the key, so if your key data is not already in SV form then use hv_store in preference to hv_store_ent.

See Understanding the Magic of Tied Hashes and Arrays in *perlguts* for more information on how to use this function on tied hashes.

```
        HE*     hv_store_ent(HV* tb, SV* key, SV* val, U32 hash)
```

hv_undef

Undefines the hash.

```
        void    hv_undef(HV* tb)
```

newHV

Creates a new HV. The reference count is set to 1.

```
        HV*     newHV()
```

72.14 Magical Functions

mg_clear

Clear something magical that the SV represents. See `sv_magic`.

```
        int     mg_clear(SV* sv)
```

mg_copy

Copies the magic from one SV to another. See `sv_magic`.

```
        int     mg_copy(SV* sv, SV* nsv, const char* key, I32 klen)
```

mg_find

Finds the magic pointer for type matching the SV. See `sv_magic`.

```
        MAGIC*  mg_find(SV* sv, int type)
```

mg_free

Free any magic storage used by the SV. See `sv_magic`.

```
        int     mg_free(SV* sv)
```

mg_get

Do magic after a value is retrieved from the SV. See `sv_magic`.

```
        int     mg_get(SV* sv)
```

mg_length

Report on the SV's length. See `sv_magic`.

```
        U32     mg_length(SV* sv)
```

mg_magical

Turns on the magical status of an SV. See `sv_magic`.

```
        void    mg_magical(SV* sv)
```

mg_set

Do magic after a value is assigned to the SV. See `sv_magic`.

```
        int     mg_set(SV* sv)
```

SvGETMAGIC

Invokes `mg_get` on an SV if it has 'get' magic. This macro evaluates its argument more than once.

```
        void    SvGETMAGIC(SV* sv)
```

SvLOCK

Arranges for a mutual exclusion lock to be obtained on sv if a suitable module has been loaded.

```
        void    SvLOCK(SV* sv)
```

SvSETMAGIC

Invokes `mg_set` on an SV if it has 'set' magic. This macro evaluates its argument more than once.

```
        void    SvSETMAGIC(SV* sv)
```

SvSetMagicSV

Like `SvSetSV`, but does any set magic required afterwards.

```
        void    SvSetMagicSV(SV* dsb, SV* ssv)
```

SvSetMagicSV_nosteal

Like `SvSetMagicSV`, but does any set magic required afterwards.

```
        void    SvSetMagicSV_nosteal(SV* dsv, SV* ssv)
```

SvSetSV

Calls `sv_setsv` if dsv is not the same as ssv. May evaluate arguments more than once.

```
        void    SvSetSV(SV* dsb, SV* ssv)
```

SvSetSV_nosteal

Calls a non-destructive version of `sv_setsv` if dsv is not the same as ssv. May evaluate arguments more than once.

```
        void    SvSetSV_nosteal(SV* dsv, SV* ssv)
```

SvSHARE

Arranges for sv to be shared between threads if a suitable module has been loaded.

```
        void    SvSHARE(SV* sv)
```

SvUNLOCK

Releases a mutual exclusion lock on sv if a suitable module has been loaded.

```
        void    SvUNLOCK(SV* sv)
```

72.15 Memory Management

Copy

The XSUB-writer's interface to the C `memcpy` function. The `src` is the source, `dest` is the destination, `nitems` is the number of items, and `type` is the type. May fail on overlapping copies. See also `Move`.

```
        void    Copy(void* src, void* dest, int nitems, type)
```

Move

The XSUB-writer's interface to the C `memmove` function. The `src` is the source, `dest` is the destination, `nitems` is the number of items, and `type` is the type. Can do overlapping moves. See also `Copy`.

```
        void    Move(void* src, void* dest, int nitems, type)
```

New

The XSUB-writer's interface to the C `malloc` function.

```
        void    New(int id, void* ptr, int nitems, type)
```

Newc

The XSUB-writer's interface to the C `malloc` function, with cast.

```
        void    Newc(int id, void* ptr, int nitems, type, cast)
```

Newz

The XSUB-writer's interface to the C `malloc` function. The allocated memory is zeroed with `memzero`.

```
        void    Newz(int id, void* ptr, int nitems, type)
```

Poison

Fill up memory with a pattern (byte 0xAB over and over again) that hopefully catches attempts to access uninitialized memory.

```
        void    Poison(void* dest, int nitems, type)
```

Renew

The XSUB-writer's interface to the C `realloc` function.

```
        void    Renew(void* ptr, int nitems, type)
```

Renewc

The XSUB-writer's interface to the C `realloc` function, with cast.

```
        void    Renewc(void* ptr, int nitems, type, cast)
```

Safefree

The XSUB-writer's interface to the C `free` function.

```
        void    Safefree(void* ptr)
```

savepv

Perl's version of `strdup()`. Returns a pointer to a newly allocated string which is a duplicate of `pv`. The size of the string is determined by `strlen()`. The memory allocated for the new string can be freed with the `Safefree()` function.

```
        char*   savepv(const char* pv)
```

savepvn

Perl's version of what `strndup()` would be if it existed. Returns a pointer to a newly allocated string which is a duplicate of the first `len` bytes from `pv`. The memory allocated for the new string can be freed with the `Safefree()` function.

```
        char*   savepvn(const char* pv, I32 len)
```

savesharedpv

A version of `savepv()` which allocates the duplicate string in memory which is shared between threads.

```
        char*   savesharedpv(const char* pv)
```

StructCopy

This is an architecture-independent macro to copy one structure to another.

```
        void    StructCopy(type src, type dest, type)
```

Zero

The XSUB-writer's interface to the C `memzero` function. The `dest` is the destination, `nitems` is the number of items, and `type` is the type.

```
        void    Zero(void* dest, int nitems, type)
```

72.16 Miscellaneous Functions

fbm_compile

Analyses the string in order to make fast searches on it using fbm_instr() – the Boyer-Moore algorithm.

```
        void    fbm_compile(SV* sv, U32 flags)
```

fbm_instr

Returns the location of the SV in the string delimited by `str` and `strend`. It returns `Nullch` if the string can't be found. The `sv` does not have to be fbm_compiled, but the search will not be as fast then.

```
        char*   fbm_instr(unsigned char* big, unsigned char* bigend, SV* littlesv, U32 flags)
```

form

Takes a sprintf-style format pattern and conventional (non-SV) arguments and returns the formatted string.

```
    (char *) Perl_form(pTHX_ const char* pat, ...)
```

can be used any place a string (char *) is required:

```
    char * s = Perl_form("%d.%d",major,minor);
```

Uses a single private buffer so if you want to format several strings you must explicitly copy the earlier strings away (and free the copies when you are done).

```
        char*   form(const char* pat, ...)
```

getcwd_sv

Fill the sv with current working directory

```
        int     getcwd_sv(SV* sv)
```

strEQ

Test two strings to see if they are equal. Returns true or false.

```
        bool    strEQ(char* s1, char* s2)
```

strGE

Test two strings to see if the first, `s1`, is greater than or equal to the second, `s2`. Returns true or false.

```
        bool    strGE(char* s1, char* s2)
```

strGT

Test two strings to see if the first, `s1`, is greater than the second, `s2`. Returns true or false.

```
        bool    strGT(char* s1, char* s2)
```

strLE

Test two strings to see if the first, `s1`, is less than or equal to the second, `s2`. Returns true or false.

```
        bool    strLE(char* s1, char* s2)
```

strLT

Test two strings to see if the first, `s1`, is less than the second, `s2`. Returns true or false.

```
        bool    strLT(char* s1, char* s2)
```

strNE

Test two strings to see if they are different. Returns true or false.

```
        bool    strNE(char* s1, char* s2)
```

strnEQ

Test two strings to see if they are equal. The `len` parameter indicates the number of bytes to compare. Returns true or false. (A wrapper for `strncmp`).

```
        bool    strnEQ(char* s1, char* s2, STRLEN len)
```

strnNE

Test two strings to see if they are different. The `len` parameter indicates the number of bytes to compare. Returns true or false. (A wrapper for `strncmp`).

```
        bool    strnNE(char* s1, char* s2, STRLEN len)
```

sv_nolocking

Dummy routine which "locks" an SV when there is no locking module present. Exists to avoid test for a NULL function pointer and because it could potentially warn under some level of strict-ness.

```
        void    sv_nolocking(SV *)
```

sv_nosharing

Dummy routine which "shares" an SV when there is no sharing module present. Exists to avoid test for a NULL function pointer and because it could potentially warn under some level of strict-ness.

```
        void    sv_nosharing(SV *)
```

sv_nounlocking

Dummy routine which "unlocks" an SV when there is no locking module present. Exists to avoid test for a NULL function pointer and because it could potentially warn under some level of strict-ness.

```
        void    sv_nounlocking(SV *)
```

72.17 Numeric functions

grok_bin

converts a string representing a binary number to numeric form.

On entry *start* and **len* give the string to scan, **flags* gives conversion flags, and *result* should be NULL or a pointer to an NV. The scan stops at the end of the string, or the first invalid character. On return **len* is set to the length scanned string, and **flags* gives output flags.

If the value is <= UV_MAX it is returned as a UV, the output flags are clear, and nothing is written to **result*. If the value is > UV_MAX `grok_bin` returns UV_MAX, sets `PERL_SCAN_GREATER_THAN_UV_MAX` in the output flags, and writes the value to **result* (or the value is discarded if *result* is NULL).

The hex number may optionally be prefixed with "0b" or "b" unless `PERL_SCAN_DISALLOW_PREFIX` is set in **flags* on entry. If `PERL_SCAN_ALLOW_UNDERSCORES` is set in **flags* then the binary number may use '_' characters to separate digits.

```
        UV      grok_bin(char* start, STRLEN* len, I32* flags, NV *result)
```

grok_hex

converts a string representing a hex number to numeric form.

On entry *start* and **len* give the string to scan, **flags* gives conversion flags, and *result* should be NULL or a pointer to an NV. The scan stops at the end of the string, or the first non-hex-digit character. On return **len* is set to the length scanned string, and **flags* gives output flags.

If the value is <= UV_MAX it is returned as a UV, the output flags are clear, and nothing is written to **result*. If the value is > UV_MAX `grok_hex` returns UV_MAX, sets `PERL_SCAN_GREATER_THAN_UV_MAX` in the output flags, and writes the value to **result* (or the value is discarded if *result* is NULL).

The hex number may optionally be prefixed with "0x" or "x" unless `PERL_SCAN_DISALLOW_PREFIX` is set in **flags* on entry. If `PERL_SCAN_ALLOW_UNDERSCORES` is set in **flags* then the hex number may use '_' characters to separate digits.

```
        UV      grok_hex(char* start, STRLEN* len, I32* flags, NV *result)
```

grok_number

Recognise (or not) a number. The type of the number is returned (0 if unrecognised), otherwise it is a bit-ORed combination of IS_NUMBER_IN_UV, IS_NUMBER_GREATER_THAN_UV_MAX, IS_NUMBER_NOT_INT, IS_NUMBER_NEG, IS_NUMBER_INFINITY, IS_NUMBER_NAN (defined in perl.h).

If the value of the number can fit an in UV, it is returned in the *valuep IS_NUMBER_IN_UV will be set to indicate that *valuep is valid, IS_NUMBER_IN_UV will never be set unless *valuep is valid, but *valuep may have been assigned to during processing even though IS_NUMBER_IN_UV is not set on return. If valuep is NULL, IS_NUMBER_IN_UV will be set for the same cases as when valuep is non-NULL, but no actual assignment (or SEGV) will occur.

IS_NUMBER_NOT_INT will be set with IS_NUMBER_IN_UV if trailing decimals were seen (in which case *valuep gives the true value truncated to an integer), and IS_NUMBER_NEG if the number is negative (in which case *valuep holds the absolute value). IS_NUMBER_IN_UV is not set if e notation was used or the number is larger than a UV.

```
        int     grok_number(const char *pv, STRLEN len, UV *valuep)
```

grok_numeric_radix

Scan and skip for a numeric decimal separator (radix).

```
        bool    grok_numeric_radix(const char **sp, const char *send)
```

grok_oct

```
        UV      grok_oct(char* start, STRLEN* len, I32* flags, NV *result)
```

scan_bin

For backwards compatibility. Use `grok_bin` instead.

```
        NV      scan_bin(char* start, STRLEN len, STRLEN* retlen)
```

scan_hex

For backwards compatibility. Use `grok_hex` instead.

```
        NV      scan_hex(char* start, STRLEN len, STRLEN* retlen)
```

scan_oct

For backwards compatibility. Use `grok_oct` instead.

```
        NV      scan_oct(char* start, STRLEN len, STRLEN* retlen)
```

72.18 Optree Manipulation Functions

cv_const_sv

If `cv` is a constant sub eligible for inlining. returns the constant value returned by the sub. Otherwise, returns NULL.

Constant subs can be created with `newCONSTSUB` or as described in Constant Functions in *perlsub*.

```
        SV*     cv_const_sv(CV* cv)
```

newCONSTSUB

Creates a constant sub equivalent to Perl `sub FOO () { 123 }` which is eligible for inlining at compile-time.

```
        CV*     newCONSTSUB(HV* stash, char* name, SV* sv)
```

newXS

Used by `xsubpp` to hook up XSUBs as Perl subs.

72.19 Pad Data Structures

pad_sv

Get the value at offset po in the current pad. Use macro PAD_SV instead of calling this function directly.

```
        SV*     pad_sv(PADOFFSET po)
```

72.20 Stack Manipulation Macros

dMARK

Declare a stack marker variable, `mark`, for the XSUB. See `MARK` and `dORIGMARK`.

```
                dMARK;
```

dORIGMARK

Saves the original stack mark for the XSUB. See `ORIGMARK`.

```
                dORIGMARK;
```

dSP

Declares a local copy of perl's stack pointer for the XSUB, available via the `SP` macro. See `SP`.

```
                dSP;
```

EXTEND

Used to extend the argument stack for an XSUB's return values. Once used, guarantees that there is room for at least `nitems` to be pushed onto the stack.

```
        void    EXTEND(SP, int nitems)
```

MARK

Stack marker variable for the XSUB. See `dMARK`.

mPUSHi

Push an integer onto the stack. The stack must have room for this element. Handles 'set' magic. Does not use `TARG`. See also `PUSHi`, `mXPUSHi` and `XPUSHi`.

```
        void    mPUSHi(IV iv)
```

mPUSHn

Push a double onto the stack. The stack must have room for this element. Handles 'set' magic. Does not use `TARG`. See also `PUSHn`, `mXPUSHn` and `XPUSHn`.

```
        void    mPUSHn(NV nv)
```

mPUSHp

Push a string onto the stack. The stack must have room for this element. The `len` indicates the length of the string. Handles 'set' magic. Does not use `TARG`. See also `PUSHp`, `mXPUSHp` and `XPUSHp`.

```
        void    mPUSHp(char* str, STRLEN len)
```

mPUSHu

Push an unsigned integer onto the stack. The stack must have room for this element. Handles 'set' magic. Does not use `TARG`. See also `PUSHu`, `mXPUSHu` and `XPUSHu`.

```
        void    mPUSHu(UV uv)
```

mXPUSHi

Push an integer onto the stack, extending the stack if necessary. Handles 'set' magic. Does not use `TARG`. See also `XPUSHi`, `mPUSHi` and `PUSHi`.

```
        void    mXPUSHi(IV iv)
```

mXPUSHn

Push a double onto the stack, extending the stack if necessary. Handles 'set' magic. Does not use `TARG`. See also `XPUSHn`, `mPUSHn` and `PUSHn`.

```
        void    mXPUSHn(NV nv)
```

mXPUSHp

Push a string onto the stack, extending the stack if necessary. The `len` indicates the length of the string. Handles 'set' magic. Does not use `TARG`. See also `XPUSHp`, `mPUSHp` and `PUSHp`.

```
        void    mXPUSHp(char* str, STRLEN len)
```

mXPUSHu

Push an unsigned integer onto the stack, extending the stack if necessary. Handles 'set' magic. Does not use `TARG`. See also `XPUSHu`, `mPUSHu` and `PUSHu`.

```
        void    mXPUSHu(UV uv)
```

ORIGMARK

The original stack mark for the XSUB. See `dORIGMARK`.

POPi

Pops an integer off the stack.

```
        IV      POPi
```

POPl

Pops a long off the stack.

```
        long    POPl
```

POPn

Pops a double off the stack.

```
        NV      POPn
```

POPp

Pops a string off the stack. Deprecated. New code should provide a STRLEN n_a and use POPpx.

```
        char*   POPp
```

POPpbytex

Pops a string off the stack which must consist of bytes i.e. characters < 256. Requires a variable STRLEN n_a in scope.

```
        char*   POPpbytex
```

POPpx

Pops a string off the stack. Requires a variable STRLEN n_a in scope.

```
char*   POPpx
```

POPs

Pops an SV off the stack.

```
SV*     POPs
```

PUSHi

Push an integer onto the stack. The stack must have room for this element. Handles 'set' magic. Uses `TARG`, so `dTARGET` or `dXSTARG` should be called to declare it. Do not call multiple `TARG`-oriented macros to return lists from XSUB's - see `mPUSHi` instead. See also `XPUSHi` and `mXPUSHi`.

```
void    PUSHi(IV iv)
```

PUSHMARK

Opening bracket for arguments on a callback. See `PUTBACK` and *perlcall*.

```
void    PUSHMARK(SP)
```

PUSHmortal

Push a new mortal SV onto the stack. The stack must have room for this element. Does not handle 'set' magic. Does not use `TARG`. See also `PUSHs`, `XPUSHmortal` and `XPUSHs`.

```
void    PUSHmortal()
```

PUSHn

Push a double onto the stack. The stack must have room for this element. Handles 'set' magic. Uses `TARG`, so `dTARGET` or `dXSTARG` should be called to declare it. Do not call multiple `TARG`-oriented macros to return lists from XSUB's - see `mPUSHn` instead. See also `XPUSHn` and `mXPUSHn`.

```
void    PUSHn(NV nv)
```

PUSHp

Push a string onto the stack. The stack must have room for this element. The `len` indicates the length of the string. Handles 'set' magic. Uses `TARG`, so `dTARGET` or `dXSTARG` should be called to declare it. Do not call multiple `TARG`-oriented macros to return lists from XSUB's - see `mPUSHp` instead. See also `XPUSHp` and `mXPUSHp`.

```
void    PUSHp(char* str, STRLEN len)
```

PUSHs

Push an SV onto the stack. The stack must have room for this element. Does not handle 'set' magic. Does not use `TARG`. See also `PUSHmortal`, `XPUSHs` and `XPUSHmortal`.

```
void    PUSHs(SV* sv)
```

PUSHu

Push an unsigned integer onto the stack. The stack must have room for this element. Handles 'set' magic. Uses `TARG`, so `dTARGET` or `dXSTARG` should be called to declare it. Do not call multiple `TARG`-oriented macros to return lists from XSUB's - see `mPUSHu` instead. See also `XPUSHu` and `mXPUSHu`.

```
        void    PUSHu(UV uv)
```

PUTBACK

Closing bracket for XSUB arguments. This is usually handled by `xsubpp`. See `PUSHMARK` and *perlcall* for other uses.

```
                PUTBACK;
```

SP

Stack pointer. This is usually handled by `xsubpp`. See `dSP` and `SPAGAIN`.

SPAGAIN

Refetch the stack pointer. Used after a callback. See *perlcall*.

```
                SPAGAIN;
```

XPUSHi

Push an integer onto the stack, extending the stack if necessary. Handles 'set' magic. Uses `TARG`, so `dTARGET` or `dXSTARG` should be called to declare it. Do not call multiple `TARG`-oriented macros to return lists from XSUB's - see `mXPUSHi` instead. See also `PUSHi` and `mPUSHi`.

```
        void    XPUSHi(IV iv)
```

XPUSHmortal

Push a new mortal SV onto the stack, extending the stack if necessary. Does not handle 'set' magic. Does not use `TARG`. See also `XPUSHs`, `PUSHmortal` and `PUSHs`.

```
        void    XPUSHmortal()
```

XPUSHn

Push a double onto the stack, extending the stack if necessary. Handles 'set' magic. Uses `TARG`, so `dTARGET` or `dXSTARG` should be called to declare it. Do not call multiple `TARG`-oriented macros to return lists from XSUB's - see `mXPUSHn` instead. See also `PUSHn` and `mPUSHn`.

```
        void    XPUSHn(NV nv)
```

XPUSHp

Push a string onto the stack, extending the stack if necessary. The `len` indicates the length of the string. Handles 'set' magic. Uses `TARG`, so `dTARGET` or `dXSTARG` should be called to declare it. Do not call multiple `TARG`-oriented macros to return lists from XSUB's - see `mXPUSHp` instead. See also `PUSHp` and `mPUSHp`.

```
        void    XPUSHp(char* str, STRLEN len)
```

XPUSHs

Push an SV onto the stack, extending the stack if necessary. Does not handle 'set' magic. Does not use `TARG`. See also `XPUSHmortal`, `PUSHs` and `PUSHmortal`.

```
        void    XPUSHs(SV* sv)
```

XPUSHu

Push an unsigned integer onto the stack, extending the stack if necessary. Handles 'set' magic. Uses `TARG`, so `dTARGET` or `dXSTARG` should be called to declare it. Do not call multiple `TARG`-oriented macros to return lists from XSUB's - see `mXPUSHu` instead. See also `PUSHu` and `mPUSHu`.

```
        void    XPUSHu(UV uv)
```

XSRETURN

Return from XSUB, indicating number of items on the stack. This is usually handled by `xsubpp`.

```
        void    XSRETURN(int nitems)
```

XSRETURN_EMPTY

Return an empty list from an XSUB immediately.

```
                XSRETURN_EMPTY;
```

XSRETURN_IV

Return an integer from an XSUB immediately. Uses `XST_mIV`.

```
        void    XSRETURN_IV(IV iv)
```

XSRETURN_NO

Return `&PL_sv_no` from an XSUB immediately. Uses `XST_mNO`.

```
                XSRETURN_NO;
```

XSRETURN_NV

Return a double from an XSUB immediately. Uses `XST_mNV`.

```
        void    XSRETURN_NV(NV nv)
```

XSRETURN_PV

Return a copy of a string from an XSUB immediately. Uses `XST_mPV`.

```
        void    XSRETURN_PV(char* str)
```

XSRETURN_UNDEF

Return `&PL_sv_undef` from an XSUB immediately. Uses `XST_mUNDEF`.

```
                XSRETURN_UNDEF;
```

XSRETURN_UV

Return an integer from an XSUB immediately. Uses `XST_mUV`.

```
        void    XSRETURN_UV(IV uv)
```

XSRETURN_YES

Return `&PL_sv_yes` from an XSUB immediately. Uses `XST_mYES`.

```
                XSRETURN_YES;
```

XST_mIV

Place an integer into the specified position `pos` on the stack. The value is stored in a new mortal SV.

```
        void    XST_mIV(int pos, IV iv)
```

XST_mNO

Place `&PL_sv_no` into the specified position `pos` on the stack.

```
        void    XST_mNO(int pos)
```

XST_mNV

Place a double into the specified position `pos` on the stack. The value is stored in a new mortal SV.

```
        void    XST_mNV(int pos, NV nv)
```

XST_mPV

Place a copy of a string into the specified position `pos` on the stack. The value is stored in a new mortal SV.

```
        void    XST_mPV(int pos, char* str)
```

XST_mUNDEF

Place `&PL_sv_undef` into the specified position `pos` on the stack.

```
        void    XST_mUNDEF(int pos)
```

XST_mYES

Place `&PL_sv_yes` into the specified position `pos` on the stack.

```
        void    XST_mYES(int pos)
```

72.21 SV Flags

svtype

An enum of flags for Perl types. These are found in the file **sv.h** in the `svtype` enum. Test these flags with the `SvTYPE` macro.

SVt_IV

Integer type flag for scalars. See `svtype`.

SVt_NV

Double type flag for scalars. See `svtype`.

SVt_PV

Pointer type flag for scalars. See `svtype`.

SVt_PVAV

Type flag for arrays. See `svtype`.

SVt_PVCV

Type flag for code refs. See `svtype`.

SVt_PVHV

Type flag for hashes. See `svtype`.

SVt_PVMG

Type flag for blessed scalars. See `svtype`.

72.22 SV Manipulation Functions

get_sv

Returns the SV of the specified Perl scalar. If `create` is set and the Perl variable does not exist then it will be created. If `create` is not set and the variable does not exist then NULL is returned.

NOTE: the perl_ form of this function is deprecated.

```
        SV*     get_sv(const char* name, I32 create)
```

looks_like_number

Test if the content of an SV looks like a number (or is a number). `Inf` and `Infinity` are treated as numbers (so will not issue a non-numeric warning), even if your atof() doesn't grok them.

```
        I32     looks_like_number(SV* sv)
```

newRV_inc

Creates an RV wrapper for an SV. The reference count for the original SV is incremented.

```
        SV*     newRV_inc(SV* sv)
```

newRV_noinc

Creates an RV wrapper for an SV. The reference count for the original SV is **not** incremented.

```
        SV*     newRV_noinc(SV *sv)
```

NEWSV

Creates a new SV. A non-zero `len` parameter indicates the number of bytes of preallocated string space the SV should have. An extra byte for a tailing NUL is also reserved. (SvPOK is not set for the SV even if string space is allocated.) The reference count for the new SV is set to 1. `id` is an integer id between 0 and 1299 (used to identify leaks).

```
        SV*     NEWSV(int id, STRLEN len)
```

newSV

Create a new null SV, or if len > 0, create a new empty SVt_PV type SV with an initial PV allocation of len+1. Normally accessed via the `NEWSV` macro.

```
        SV*     newSV(STRLEN len)
```

newSViv

Creates a new SV and copies an integer into it. The reference count for the SV is set to 1.

```
        SV*     newSViv(IV i)
```

newSVnv

Creates a new SV and copies a floating point value into it. The reference count for the SV is set to 1.

```
        SV*     newSVnv(NV n)
```

newSVpv

Creates a new SV and copies a string into it. The reference count for the SV is set to 1. If `len` is zero, Perl will compute the length using strlen(). For efficiency, consider using `newSVpvn` instead.

```
        SV*     newSVpv(const char* s, STRLEN len)
```

newSVpvf

Creates a new SV and initializes it with the string formatted like `sprintf`.

```
        SV*     newSVpvf(const char* pat, ...)
```

newSVpvn

Creates a new SV and copies a string into it. The reference count for the SV is set to 1. Note that if `len` is zero, Perl will create a zero length string. You are responsible for ensuring that the source string is at least `len` bytes long. If the `s` argument is NULL the new SV will be undefined.

```
        SV*     newSVpvn(const char* s, STRLEN len)
```

newSVpvn_share

Creates a new SV with its SvPVX pointing to a shared string in the string table. If the string does not already exist in the table, it is created first. Turns on READONLY and FAKE. The string's hash is stored in the UV slot of the SV; if the `hash` parameter is non-zero, that value is used; otherwise the hash is computed. The idea here is that as the string table is used for shared hash keys these strings will have SvPVX == HeKEY and hash lookup will avoid string compare.

```
        SV*     newSVpvn_share(const char* s, I32 len, U32 hash)
```

newSVrv

Creates a new SV for the RV, `rv`, to point to. If `rv` is not an RV then it will be upgraded to one. If `classname` is non-null then the new SV will be blessed in the specified package. The new SV is returned and its reference count is 1.

```
        SV*     newSVrv(SV* rv, const char* classname)
```

newSVsv

Creates a new SV which is an exact duplicate of the original SV. (Uses `sv_setsv`).

```
        SV*     newSVsv(SV* old)
```

newSVuv

Creates a new SV and copies an unsigned integer into it. The reference count for the SV is set to 1.

```
        SV*     newSVuv(UV u)
```

SvCUR

Returns the length of the string which is in the SV. See `SvLEN`.

```
        STRLEN  SvCUR(SV* sv)
```

SvCUR_set

Set the length of the string which is in the SV. See `SvCUR`.

```
        void    SvCUR_set(SV* sv, STRLEN len)
```

SvEND

Returns a pointer to the last character in the string which is in the SV. See `SvCUR`. Access the character as *(SvEND(sv)).

```
        char*   SvEND(SV* sv)
```

SvGROW

Expands the character buffer in the SV so that it has room for the indicated number of bytes (remember to reserve space for an extra trailing NUL character). Calls `sv_grow` to perform the expansion if necessary. Returns a pointer to the character buffer.

```
        char *  SvGROW(SV* sv, STRLEN len)
```

SvIOK

Returns a boolean indicating whether the SV contains an integer.

```
        bool    SvIOK(SV* sv)
```

SvIOKp

Returns a boolean indicating whether the SV contains an integer. Checks the **private** setting. Use `SvIOK`.

```
        bool    SvIOKp(SV* sv)
```

SvIOK_notUV

Returns a boolean indicating whether the SV contains a signed integer.

```
        bool    SvIOK_notUV(SV* sv)
```

SvIOK_off

Unsets the IV status of an SV.

```
        void    SvIOK_off(SV* sv)
```

SvIOK_on

Tells an SV that it is an integer.

```
        void    SvIOK_on(SV* sv)
```

SvIOK_only

Tells an SV that it is an integer and disables all other OK bits.

```
        void    SvIOK_only(SV* sv)
```

SvIOK_only_UV

Tells and SV that it is an unsigned integer and disables all other OK bits.

```
        void    SvIOK_only_UV(SV* sv)
```

SvIOK_UV

Returns a boolean indicating whether the SV contains an unsigned integer.

```
        bool    SvIOK_UV(SV* sv)
```

SvIsCOW

Returns a boolean indicating whether the SV is Copy-On-Write. (either shared hash key scalars, or full Copy On Write scalars if 5.9.0 is configured for COW)

```
        bool    SvIsCOW(SV* sv)
```

SvIsCOW_shared_hash

Returns a boolean indicating whether the SV is Copy-On-Write shared hash key scalar.

```
        bool    SvIsCOW_shared_hash(SV* sv)
```

SvIV

Coerces the given SV to an integer and returns it. See `SvIVx` for a version which guarantees to evaluate sv only once.

```
        IV      SvIV(SV* sv)
```

SvIVx

Coerces the given SV to an integer and returns it. Guarantees to evaluate sv only once. Use the more efficient `SvIV` otherwise.

```
        IV      SvIVx(SV* sv)
```

SvIVX

Returns the raw value in the SV's IV slot, without checks or conversions. Only use when you are sure SvIOK is true. See also `SvIV()`.

```
        IV      SvIVX(SV* sv)
```

SvLEN

Returns the size of the string buffer in the SV, not including any part attributable to `SvOOK`. See `SvCUR`.

```
        STRLEN  SvLEN(SV* sv)
```

SvNIOK

Returns a boolean indicating whether the SV contains a number, integer or double.

```
        bool    SvNIOK(SV* sv)
```

SvNIOKp

Returns a boolean indicating whether the SV contains a number, integer or double. Checks the **private** setting. Use `SvNIOK`.

```
        bool    SvNIOKp(SV* sv)
```

SvNIOK_off

Unsets the NV/IV status of an SV.

```
        void    SvNIOK_off(SV* sv)
```

SvNOK

Returns a boolean indicating whether the SV contains a double.

```
        bool    SvNOK(SV* sv)
```

SvNOKp

Returns a boolean indicating whether the SV contains a double. Checks the **private** setting. Use `SvNOK`.

```
        bool    SvNOKp(SV* sv)
```

SvNOK_off

Unsets the NV status of an SV.

```
        void    SvNOK_off(SV* sv)
```

SvNOK_on

Tells an SV that it is a double.

```
        void    SvNOK_on(SV* sv)
```

SvNOK_only

Tells an SV that it is a double and disables all other OK bits.

```
        void    SvNOK_only(SV* sv)
```

SvNV

Coerce the given SV to a double and return it. See `SvNVx` for a version which guarantees to evaluate sv only once.

```
        NV      SvNV(SV* sv)
```

SvNVx

Coerces the given SV to a double and returns it. Guarantees to evaluate sv only once. Use the more efficient `SvNV` otherwise.

```
        NV      SvNVx(SV* sv)
```

SvNVX

Returns the raw value in the SV's NV slot, without checks or conversions. Only use when you are sure SvNOK is true. See also `SvNV()`.

```
        NV      SvNVX(SV* sv)
```

SvOK

Returns a boolean indicating whether the value is an SV. It also tells whether the value is defined or not.

```
        bool    SvOK(SV* sv)
```

SvOOK

Returns a boolean indicating whether the SvIVX is a valid offset value for the SvPVX. This hack is used internally to speed up removal of characters from the beginning of a SvPV. When SvOOK is true, then the start of the allocated string buffer is really (SvPVX - SvIVX).

```
        bool    SvOOK(SV* sv)
```

SvPOK

Returns a boolean indicating whether the SV contains a character string.

```
bool    SvPOK(SV* sv)
```

SvPOKp

Returns a boolean indicating whether the SV contains a character string. Checks the **private** setting. Use `SvPOK`.

```
bool    SvPOKp(SV* sv)
```

SvPOK_off

Unsets the PV status of an SV.

```
void    SvPOK_off(SV* sv)
```

SvPOK_on

Tells an SV that it is a string.

```
void    SvPOK_on(SV* sv)
```

SvPOK_only

Tells an SV that it is a string and disables all other OK bits. Will also turn off the UTF-8 status.

```
void    SvPOK_only(SV* sv)
```

SvPOK_only_UTF8

Tells an SV that it is a string and disables all other OK bits, and leaves the UTF-8 status as it was.

```
void    SvPOK_only_UTF8(SV* sv)
```

SvPV

Returns a pointer to the string in the SV, or a stringified form of the SV if the SV does not contain a string. The SV may cache the stringified version becoming `SvPOK`. Handles 'get' magic. See also `SvPVx` for a version which guarantees to evaluate sv only once.

```
char*   SvPV(SV* sv, STRLEN len)
```

SvPVbyte

Like `SvPV`, but converts sv to byte representation first if necessary.

```
char*   SvPVbyte(SV* sv, STRLEN len)
```

SvPVbytex

Like `SvPV`, but converts sv to byte representation first if necessary. Guarantees to evaluate sv only once; use the more efficient `SvPVbyte` otherwise.

```
char*   SvPVbytex(SV* sv, STRLEN len)
```

SvPVbytex_force

Like `SvPV_force`, but converts sv to byte representation first if necessary. Guarantees to evaluate sv only once; use the more efficient `SvPVbyte_force` otherwise.

```
char*   SvPVbytex_force(SV* sv, STRLEN len)
```

SvPVbyte_force

Like `SvPV_force`, but converts sv to byte representation first if necessary.

```
        char*   SvPVbyte_force(SV* sv, STRLEN len)
```

SvPVbyte_nolen

Like `SvPV_nolen`, but converts sv to byte representation first if necessary.

```
        char*   SvPVbyte_nolen(SV* sv)
```

SvPVutf8

Like `SvPV`, but converts sv to utf8 first if necessary.

```
        char*   SvPVutf8(SV* sv, STRLEN len)
```

SvPVutf8x

Like `SvPV`, but converts sv to utf8 first if necessary. Guarantees to evaluate sv only once; use the more efficient `SvPVutf8` otherwise.

```
        char*   SvPVutf8x(SV* sv, STRLEN len)
```

SvPVutf8x_force

Like `SvPV_force`, but converts sv to utf8 first if necessary. Guarantees to evaluate sv only once; use the more efficient `SvPVutf8_force` otherwise.

```
        char*   SvPVutf8x_force(SV* sv, STRLEN len)
```

SvPVutf8_force

Like `SvPV_force`, but converts sv to utf8 first if necessary.

```
        char*   SvPVutf8_force(SV* sv, STRLEN len)
```

SvPVutf8_nolen

Like `SvPV_nolen`, but converts sv to utf8 first if necessary.

```
        char*   SvPVutf8_nolen(SV* sv)
```

SvPVX

Returns a pointer to the physical string in the SV. The SV must contain a string.

```
        char*   SvPVX(SV* sv)
```

SvPVx

A version of `SvPV` which guarantees to evaluate sv only once.

```
        char*   SvPVx(SV* sv, STRLEN len)
```

SvPV_force

Like `SvPV` but will force the SV into containing just a string (`SvPOK_only`). You want force if you are going to update the `SvPVX` directly.

```
        char*   SvPV_force(SV* sv, STRLEN len)
```

SvPV_force_nomg

Like `SvPV` but will force the SV into containing just a string (`SvPOK_only`). You want force if you are going to update the `SvPVX` directly. Doesn't process magic.

```
        char*   SvPV_force_nomg(SV* sv, STRLEN len)
```

SvPV_nolen

Returns a pointer to the string in the SV, or a stringified form of the SV if the SV does not contain a string. The SV may cache the stringified form becoming `SvPOK`. Handles 'get' magic.

```
        char*   SvPV_nolen(SV* sv)
```

SvREFCNT

Returns the value of the object's reference count.

```
        U32     SvREFCNT(SV* sv)
```

SvREFCNT_dec

Decrements the reference count of the given SV.

```
        void    SvREFCNT_dec(SV* sv)
```

SvREFCNT_inc

Increments the reference count of the given SV.

```
        SV*     SvREFCNT_inc(SV* sv)
```

SvROK

Tests if the SV is an RV.

```
        bool    SvROK(SV* sv)
```

SvROK_off

Unsets the RV status of an SV.

```
        void    SvROK_off(SV* sv)
```

SvROK_on

Tells an SV that it is an RV.

```
        void    SvROK_on(SV* sv)
```

SvRV

Dereferences an RV to return the SV.

```
        SV*     SvRV(SV* sv)
```

SvSTASH

Returns the stash of the SV.

```
        HV*     SvSTASH(SV* sv)
```

SvTAINT

Taints an SV if tainting is enabled.

```
        void    SvTAINT(SV* sv)
```

SvTAINTED

Checks to see if an SV is tainted. Returns TRUE if it is, FALSE if not.

```
        bool    SvTAINTED(SV* sv)
```

SvTAINTED_off

Untaints an SV. Be *very* careful with this routine, as it short-circuits some of Perl's fundamental security features. XS module authors should not use this function unless they fully understand all the implications of unconditionally untainting the value. Untainting should be done in the standard perl fashion, via a carefully crafted regexp, rather than directly untainting variables.

```
        void    SvTAINTED_off(SV* sv)
```

SvTAINTED_on

Marks an SV as tainted if tainting is enabled.

```
        void    SvTAINTED_on(SV* sv)
```

SvTRUE

Returns a boolean indicating whether Perl would evaluate the SV as true or false, defined or undefined. Does not handle 'get' magic.

```
        bool    SvTRUE(SV* sv)
```

SvTYPE

Returns the type of the SV. See `svtype`.

```
        svtype  SvTYPE(SV* sv)
```

SvUOK

Returns a boolean indicating whether the SV contains an unsigned integer.

```
        void    SvUOK(SV* sv)
```

SvUPGRADE

Used to upgrade an SV to a more complex form. Uses `sv_upgrade` to perform the upgrade if necessary. See `svtype`.

```
        void    SvUPGRADE(SV* sv, svtype type)
```

SvUTF8

Returns a boolean indicating whether the SV contains UTF-8 encoded data.

```
        bool    SvUTF8(SV* sv)
```

SvUTF8_off

Unsets the UTF-8 status of an SV.

```
void    SvUTF8_off(SV *sv)
```

SvUTF8_on

Turn on the UTF-8 status of an SV (the data is not changed, just the flag). Do not use frivolously.

```
void    SvUTF8_on(SV *sv)
```

SvUV

Coerces the given SV to an unsigned integer and returns it. See `SvUVx` for a version which guarantees to evaluate sv only once.

```
UV      SvUV(SV* sv)
```

SvUVx

Coerces the given SV to an unsigned integer and returns it. Guarantees to evaluate sv only once. Use the more efficient `SvUV` otherwise.

```
UV      SvUVx(SV* sv)
```

SvUVX

Returns the raw value in the SV's UV slot, without checks or conversions. Only use when you are sure SvIOK is true. See also `SvUV()`.

```
UV      SvUVX(SV* sv)
```

sv_2bool

This function is only called on magical items, and is only used by sv_true() or its macro equivalent.

```
bool    sv_2bool(SV* sv)
```

sv_2cv

Using various gambits, try to get a CV from an SV; in addition, try if possible to set `*st` and `*gvp` to the stash and GV associated with it.

```
CV*     sv_2cv(SV* sv, HV** st, GV** gvp, I32 lref)
```

sv_2io

Using various gambits, try to get an IO from an SV: the IO slot if its a GV; or the recursive result if we're an RV; or the IO slot of the symbol named after the PV if we're a string.

```
IO*     sv_2io(SV* sv)
```

sv_2iv

Return the integer value of an SV, doing any necessary string conversion, magic etc. Normally used via the `SvIV(sv)` and `SvIVx(sv)` macros.

```
IV      sv_2iv(SV* sv)
```

sv_2mortal

Marks an existing SV as mortal. The SV will be destroyed "soon", either by an explicit call to FREETMPS, or by an implicit call at places such as statement boundaries. See also `sv_newmortal` and `sv_mortalcopy`.

```
        SV*     sv_2mortal(SV* sv)
```

sv_2nv

Return the num value of an SV, doing any necessary string or integer conversion, magic etc. Normally used via the `SvNV(sv)` and `SvNVx(sv)` macros.

```
        NV      sv_2nv(SV* sv)
```

sv_2pvbyte

Return a pointer to the byte-encoded representation of the SV, and set *lp to its length. May cause the SV to be downgraded from UTF-8 as a side-effect.

Usually accessed via the `SvPVbyte` macro.

```
        char*   sv_2pvbyte(SV* sv, STRLEN* lp)
```

sv_2pvbyte_nolen

Return a pointer to the byte-encoded representation of the SV. May cause the SV to be downgraded from UTF-8 as a side-effect.

Usually accessed via the `SvPVbyte_nolen` macro.

```
        char*   sv_2pvbyte_nolen(SV* sv)
```

sv_2pvutf8

Return a pointer to the UTF-8-encoded representation of the SV, and set *lp to its length. May cause the SV to be upgraded to UTF-8 as a side-effect.

Usually accessed via the `SvPVutf8` macro.

```
        char*   sv_2pvutf8(SV* sv, STRLEN* lp)
```

sv_2pvutf8_nolen

Return a pointer to the UTF-8-encoded representation of the SV. May cause the SV to be upgraded to UTF-8 as a side-effect.

Usually accessed via the `SvPVutf8_nolen` macro.

```
        char*   sv_2pvutf8_nolen(SV* sv)
```

sv_2pv_flags

Returns a pointer to the string value of an SV, and sets *lp to its length. If flags includes SV_GMAGIC, does an mg_get() first. Coerces sv to a string if necessary. Normally invoked via the `SvPV_flags` macro. `sv_2pv()` and `sv_2pv_nomg` usually end up here too.

```
        char*   sv_2pv_flags(SV* sv, STRLEN* lp, I32 flags)
```

sv_2pv_nolen

Like `sv_2pv()`, but doesn't return the length too. You should usually use the macro wrapper `SvPV_nolen(sv)` instead. char* sv_2pv_nolen(SV* sv)

sv_2uv

Return the unsigned integer value of an SV, doing any necessary string conversion, magic etc. Normally used via the `SvUV(sv)` and `SvUVx(sv)` macros.

```
        UV      sv_2uv(SV* sv)
```

sv_backoff

Remove any string offset. You should normally use the `SvOOK_off` macro wrapper instead.

```
        int     sv_backoff(SV* sv)
```

sv_bless

Blesses an SV into a specified package. The SV must be an RV. The package must be designated by its stash (see `gv_stashpv()`). The reference count of the SV is unaffected.

```
        SV*     sv_bless(SV* sv, HV* stash)
```

sv_catpv

Concatenates the string onto the end of the string which is in the SV. If the SV has the UTF-8 status set, then the bytes appended should be valid UTF-8. Handles 'get' magic, but not 'set' magic. See `sv_catpv_mg`.

```
        void    sv_catpv(SV* sv, const char* ptr)
```

sv_catpvf

Processes its arguments like `sprintf` and appends the formatted output to an SV. If the appended data contains "wide" characters (including, but not limited to, SVs with a UTF-8 PV formatted with %s, and characters >255 formatted with %c), the original SV might get upgraded to UTF-8. Handles 'get' magic, but not 'set' magic. `SvSETMAGIC()` must typically be called after calling this function to handle 'set' magic.

```
        void    sv_catpvf(SV* sv, const char* pat, ...)
```

sv_catpvf_mg

Like `sv_catpvf`, but also handles 'set' magic.

```
        void    sv_catpvf_mg(SV *sv, const char* pat, ...)
```

sv_catpvn

Concatenates the string onto the end of the string which is in the SV. The `len` indicates number of bytes to copy. If the SV has the UTF-8 status set, then the bytes appended should be valid UTF-8. Handles 'get' magic, but not 'set' magic. See `sv_catpvn_mg`.

```
        void    sv_catpvn(SV* sv, const char* ptr, STRLEN len)
```

sv_catpvn_flags

Concatenates the string onto the end of the string which is in the SV. The `len` indicates number of bytes to copy. If the SV has the UTF-8 status set, then the bytes appended should be valid UTF-8. If `flags` has `SV_GMAGIC` bit set, will `mg_get` on `dsv` if appropriate, else not. `sv_catpvn` and `sv_catpvn_nomg` are implemented in terms of this function.

```
        void    sv_catpvn_flags(SV* sv, const char* ptr, STRLEN len, I32 flags)
```

sv_catpvn_mg

Like `sv_catpvn`, but also handles 'set' magic.

```
        void    sv_catpvn_mg(SV *sv, const char *ptr, STRLEN len)
```

sv_catpv_mg

Like `sv_catpv`, but also handles 'set' magic.

```
        void    sv_catpv_mg(SV *sv, const char *ptr)
```

sv_catsv

Concatenates the string from SV `ssv` onto the end of the string in SV `dsv`. Modifies `dsv` but not `ssv`. Handles 'get' magic, but not 'set' magic. See `sv_catsv_mg`.

```
        void    sv_catsv(SV* dsv, SV* ssv)
```

sv_catsv_flags

Concatenates the string from SV `ssv` onto the end of the string in SV `dsv`. Modifies `dsv` but not `ssv`. If `flags` has `SV_GMAGIC` bit set, will `mg_get` on the SVs if appropriate, else not. `sv_catsv` and `sv_catsv_nomg` are implemented in terms of this function.

```
        void    sv_catsv_flags(SV* dsv, SV* ssv, I32 flags)
```

sv_catsv_mg

Like `sv_catsv`, but also handles 'set' magic.

```
        void    sv_catsv_mg(SV *dstr, SV *sstr)
```

sv_chop

Efficient removal of characters from the beginning of the string buffer. SvPOK(sv) must be true and the `ptr` must be a pointer to somewhere inside the string buffer. The `ptr` becomes the first character of the adjusted string. Uses the "OOK hack". Beware: after this function returns, `ptr` and SvPVX(sv) may no longer refer to the same chunk of data.

```
        void    sv_chop(SV* sv, char* ptr)
```

sv_clear

Clear an SV: call any destructors, free up any memory used by the body, and free the body itself. The SV's head is *not* freed, although its type is set to all 1's so that it won't inadvertently be assumed to be live during global destruction etc. This function should only be called when REFCNT is zero. Most of the time you'll want to call `sv_free()` (or its macro wrapper `SvREFCNT_dec`) instead.

```
        void    sv_clear(SV* sv)
```

sv_cmp

Compares the strings in two SVs. Returns -1, 0, or 1 indicating whether the string in `sv1` is less than, equal to, or greater than the string in `sv2`. Is UTF-8 and 'use bytes' aware, handles get magic, and will coerce its args to strings if necessary. See also `sv_cmp_locale`.

```
        I32     sv_cmp(SV* sv1, SV* sv2)
```

sv_cmp_locale

Compares the strings in two SVs in a locale-aware manner. Is UTF-8 and 'use bytes' aware, handles get magic, and will coerce its args to strings if necessary. See also `sv_cmp_locale`. See also `sv_cmp`.

```
        I32     sv_cmp_locale(SV* sv1, SV* sv2)
```

sv_collxfrm

Add Collate Transform magic to an SV if it doesn't already have it.

Any scalar variable may carry PERL_MAGIC_collxfrm magic that contains the scalar data of the variable, but transformed to such a format that a normal memory comparison can be used to compare the data according to the locale settings.

```
        char*   sv_collxfrm(SV* sv, STRLEN* nxp)
```

sv_copypv

Copies a stringified representation of the source SV into the destination SV. Automatically performs any necessary mg_get and coercion of numeric values into strings. Guaranteed to preserve UTF-8 flag even from overloaded objects. Similar in nature to sv_2pv[_flags] but operates directly on an SV instead of just the string. Mostly uses sv_2pv_flags to do its work, except when that would lose the UTF-8'ness of the PV.

```
        void    sv_copypv(SV* dsv, SV* ssv)
```

sv_dec

Auto-decrement of the value in the SV, doing string to numeric conversion if necessary. Handles 'get' magic.

```
        void    sv_dec(SV* sv)
```

sv_derived_from

Returns a boolean indicating whether the SV is derived from the specified class. This is the function that implements `UNIVERSAL::isa`. It works for class names as well as for objects.

```
        bool    sv_derived_from(SV* sv, const char* name)
```

sv_eq

Returns a boolean indicating whether the strings in the two SVs are identical. Is UTF-8 and 'use bytes' aware, handles get magic, and will coerce its args to strings if necessary.

```
        I32     sv_eq(SV* sv1, SV* sv2)
```

sv_force_normal

Undo various types of fakery on an SV: if the PV is a shared string, make a private copy; if we're a ref, stop refing; if we're a glob, downgrade to an xpvmg. See also `sv_force_normal_flags`.

```
        void    sv_force_normal(SV *sv)
```

sv_force_normal_flags

Undo various types of fakery on an SV: if the PV is a shared string, make a private copy; if we're a ref, stop refing; if we're a glob, downgrade to an xpvmg. The `flags` parameter gets passed to `sv_unref_flags()` when unrefing. `sv_force_normal` calls this function with flags set to 0.

```
        void    sv_force_normal_flags(SV *sv, U32 flags)
```

sv_free

Decrement an SV's reference count, and if it drops to zero, call `sv_clear` to invoke destructors and free up any memory used by the body; finally, deallocate the SV's head itself. Normally called via a wrapper macro `SvREFCNT_dec`.

```
void    sv_free(SV* sv)
```

sv_gets

Get a line from the filehandle and store it into the SV, optionally appending to the currently-stored string.

```
char*   sv_gets(SV* sv, PerlIO* fp, I32 append)
```

sv_grow

Expands the character buffer in the SV. If necessary, uses `sv_unref` and upgrades the SV to `SVt_PV`. Returns a pointer to the character buffer. Use the `SvGROW` wrapper instead.

```
char*   sv_grow(SV* sv, STRLEN newlen)
```

sv_inc

Auto-increment of the value in the SV, doing string to numeric conversion if necessary. Handles 'get' magic.

```
void    sv_inc(SV* sv)
```

sv_insert

Inserts a string at the specified offset/length within the SV. Similar to the Perl substr() function.

```
void    sv_insert(SV* bigsv, STRLEN offset, STRLEN len, char* little, STRLEN littlelen)
```

sv_isa

Returns a boolean indicating whether the SV is blessed into the specified class. This does not check for subtypes; use `sv_derived_from` to verify an inheritance relationship.

```
int     sv_isa(SV* sv, const char* name)
```

sv_isobject

Returns a boolean indicating whether the SV is an RV pointing to a blessed object. If the SV is not an RV, or if the object is not blessed, then this will return false.

```
int     sv_isobject(SV* sv)
```

sv_iv

A private implementation of the `SvIVx` macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

```
IV      sv_iv(SV* sv)
```

sv_len

Returns the length of the string in the SV. Handles magic and type coercion. See also `SvCUR`, which gives raw access to the xpv_cur slot.

```
STRLEN  sv_len(SV* sv)
```

sv_len_utf8

Returns the number of characters in the string in an SV, counting wide UTF-8 bytes as a single character. Handles magic and type coercion.

```
        STRLEN  sv_len_utf8(SV* sv)
```

sv_magic

Adds magic to an SV. First upgrades `sv` to type `SVt_PVMG` if necessary, then adds a new magic item of type `how` to the head of the magic list.

```
        void    sv_magic(SV* sv, SV* obj, int how, const char* name, I32 namlen)
```

sv_magicext

Adds magic to an SV, upgrading it if necessary. Applies the supplied vtable and returns pointer to the magic added.

Note that sv_magicext will allow things that sv_magic will not. In particular you can add magic to SvREADONLY SVs and and more than one instance of the same 'how'

I `namelen` is greater then zero then a savepvn() *copy* of `name` is stored, if `namelen` is zero then `name` is stored as-is and - as another special case - if `(name && namelen == HEf_SVKEY)` then `name` is assumed to contain an `SV*` and has its REFCNT incremented

(This is now used as a subroutine by sv_magic.)

```
        MAGIC * sv_magicext(SV* sv, SV* obj, int how, MGVTBL *vtbl, const char* name, I32 namlen
```

sv_mortalcopy

Creates a new SV which is a copy of the original SV (using `sv_setsv`). The new SV is marked as mortal. It will be destroyed "soon", either by an explicit call to FREETMPS, or by an implicit call at places such as statement boundaries. See also `sv_newmortal` and `sv_2mortal`.

```
        SV*     sv_mortalcopy(SV* oldsv)
```

sv_newmortal

Creates a new null SV which is mortal. The reference count of the SV is set to 1. It will be destroyed "soon", either by an explicit call to FREETMPS, or by an implicit call at places such as statement boundaries. See also `sv_mortalcopy` and `sv_2mortal`.

```
        SV*     sv_newmortal()
```

sv_newref

Increment an SV's reference count. Use the `SvREFCNT_inc()` wrapper instead.

```
        SV*     sv_newref(SV* sv)
```

sv_nv

A private implementation of the `SvNVx` macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

```
        NV      sv_nv(SV* sv)
```

sv_pos_b2u

Converts the value pointed to by offsetp from a count of bytes from the start of the string, to a count of the equivalent number of UTF-8 chars. Handles magic and type coercion.

```
        void    sv_pos_b2u(SV* sv, I32* offsetp)
```

sv_pos_u2b

Converts the value pointed to by offsetp from a count of UTF-8 chars from the start of the string, to a count of the equivalent number of bytes; if lenp is non-zero, it does the same to lenp, but this time starting from the offset, rather than from the start of the string. Handles magic and type coercion.

```
        void    sv_pos_u2b(SV* sv, I32* offsetp, I32* lenp)
```

sv_pv

Use the SvPV_nolen macro instead

```
        char*   sv_pv(SV *sv)
```

sv_pvbyte

Use SvPVbyte_nolen instead.

```
        char*   sv_pvbyte(SV *sv)
```

sv_pvbyten

A private implementation of the SvPVbyte macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

```
        char*   sv_pvbyten(SV *sv, STRLEN *len)
```

sv_pvbyten_force

A private implementation of the SvPVbytex_force macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

```
        char*   sv_pvbyten_force(SV* sv, STRLEN* lp)
```

sv_pvn

A private implementation of the SvPV macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

```
        char*   sv_pvn(SV *sv, STRLEN *len)
```

sv_pvn_force

Get a sensible string out of the SV somehow. A private implementation of the SvPV_force macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

```
        char*   sv_pvn_force(SV* sv, STRLEN* lp)
```

sv_pvn_force_flags

Get a sensible string out of the SV somehow. If flags has SV_GMAGIC bit set, will mg_get on sv if appropriate, else not. sv_pvn_force and sv_pvn_force_nomg are implemented in terms of this function. You normally want to use the various wrapper macros instead: see SvPV_force and SvPV_force_nomg

```
        char*   sv_pvn_force_flags(SV* sv, STRLEN* lp, I32 flags)
```

sv_pvutf8

Use the SvPVutf8_nolen macro instead

```
char*   sv_pvutf8(SV *sv)
```

sv_pvutf8n

A private implementation of the `SvPVutf8` macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

```
char*   sv_pvutf8n(SV *sv, STRLEN *len)
```

sv_pvutf8n_force

A private implementation of the `SvPVutf8_force` macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

```
char*   sv_pvutf8n_force(SV* sv, STRLEN* lp)
```

sv_reftype

Returns a string describing what the SV is a reference to.

```
char*   sv_reftype(SV* sv, int ob)
```

sv_replace

Make the first argument a copy of the second, then delete the original. The target SV physically takes over ownership of the body of the source SV and inherits its flags; however, the target keeps any magic it owns, and any magic in the source is discarded. Note that this is a rather specialist SV copying operation; most of the time you'll want to use `sv_setsv` or one of its many macro front-ends.

```
void    sv_replace(SV* sv, SV* nsv)
```

sv_report_used

Dump the contents of all SVs not yet freed. (Debugging aid).

```
void    sv_report_used()
```

sv_reset

Underlying implementation for the `reset` Perl function. Note that the perl-level function is vaguely deprecated.

```
void    sv_reset(char* s, HV* stash)
```

sv_rvweaken

Weaken a reference: set the `SvWEAKREF` flag on this RV; give the referred-to SV `PERL_MAGIC_backref` magic if it hasn't already; and push a back-reference to this RV onto the array of backreferences associated with that magic.

```
SV*     sv_rvweaken(SV *sv)
```

sv_setiv

Copies an integer into the given SV, upgrading first if necessary. Does not handle 'set' magic. See also `sv_setiv_mg`.

```
void    sv_setiv(SV* sv, IV num)
```

sv_setiv_mg

Like `sv_setiv`, but also handles 'set' magic.

```
        void    sv_setiv_mg(SV *sv, IV i)
```

sv_setnv

Copies a double into the given SV, upgrading first if necessary. Does not handle 'set' magic. See also `sv_setnv_mg`.

```
        void    sv_setnv(SV* sv, NV num)
```

sv_setnv_mg

Like `sv_setnv`, but also handles 'set' magic.

```
        void    sv_setnv_mg(SV *sv, NV num)
```

sv_setpv

Copies a string into an SV. The string must be null-terminated. Does not handle 'set' magic. See `sv_setpv_mg`.

```
        void    sv_setpv(SV* sv, const char* ptr)
```

sv_setpvf

Processes its arguments like `sprintf` and sets an SV to the formatted output. Does not handle 'set' magic. See `sv_setpvf_mg`.

```
        void    sv_setpvf(SV* sv, const char* pat, ...)
```

sv_setpvf_mg

Like `sv_setpvf`, but also handles 'set' magic.

```
        void    sv_setpvf_mg(SV *sv, const char* pat, ...)
```

sv_setpviv

Copies an integer into the given SV, also updating its string value. Does not handle 'set' magic. See `sv_setpviv_mg`.

```
        void    sv_setpviv(SV* sv, IV num)
```

sv_setpviv_mg

Like `sv_setpviv`, but also handles 'set' magic.

```
        void    sv_setpviv_mg(SV *sv, IV iv)
```

sv_setpvn

Copies a string into an SV. The `len` parameter indicates the number of bytes to be copied. If the `ptr` argument is NULL the SV will become undefined. Does not handle 'set' magic. See `sv_setpvn_mg`.

```
        void    sv_setpvn(SV* sv, const char* ptr, STRLEN len)
```

sv_setpvn_mg

Like `sv_setpvn`, but also handles 'set' magic.

```
        void    sv_setpvn_mg(SV *sv, const char *ptr, STRLEN len)
```

sv_setpv_mg

Like `sv_setpv`, but also handles 'set' magic.

```
        void    sv_setpv_mg(SV *sv, const char *ptr)
```

sv_setref_iv

Copies an integer into a new SV, optionally blessing the SV. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will have a reference count of 1, and the RV will be returned.

```
        SV*     sv_setref_iv(SV* rv, const char* classname, IV iv)
```

sv_setref_nv

Copies a double into a new SV, optionally blessing the SV. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will have a reference count of 1, and the RV will be returned.

```
        SV*     sv_setref_nv(SV* rv, const char* classname, NV nv)
```

sv_setref_pv

Copies a pointer into a new SV, optionally blessing the SV. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. If the `pv` argument is NULL then `PL_sv_undef` will be placed into the SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will have a reference count of 1, and the RV will be returned.

Do not use with other Perl types such as HV, AV, SV, CV, because those objects will become corrupted by the pointer copy process.

Note that `sv_setref_pvn` copies the string while this copies the pointer.

```
        SV*     sv_setref_pv(SV* rv, const char* classname, void* pv)
```

sv_setref_pvn

Copies a string into a new SV, optionally blessing the SV. The length of the string must be specified with `n`. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will have a reference count of 1, and the RV will be returned.

Note that `sv_setref_pv` copies the pointer while this copies the string.

```
        SV*     sv_setref_pvn(SV* rv, const char* classname, char* pv, STRLEN n)
```

sv_setref_uv

Copies an unsigned integer into a new SV, optionally blessing the SV. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will have a reference count of 1, and the RV will be returned.

```
        SV*     sv_setref_uv(SV* rv, const char* classname, UV uv)
```

sv_setsv

Copies the contents of the source SV `ssv` into the destination SV `dsv`. The source SV may be destroyed if it is mortal, so don't use this function if the source SV needs to be reused. Does not handle 'set' magic. Loosely speaking, it performs a copy-by-value, obliterating any previous content of the destination.

You probably want to use one of the assortment of wrappers, such as `SvSetSV`, `SvSetSV_nosteal`, `SvSetMagicSV` and `SvSetMagicSV_nosteal`.

```
        void    sv_setsv(SV* dsv, SV* ssv)
```

sv_setsv_flags

Copies the contents of the source SV `ssv` into the destination SV `dsv`. The source SV may be destroyed if it is mortal, so don't use this function if the source SV needs to be reused. Does not handle 'set' magic. Loosely speaking, it performs a copy-by-value, obliterating any previous content of the destination. If the `flags` parameter has the `SV_GMAGIC` bit set, will `mg_get` on `ssv` if appropriate, else not. `sv_setsv` and `sv_setsv_nomg` are implemented in terms of this function.

You probably want to use one of the assortment of wrappers, such as `SvSetSV`, `SvSetSV_nosteal`, `SvSetMagicSV` and `SvSetMagicSV_nosteal`.

This is the primary function for copying scalars, and most other copy-ish functions and macros use this underneath.

```
        void    sv_setsv_flags(SV* dsv, SV* ssv, I32 flags)
```

sv_setsv_mg

Like `sv_setsv`, but also handles 'set' magic.

```
        void    sv_setsv_mg(SV *dstr, SV *sstr)
```

sv_setuv

Copies an unsigned integer into the given SV, upgrading first if necessary. Does not handle 'set' magic. See also `sv_setuv_mg`.

```
        void    sv_setuv(SV* sv, UV num)
```

sv_setuv_mg

Like `sv_setuv`, but also handles 'set' magic.

```
        void    sv_setuv_mg(SV *sv, UV u)
```

sv_taint

Taint an SV. Use `SvTAINTED_on` instead. void sv_taint(SV* sv)

sv_tainted

Test an SV for taintedness. Use `SvTAINTED` instead. bool sv_tainted(SV* sv)

sv_true

Returns true if the SV has a true value by Perl's rules. Use the `SvTRUE` macro instead, which may call `sv_true()` or may instead use an in-line version.

```
        I32     sv_true(SV *sv)
```

sv_unmagic

Removes all magic of type `type` from an SV.

```
        int     sv_unmagic(SV* sv, int type)
```

sv_unref

Unsets the RV status of the SV, and decrements the reference count of whatever was being referenced by the RV. This can almost be thought of as a reversal of `newSVrv`. This is `sv_unref_flags` with the `flag` being zero. See `SvROK_off`.

```
        void    sv_unref(SV* sv)
```

sv_unref_flags

Unsets the RV status of the SV, and decrements the reference count of whatever was being referenced by the RV. This can almost be thought of as a reversal of `newSVrv`. The `cflags` argument can contain `SV_IMMEDIATE_UNREF` to force the reference count to be decremented (otherwise the decrementing is conditional on the reference count being different from one or the reference being a readonly SV). See `SvROK_off`.

```
        void    sv_unref_flags(SV* sv, U32 flags)
```

sv_untaint

Untaint an SV. Use `SvTAINTED_off` instead. void sv_untaint(SV* sv)

sv_upgrade

Upgrade an SV to a more complex form. Generally adds a new body type to the SV, then copies across as much information as possible from the old body. You generally want to use the `SvUPGRADE` macro wrapper. See also `svtype`.

```
        bool    sv_upgrade(SV* sv, U32 mt)
```

sv_usepvn

Tells an SV to use `ptr` to find its string value. Normally the string is stored inside the SV but sv_usepvn allows the SV to use an outside string. The `ptr` should point to memory that was allocated by `malloc`. The string length, `len`, must be supplied. This function will realloc the memory pointed to by `ptr`, so that pointer should not be freed or used by the programmer after giving it to sv_usepvn. Does not handle 'set' magic. See `sv_usepvn_mg`.

```
        void    sv_usepvn(SV* sv, char* ptr, STRLEN len)
```

sv_usepvn_mg

Like `sv_usepvn`, but also handles 'set' magic.

```
        void    sv_usepvn_mg(SV *sv, char *ptr, STRLEN len)
```

sv_utf8_decode

If the PV of the SV is an octet sequence in UTF-8 and contains a multiple-byte character, the `SvUTF8` flag is turned on so that it looks like a character. If the PV contains only single-byte characters, the `SvUTF8` flag stays being off. Scans PV for validity and returns false if the PV is invalid UTF-8.

NOTE: this function is experimental and may change or be removed without notice.

```
        bool    sv_utf8_decode(SV *sv)
```

sv_utf8_downgrade

Attempts to convert the PV of an SV from characters to bytes. If the PV contains a character beyond byte, this conversion will fail; in this case, either returns false or, if `fail_ok` is not true, croaks.

This is not as a general purpose Unicode to byte encoding interface: use the Encode extension for that.

NOTE: this function is experimental and may change or be removed without notice.

```
        bool    sv_utf8_downgrade(SV *sv, bool fail_ok)
```

sv_utf8_encode

Converts the PV of an SV to UTF-8, but then turns the SvUTF8 flag off so that it looks like octets again.

```
        void    sv_utf8_encode(SV *sv)
```

sv_utf8_upgrade

Converts the PV of an SV to its UTF-8-encoded form. Forces the SV to string form if it is not already. Always sets the SvUTF8 flag to avoid future validity checks even if all the bytes have hibit clear.

This is not as a general purpose byte encoding to Unicode interface: use the Encode extension for that.

```
        STRLEN  sv_utf8_upgrade(SV *sv)
```

sv_utf8_upgrade_flags

Converts the PV of an SV to its UTF-8-encoded form. Forces the SV to string form if it is not already. Always sets the SvUTF8 flag to avoid future validity checks even if all the bytes have hibit clear. If `flags` has `SV_GMAGIC` bit set, will `mg_get` on `sv` if appropriate, else not. `sv_utf8_upgrade` and `sv_utf8_upgrade_nomg` are implemented in terms of this function.

This is not as a general purpose byte encoding to Unicode interface: use the Encode extension for that.

```
        STRLEN  sv_utf8_upgrade_flags(SV *sv, I32 flags)
```

sv_uv

A private implementation of the `SvUVx` macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

```
        UV      sv_uv(SV* sv)
```

sv_vcatpvfn

Processes its arguments like `vsprintf` and appends the formatted output to an SV. Uses an array of SVs if the C style variable argument list is missing (NULL). When running with taint checks enabled, indicates via `maybe_tainted` if results are untrustworthy (often due to the use of locales).

Usually used via one of its frontends `sv_catpvf` and `sv_catpvf_mg`.

```
        void    sv_vcatpvfn(SV* sv, const char* pat, STRLEN patlen, va_list* args, SV** svargs, I3
```

sv_vsetpvfn

Works like `vcatpvfn` but copies the text into the SV instead of appending it.

Usually used via one of its frontends `sv_setpvf` and `sv_setpvf_mg`.

```
        void    sv_vsetpvfn(SV* sv, const char* pat, STRLEN patlen, va_list* args, SV** svargs, I3
```

72.23 Unicode Support

bytes_from_utf8

Converts a string `s` of length `len` from UTF-8 into byte encoding. Unlike <utf8_to_bytes> but like `bytes_to_utf8`, returns a pointer to the newly-created string, and updates `len` to contain the new length. Returns the original string if no conversion occurs, `len` is unchanged. Do nothing if `is_utf8` points to 0. Sets `is_utf8` to 0 if `s` is converted or contains all 7bit characters.

NOTE: this function is experimental and may change or be removed without notice.

```
U8*     bytes_from_utf8(U8 *s, STRLEN *len, bool *is_utf8)
```

bytes_to_utf8

Converts a string `s` of length `len` from ASCII into UTF-8 encoding. Returns a pointer to the newly-created string, and sets `len` to reflect the new length.

If you want to convert to UTF-8 from other encodings than ASCII, see sv_recode_to_utf8().

NOTE: this function is experimental and may change or be removed without notice.

```
U8*     bytes_to_utf8(U8 *s, STRLEN *len)
```

ibcmp_utf8

Return true if the strings s1 and s2 differ case-insensitively, false if not (if they are equal case-insensitively). If u1 is true, the string s1 is assumed to be in UTF-8-encoded Unicode. If u2 is true, the string s2 is assumed to be in UTF-8-encoded Unicode. If u1 or u2 are false, the respective string is assumed to be in native 8-bit encoding.

If the pe1 and pe2 are non-NULL, the scanning pointers will be copied in there (they will point at the beginning of the *next* character). If the pointers behind pe1 or pe2 are non-NULL, they are the end pointers beyond which scanning will not continue under any circustances. If the byte lengths l1 and l2 are non-zero, s1+l1 and s2+l2 will be used as goal end pointers that will also stop the scan, and which qualify towards defining a successful match: all the scans that define an explicit length must reach their goal pointers for a match to succeed).

For case-insensitiveness, the "casefolding" of Unicode is used instead of upper/lowercasing both the characters, see http://www.unicode.org/unicode/reports/tr21/ (Case Mappings).

```
I32     ibcmp_utf8(const char* a, char **pe1, UV l1, bool u1, const char* b, char **pe2, U
```

is_utf8_char

Tests if some arbitrary number of bytes begins in a valid UTF-8 character. Note that an INVARIANT (i.e. ASCII) character is a valid UTF-8 character. The actual number of bytes in the UTF-8 character will be returned if it is valid, otherwise 0.

```
STRLEN  is_utf8_char(U8 *p)
```

is_utf8_string

Returns true if first `len` bytes of the given string form a valid UTF-8 string, false otherwise. Note that 'a valid UTF-8 string' does not mean 'a string that contains code points above 0x7F encoded in UTF-8' because a valid ASCII string is a valid UTF-8 string.

```
bool    is_utf8_string(U8 *s, STRLEN len)
```

is_utf8_string_loc

Like is_ut8_string but store the location of the failure in the last argument.

```
bool    is_utf8_string_loc(U8 *s, STRLEN len, U8 **p)
```

pv_uni_display

Build to the scalar dsv a displayable version of the string spv, length len, the displayable version being at most pvlim bytes long (if longer, the rest is truncated and "..." will be appended).

The flags argument can have UNI_DISPLAY_ISPRINT set to display isPRINT()able characters as themselves, UNI_DISPLAY_BACKSLASH to display the \\[nrfta\\] as the backslashed versions (like '\n') (UNI_DISPLAY_BACKSLASH is preferred over UNI_DISPLAY_ISPRINT for \\). UNI_DISPLAY_QQ (and its alias UNI_DISPLAY_REGEX) have both UNI_DISPLAY_BACKSLASH and UNI_DISPLAY_ISPRINT turned on.

The pointer to the PV of the dsv is returned.

```
        char*   pv_uni_display(SV *dsv, U8 *spv, STRLEN len, STRLEN pvlim, UV flags)
```

sv_cat_decode

The encoding is assumed to be an Encode object, the PV of the ssv is assumed to be octets in that encoding and decoding the input starts from the position which (PV + *offset) pointed to. The dsv will be concatenated the decoded UTF-8 string from ssv. Decoding will terminate when the string tstr appears in decoding output or the input ends on the PV of the ssv. The value which the offset points will be modified to the last input position on the ssv.

Returns TRUE if the terminator was found, else returns FALSE.

```
        bool    sv_cat_decode(SV* dsv, SV *encoding, SV *ssv, int *offset, char* tstr, int tlen)
```

sv_recode_to_utf8

The encoding is assumed to be an Encode object, on entry the PV of the sv is assumed to be octets in that encoding, and the sv will be converted into Unicode (and UTF-8).

If the sv already is UTF-8 (or if it is not POK), or if the encoding is not a reference, nothing is done to the sv. If the encoding is not an `Encode::XS` Encoding object, bad things will happen. (See *lib/encoding.pm* and *Encode*).

The PV of the sv is returned.

```
        char*   sv_recode_to_utf8(SV* sv, SV *encoding)
```

sv_uni_display

Build to the scalar dsv a displayable version of the scalar sv, the displayable version being at most pvlim bytes long (if longer, the rest is truncated and "..." will be appended).

The flags argument is as in pv_uni_display().

The pointer to the PV of the dsv is returned.

```
        char*   sv_uni_display(SV *dsv, SV *ssv, STRLEN pvlim, UV flags)
```

to_utf8_case

The "p" contains the pointer to the UTF-8 string encoding the character that is being converted.

The "ustrp" is a pointer to the character buffer to put the conversion result to. The "lenp" is a pointer to the length of the result.

The "swashp" is a pointer to the swash to use.

Both the special and normal mappings are stored lib/unicore/To/Foo.pl, and loaded by SWASHGET, using lib/utf8_heavy.pl. The special (usually, but not always, a multicharacter mapping), is tried first.

The "special" is a string like "utf8::ToSpecLower", which means the hash %utf8::ToSpecLower. The access to the hash is through Perl_to_utf8_case().

The "normal" is a string like "ToLower" which means the swash %utf8::ToLower.

```
        UV      to_utf8_case(U8 *p, U8* ustrp, STRLEN *lenp, SV **swash, char *normal, char *speci
```

to_utf8_fold

Convert the UTF-8 encoded character at p to its foldcase version and store that in UTF-8 in ustrp and its length in bytes in lenp. Note that the ustrp needs to be at least UTF8_MAXLEN_FOLD+1 bytes since the foldcase version may be longer than the original character (up to three characters).

The first character of the foldcased version is returned (but note, as explained above, that there may be more.)

```
        UV      to_utf8_fold(U8 *p, U8* ustrp, STRLEN *lenp)
```

to_utf8_lower

Convert the UTF-8 encoded character at p to its lowercase version and store that in UTF-8 in ustrp and its length in bytes in lenp. Note that the ustrp needs to be at least UTF8_MAXLEN_UCLC+1 bytes since the lowercase version may be longer than the original character (up to two characters).

The first character of the lowercased version is returned (but note, as explained above, that there may be more.)

```
        UV      to_utf8_lower(U8 *p, U8* ustrp, STRLEN *lenp)
```

to_utf8_title

Convert the UTF-8 encoded character at p to its titlecase version and store that in UTF-8 in ustrp and its length in bytes in lenp. Note that the ustrp needs to be at least UTF8_MAXLEN_UCLC+1 bytes since the titlecase version may be longer than the original character (up to two characters).

The first character of the titlecased version is returned (but note, as explained above, that there may be more.)

```
        UV      to_utf8_title(U8 *p, U8* ustrp, STRLEN *lenp)
```

to_utf8_upper

Convert the UTF-8 encoded character at p to its uppercase version and store that in UTF-8 in ustrp and its length in bytes in lenp. Note that the ustrp needs to be at least UTF8_MAXLEN_UCLC+1 bytes since the uppercase version may be longer than the original character (up to two characters).

The first character of the uppercased version is returned (but note, as explained above, that there may be more.)

```
        UV      to_utf8_upper(U8 *p, U8* ustrp, STRLEN *lenp)
```

utf8n_to_uvchr

Returns the native character value of the first character in the string `s` which is assumed to be in UTF-8 encoding; `retlen` will be set to the length, in bytes, of that character.

Allows length and flags to be passed to low level routine.

```
        UV      utf8n_to_uvchr(U8 *s, STRLEN curlen, STRLEN* retlen, U32 flags)
```

utf8n_to_uvuni

Bottom level UTF-8 decode routine. Returns the unicode code point value of the first character in the string `s` which is assumed to be in UTF-8 encoding and no longer than `curlen`; `retlen` will be set to the length, in bytes, of that character.

If `s` does not point to a well-formed UTF-8 character, the behaviour is dependent on the value of `flags`: if it contains UTF8_CHECK_ONLY, it is assumed that the caller will raise a warning, and this function will silently just set `retlen` to `-1` and return zero. If the `flags` does not contain UTF8_CHECK_ONLY, warnings about malformations will be given, `retlen` will be set to the expected length of the UTF-8 character in bytes, and zero will be returned.

The `flags` can also contain various flags to allow deviations from the strict UTF-8 encoding (see *utf8.h*).

Most code should use utf8_to_uvchr() rather than call this directly.

```
        UV      utf8n_to_uvuni(U8 *s, STRLEN curlen, STRLEN* retlen, U32 flags)
```

utf8_distance

Returns the number of UTF-8 characters between the UTF-8 pointers a and b.

WARNING: use only if you *know* that the pointers point inside the same UTF-8 buffer.

```
        IV      utf8_distance(U8 *a, U8 *b)
```

utf8_hop

Return the UTF-8 pointer s displaced by off characters, either forward or backward.

WARNING: do not use the following unless you *know* off is within the UTF-8 data pointed to by s *and* that on entry s is aligned on the first byte of character or just after the last byte of a character.

```
        U8*     utf8_hop(U8 *s, I32 off)
```

utf8_length

Return the length of the UTF-8 char encoded string s in characters. Stops at e (inclusive). If e < s or if the scan would end up past e, croaks.

```
        STRLEN  utf8_length(U8* s, U8 *e)
```

utf8_to_bytes

Converts a string s of length len from UTF-8 into byte encoding. Unlike bytes_to_utf8, this over-writes the original string, and updates len to contain the new length. Returns zero on failure, setting len to -1.

NOTE: this function is experimental and may change or be removed without notice.

```
        U8*     utf8_to_bytes(U8 *s, STRLEN *len)
```

utf8_to_uvchr

Returns the native character value of the first character in the string s which is assumed to be in UTF-8 encoding; retlen will be set to the length, in bytes, of that character.

If s does not point to a well-formed UTF-8 character, zero is returned and retlen is set, if possible, to -1.

```
        UV      utf8_to_uvchr(U8 *s, STRLEN* retlen)
```

utf8_to_uvuni

Returns the Unicode code point of the first character in the string s which is assumed to be in UTF-8 encoding; retlen will be set to the length, in bytes, of that character.

This function should only be used when returned UV is considered an index into the Unicode semantic tables (e.g. swashes).

If s does not point to a well-formed UTF-8 character, zero is returned and retlen is set, if possible, to -1.

```
        UV      utf8_to_uvuni(U8 *s, STRLEN* retlen)
```

uvchr_to_utf8

Adds the UTF-8 representation of the Native codepoint uv to the end of the string d; d should be have at least UTF8_MAXLEN+1 free bytes available. The return value is the pointer to the byte after the end of the new character. In other words,

```
    d = uvchr_to_utf8(d, uv);
```

is the recommended wide native character-aware way of saying

```
*(d++) = uv;
```

```
        U8*     uvchr_to_utf8(U8 *d, UV uv)
```

uvuni_to_utf8_flags

Adds the UTF-8 representation of the Unicode codepoint `uv` to the end of the string `d`; `d` should be have at least `UTF8_MAXLEN+1` free bytes available. The return value is the pointer to the byte after the end of the new character. In other words,

```
d = uvuni_to_utf8_flags(d, uv, flags);
```

or, in most cases,

```
d = uvuni_to_utf8(d, uv);
```

(which is equivalent to)

```
d = uvuni_to_utf8_flags(d, uv, 0);
```

is the recommended Unicode-aware way of saying

```
*(d++) = uv;
```

```
        U8*     uvuni_to_utf8_flags(U8 *d, UV uv, UV flags)
```

72.24 Variables created by `xsubpp` and `xsubpp` internal functions

ax

Variable which is setup by `xsubpp` to indicate the stack base offset, used by the `ST`, `XSprePUSH` and `XSRETURN` macros. The `dMARK` macro must be called prior to setup the `MARK` variable.

```
        I32     ax
```

CLASS

Variable which is setup by `xsubpp` to indicate the class name for a C++ XS constructor. This is always a `char*`. See `THIS`.

```
        char*   CLASS
```

dAX

Sets up the `ax` variable. This is usually handled automatically by `xsubpp` by calling `dXSARGS`.

```
                dAX;
```

dITEMS

Sets up the `items` variable. This is usually handled automatically by `xsubpp` by calling `dXSARGS`.

```
                dITEMS;
```

dXSARGS

Sets up stack and mark pointers for an XSUB, calling dSP and dMARK. Sets up the `ax` and `items` variables by calling `dAX` and `dITEMS`. This is usually handled automatically by `xsubpp`.

```
                dXSARGS;
```

dXSI32

Sets up the `ix` variable for an XSUB which has aliases. This is usually handled automatically by `xsubpp`.

```
                dXSI32;
```

items

Variable which is setup by `xsubpp` to indicate the number of items on the stack. See Variable-length Parameter Lists in *perlxs*.

```
        I32     items
```

ix

Variable which is setup by `xsubpp` to indicate which of an XSUB's aliases was used to invoke it. See The ALIAS: Keyword in *perlxs*.

```
        I32     ix
```

newXSproto

Used by `xsubpp` to hook up XSUBs as Perl subs. Adds Perl prototypes to the subs.

RETVAL

Variable which is setup by `xsubpp` to hold the return value for an XSUB. This is always the proper type for the XSUB. See The RETVAL Variable in *perlxs*.

```
        (whatever)      RETVAL
```

ST

Used to access elements on the XSUB's stack.

```
        SV*     ST(int ix)
```

THIS

Variable which is setup by `xsubpp` to designate the object in a C++ XSUB. This is always the proper type for the C++ object. See `CLASS` and Using XS With C++ in *perlxs*.

```
        (whatever)      THIS
```

XS

Macro to declare an XSUB and its C parameter list. This is handled by `xsubpp`.

XS_VERSION

The version identifier for an XS module. This is usually handled automatically by `ExtUtils::MakeMaker`. See `XS_VERSION_BOOTCHECK`.

XS_VERSION_BOOTCHECK

Macro to verify that a PM module's $VERSION variable matches the XS module's `XS_VERSION` variable. This is usually handled automatically by `xsubpp`. See The VERSIONCHECK: Keyword in *perlxs*.

```
                XS_VERSION_BOOTCHECK;
```

72.25 Warning and Dieing

croak

This is the XSUB-writer's interface to Perl's `die` function. Normally call this function the same way you call the C `printf` function. Calling `croak` returns control directly to Perl, sidestepping the normal C order of execution. See `warn`.

If you want to throw an exception object, assign the object to `$@` and then pass `Nullch` to croak():

```
errsv = get_sv("@", TRUE);
sv_setsv(errsv, exception_object);
croak(Nullch);
```

```
        void    croak(const char* pat, ...)
```

warn

This is the XSUB-writer's interface to Perl's `warn` function. Call this function the same way you call the C `printf` function. See `croak`.

```
        void    warn(const char* pat, ...)
```

72.26 AUTHORS

Until May 1997, this document was maintained by Jeff Okamoto <okamoto@corp.hp.com>. It is now maintained as part of Perl itself.

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.

API Listing originally by Dean Roehrich <roehrich@cray.com>.

Updated to be autogenerated from comments in the source by Benjamin Stuhl.

72.27 SEE ALSO

perlguts(1), perlxs(1), perlxstut(1), perlintern(1)

Chapter 73

perlintern

Autogenerated documentation of purely **internal** Perl functions

73.1 DESCRIPTION

This file is the autogenerated documentation of functions in the Perl interpreter that are documented using Perl's internal documentation format but are not marked as part of the Perl API. In other words, **they are not for use in extensions**!

73.2 CV reference counts and CvOUTSIDE

CvWEAKOUTSIDE

Each CV has a pointer, `CvOUTSIDE()`, to its lexically enclosing CV (if any). Because pointers to anonymous sub prototypes are stored in `&` pad slots, it is a possible to get a circular reference, with the parent pointing to the child and vice-versa. To avoid the ensuing memory leak, we do not increment the reference count of the CV pointed to by `CvOUTSIDE` in the *one specific instance* that the parent has a `&` pad slot pointing back to us. In this case, we set the `CvWEAKOUTSIDE` flag in the child. This allows us to determine under what circumstances we should decrement the refcount of the parent when freeing the child.

There is a further complication with non-closure anonymous subs (ie those that do not refer to any lexicals outside that sub). In this case, the anonymous prototype is shared rather than being cloned. This has the consequence that the parent may be freed while there are still active children, eg

```
    BEGIN { $a = sub { eval '$x' } }
```

In this case, the BEGIN is freed immediately after execution since there are no active references to it: the anon sub prototype has `CvWEAKOUTSIDE` set since it's not a closure, and $a points to the same CV, so it doesn't contribute to BEGIN's refcount either. When $a is executed, the `eval '$x'` causes the chain of `CvOUTSIDE`s to be followed, and the freed BEGIN is accessed.

To avoid this, whenever a CV and its associated pad is freed, any `&` entries in the pad are explicitly removed from the pad, and if the refcount of the pointed-to anon sub is still positive, then that child's `CvOUTSIDE` is set to point to its grandparent. This will only occur in the single specific case of a non-closure anon prototype having one or more active references (such as `$a` above).

One other thing to consider is that a CV may be merely undefined rather than freed, eg `undef &foo`. In this case, its refcount may not have reached zero, but we still delete its pad and its `CvROOT` etc. Since various children may still have their `CvOUTSIDE` pointing at this undefined CV, we keep its own `CvOUTSIDE` for the time being, so that the chain of lexical scopes is unbroken. For example, the following should print 123:

```
my $x = 123;
sub tmp { sub { eval '$x' } }
my $a = tmp();
undef &tmp;
print  $a->();
```

```
    bool    CvWEAKOUTSIDE(CV *cv)
```

73.3 Functions in file pad.h

CX_CURPAD_SAVE

Save the current pad in the given context block structure.

```
    void    CX_CURPAD_SAVE(struct context)
```

CX_CURPAD_SV

Access the SV at offset po in the saved current pad in the given context block structure (can be used as an lvalue).

```
    SV *    CX_CURPAD_SV(struct context, PADOFFSET po)
```

PAD_BASE_SV

Get the value from slot po in the base (DEPTH=1) pad of a padlist

```
    SV *    PAD_BASE_SV     (PADLIST padlist, PADOFFSET po)
```

PAD_CLONE_VARS

|CLONE_PARAMS* param Clone the state variables associated with running and compiling pads.

```
    void    PAD_CLONE_VARS(PerlInterpreter *proto_perl \)
```

PAD_COMPNAME_FLAGS

Return the flags for the current compiling pad name at offset po. Assumes a valid slot entry.

```
    U32     PAD_COMPNAME_FLAGS(PADOFFSET po)
```

PAD_COMPNAME_GEN

The generation number of the name at offset po in the current compiling pad (lvalue). Note that SvCUR is hijacked for this purpose.

```
    STRLEN  PAD_COMPNAME_GEN(PADOFFSET po)
```

PAD_COMPNAME_OURSTASH

Return the stash associated with an our variable. Assumes the slot entry is a valid our lexical.

```
    HV *    PAD_COMPNAME_OURSTASH(PADOFFSET po)
```

PAD_COMPNAME_PV

Return the name of the current compiling pad name at offset po. Assumes a valid slot entry.

```
    char *  PAD_COMPNAME_PV(PADOFFSET po)
```

PAD_COMPNAME_TYPE

Return the type (stash) of the current compiling pad name at offset `po`. Must be a valid name. Returns null if not typed.

```
        HV *    PAD_COMPNAME_TYPE(PADOFFSET po)
```

PAD_DUP

Clone a padlist.

```
        void    PAD_DUP(PADLIST dstpad, PADLIST srcpad, CLONE_PARAMS* param)
```

PAD_RESTORE_LOCAL

Restore the old pad saved into the local variable opad by PAD_SAVE_LOCAL()

```
        void    PAD_RESTORE_LOCAL(PAD *opad)
```

PAD_SAVE_LOCAL

Save the current pad to the local variable opad, then make the current pad equal to npad

```
        void    PAD_SAVE_LOCAL(PAD *opad, PAD *npad)
```

PAD_SAVE_SETNULLPAD

Save the current pad then set it to null.

```
        void    PAD_SAVE_SETNULLPAD()
```

PAD_SETSV

Set the slot at offset `po` in the current pad to `sv`

```
        SV *    PAD_SETSV       (PADOFFSET po, SV* sv)
```

PAD_SET_CUR

Set the current pad to be pad `n` in the padlist, saving the previous current pad.

```
        void    PAD_SET_CUR     (PADLIST padlist, I32 n)
```

PAD_SET_CUR_NOSAVE

like PAD_SET_CUR, but without the save

```
        void    PAD_SET_CUR_NOSAVE      (PADLIST padlist, I32 n)
```

PAD_SV

Get the value at offset `po` in the current pad

```
        void    PAD_SV  (PADOFFSET po)
```

PAD_SVl

Lightweight and lvalue version of `PAD_SV`. Get or set the value at offset `po` in the current pad. Unlike `PAD_SV`, does not print diagnostics with -DX. For internal use only.

```
        SV *    PAD_SVl (PADOFFSET po)
```

SAVECLEARSV

Clear the pointed to pad value on scope exit. (ie the runtime action of 'my')

```
        void    SAVECLEARSV     (SV **svp)
```

SAVECOMPPAD

save PL_comppad and PL_curpad

```
        void    SAVECOMPPAD()
```

SAVEPADSV

Save a pad slot (used to restore after an iteration)

XXX DAPM it would make more sense to make the arg a PADOFFSET void SAVEPADSV (PADOFFSET po)

73.4 Functions in file pp_ctl.c

find_runcv

Locate the CV corresponding to the currently executing sub or eval. If db_seqp is non_null, skip CVs that are in the DB package and populate *db_seqp with the cop sequence number at the point that the DB:: code was entered. (allows debuggers to eval in the scope of the breakpoint rather than in in the scope of the debugger itself).

```
        CV*     find_runcv(U32 *db_seqp)
```

73.5 Global Variables

PL_DBsingle

When Perl is run in debugging mode, with the **-d** switch, this SV is a boolean which indicates whether subs are being single-stepped. Single-stepping is automatically turned on after every step. This is the C variable which corresponds to Perl's $DB::single variable. See `PL_DBsub`.

```
        SV *    PL_DBsingle
```

PL_DBsub

When Perl is run in debugging mode, with the **-d** switch, this GV contains the SV which holds the name of the sub being debugged. This is the C variable which corresponds to Perl's $DB::sub variable. See `PL_DBsingle`.

```
        GV *    PL_DBsub
```

PL_DBtrace

Trace variable used when Perl is run in debugging mode, with the **-d** switch. This is the C variable which corresponds to Perl's $DB::trace variable. See `PL_DBsingle`.

```
        SV *    PL_DBtrace
```

PL_dowarn

The C variable which corresponds to Perl's $ˆW warning variable.

```
        bool    PL_dowarn
```

PL_last_in_gv

The GV which was last used for a filehandle input operation. (`<FH>`)

```
GV*     PL_last_in_gv
```

PL_ofs_sv

The output field separator - `$,` in Perl space.

```
SV*     PL_ofs_sv
```

PL_rs

The input record separator - `$/` in Perl space.

```
SV*     PL_rs
```

73.6 GV Functions

is_gv_magical

Returns `TRUE` if given the name of a magical GV.

Currently only useful internally when determining if a GV should be created even in rvalue contexts.

`flags` is not used at present but available for future extension to allow selecting particular classes of magical variable.

```
bool    is_gv_magical(char *name, STRLEN len, U32 flags)
```

73.7 IO Functions

start_glob

Function called by `do_readline` to spawn a glob (or do the glob inside perl on VMS). This code used to be inline, but now perl uses `File::Glob` this glob starter is only used by miniperl during the build process. Moving it away shrinks pp_hot.c; shrinking pp_hot.c helps speed perl up.

```
PerlIO* start_glob(SV* pattern, IO *io)
```

73.8 Pad Data Structures

CvPADLIST

CV's can have CvPADLIST(cv) set to point to an AV.

For these purposes "forms" are a kind-of CV, eval""s are too (except they're not callable at will and are always thrown away after the eval"" is done executing).

XSUBs don't have CvPADLIST set - dXSTARG fetches values from PL_curpad, but that is really the callers pad (a slot of which is allocated by every entersub).

The CvPADLIST AV has does not have AvREAL set, so REFCNT of component items is managed "manual" (mostly in pad.c) rather than normal av.c rules. The items in the AV are not SVs as for a normal AV, but other AVs:

0'th Entry of the CvPADLIST is an AV which represents the "names" or rather the "static type information" for lexicals.

The CvDEPTH'th entry of CvPADLIST AV is an AV which is the stack frame at that depth of recursion into the CV. The 0'th slot of a frame AV is an AV which is @_. other entries are storage for variables and op targets.

During compilation: `PL_comppad_name` is set to the names AV. `PL_comppad` is set to the frame AV for the frame CvDEPTH == 1. `PL_curpad` is set to the body of the frame AV (i.e. AvARRAY(PL_comppad)).

During execution, `PL_comppad` and `PL_curpad` refer to the live frame of the currently executing sub.

Iterating over the names AV iterates over all possible pad items. Pad slots that are SVs_PADTMP (targets/GVs/constants) end up having &PL_sv_undef "names" (see pad_alloc()).

Only my/our variable (SVs_PADMY/SVs_PADOUR) slots get valid names. The rest are op targets/GVs/constants which are statically allocated or resolved at compile time. These don't have names by which they can be looked up from Perl code at run time through eval"" like my/our variables can be. Since they can't be looked up by "name" but only by their index allocated at compile time (which is usually in PL_op->op_targ), wasting a name SV for them doesn't make sense.

The SVs in the names AV have their PV being the name of the variable. NV+1..IV inclusive is a range of cop_seq numbers for which the name is valid. For typed lexicals name SV is SVt_PVMG and SvSTASH points at the type. For `our` lexicals, the type is SVt_PVGV, and GvSTASH points at the stash of the associated global (so that duplicate `our` delarations in the same package can be detected). SvCUR is sometimes hijacked to store the generation number during compilation.

If SvFAKE is set on the name SV then slot in the frame AVs are a REFCNT'ed references to a lexical from "outside". In this case, the name SV does not have a cop_seq range, since it is in scope throughout.

If the 'name' is '&' the corresponding entry in frame AV is a CV representing a possible closure. (SvFAKE and name of '&' is not a meaningful combination currently but could become so if `my sub foo {}` is implemented.)

```
AV *    CvPADLIST(CV *cv)
```

cv_clone

Clone a CV: make a new CV which points to the same code etc, but which has a newly-created pad built by copying the prototype pad and capturing any outer lexicals.

```
CV*     cv_clone(CV* proto)
```

cv_dump

dump the contents of a CV

```
void    cv_dump(CV *cv, char *title)
```

do_dump_pad

Dump the contents of a padlist

```
void    do_dump_pad(I32 level, PerlIO *file, PADLIST *padlist, int full)
```

intro_my

"Introduce" my variables to visible status.

```
U32     intro_my()
```

pad_add_anon

Add an anon code entry to the current compiling pad

```
PADOFFSET       pad_add_anon(SV* sv, OPCODE op_type)
```

pad_add_name

Create a new name in the current pad at the specified offset. If `typestash` is valid, the name is for a typed lexical; set the name's stash to that value. If `ourstash` is valid, it's an our lexical, set the name's GvSTASH to that value

Also, if the name is @.. or %.., create a new array or hash for that slot

If fake, it means we're cloning an existing entry

```
        PADOFFSET       pad_add_name(char *name, HV* typestash, HV* ourstash, bool clone)
```

pad_alloc

Allocate a new my or tmp pad entry. For a my, simply push a null SV onto the end of PL_comppad, but for a tmp, scan the pad from PL_padix upwards for a slot which has no name and and no active value.

```
        PADOFFSET       pad_alloc(I32 optype, U32 tmptype)
```

pad_block_start

Update the pad compilation state variables on entry to a new block

```
        void    pad_block_start(int full)
```

pad_check_dup

Check for duplicate declarations: report any of: * a my in the current scope with the same name; * an our (anywhere in the pad) with the same name and the same stash as `ourstash` `is_our` indicates that the name to check is an 'our' declaration

```
        void    pad_check_dup(char* name, bool is_our, HV* ourstash)
```

pad_findlex

Find a named lexical anywhere in a chain of nested pads. Add fake entries in the inner pads if it's found in an outer one. innercv is the CV *inside* the chain of outer CVs to be searched. If newoff is non-null, this is a run-time cloning: don't add fake entries, just find the lexical and add a ref to it at newoff in the current pad.

```
        PADOFFSET       pad_findlex(char* name, PADOFFSET newoff, CV* innercv)
```

pad_findmy

Given a lexical name, try to find its offset, first in the current pad, or failing that, in the pads of any lexically enclosing subs (including the complications introduced by eval). If the name is found in an outer pad, then a fake entry is added to the current pad. Returns the offset in the current pad, or NOT_IN_PAD on failure.

```
        PADOFFSET       pad_findmy(char* name)
```

pad_fixup_inner_anons

For any anon CVs in the pad, change CvOUTSIDE of that CV from old_cv to new_cv if necessary. Needed when a newly-compiled CV has to be moved to a pre-existing CV struct.

```
        void    pad_fixup_inner_anons(PADLIST *padlist, CV *old_cv, CV *new_cv)
```

pad_free

Free the SV at offet po in the current pad.

```
        void    pad_free(PADOFFSET po)
```

pad_leavemy

Cleanup at end of scope during compilation: set the max seq number for lexicals in this scope and warn of any lexicals that never got introduced.

```
        void    pad_leavemy()
```

pad_new

Create a new compiling padlist, saving and updating the various global vars at the same time as creating the pad itself. The following flags can be OR'ed together:

```
padnew_CLONE        this pad is for a cloned CV
padnew_SAVE         save old globals
padnew_SAVESUB      also save extra stuff for start of sub

    PADLIST*        pad_new(int flags)
```

pad_push

Push a new pad frame onto the padlist, unless there's already a pad at this depth, in which case don't bother creating a new one. If has_args is true, give the new pad an @_ in slot zero.

```
void    pad_push(PADLIST *padlist, int depth, int has_args)
```

pad_reset

Mark all the current temporaries for reuse

```
void    pad_reset()
```

pad_setsv

Set the entry at offset po in the current pad to sv. Use the macro PAD_SETSV() rather than calling this function directly.

```
void    pad_setsv(PADOFFSET po, SV* sv)
```

pad_swipe

Abandon the tmp in the current pad at offset po and replace with a new one.

```
void    pad_swipe(PADOFFSET po, bool refadjust)
```

pad_tidy

Tidy up a pad after we've finished compiling it: * remove most stuff from the pads of anonsub prototypes; * give it a @_; * mark tmps as such.

```
void    pad_tidy(padtidy_type type)
```

pad_undef

Free the padlist associated with a CV. If parts of it happen to be current, we null the relevant PL_*pad* global vars so that we don't have any dangling references left. We also repoint the CvOUTSIDE of any about-to-be-orphaned inner subs to the outer of this cv.

(This function should really be called pad_free, but the name was already taken)

```
void    pad_undef(CV* cv)
```

73.9 Stack Manipulation Macros

djSP

Declare Just SP. This is actually identical to dSP, and declares a local copy of perl's stack pointer, available via the SP macro. See SP. (Available for backward source code compatibility with the old (Perl 5.005) thread model.)

```
        djSP;
```

LVRET

True if this op will be the return value of an lvalue subroutine

73.10 SV Manipulation Functions

report_uninit

Print appropriate "Use of uninitialized variable" warning

```
void    report_uninit()
```

sv_add_arena

Given a chunk of memory, link it to the head of the list of arenas, and split it into a list of free SVs.

```
void    sv_add_arena(char* ptr, U32 size, U32 flags)
```

sv_clean_all

Decrement the refcnt of each remaining SV, possibly triggering a cleanup. This function may have to be called multiple times to free SVs which are in complex self-referential hierarchies.

```
I32     sv_clean_all()
```

sv_clean_objs

Attempt to destroy all objects not yet freed

```
void    sv_clean_objs()
```

sv_free_arenas

Deallocate the memory used by all arenas. Note that all the individual SV heads and bodies within the arenas must already have been freed.

```
void    sv_free_arenas()
```

73.11 AUTHORS

The autodocumentation system was originally added to the Perl core by Benjamin Stuhl. Documentation is by whoever was kind enough to document their functions.

73.12 SEE ALSO

perlguts(1), perlapi(1)

Chapter 74

perliol

C API for Perl's implementation of IO in Layers.

74.1 SYNOPSIS

```
    /* Defining a layer ... */
    #include <perliol.h>
```

74.2 DESCRIPTION

This document describes the behavior and implementation of the PerlIO abstraction described in *perlapio* when `USE_PERLIO` is defined (and `USE_SFIO` is not).

74.2.1 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 perl6.

74.2.2 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*, PERLIO in *perlrun*, 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.

74.2.3 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.

74.2.4 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 */
         IV             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.

74.2.5 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,
                     AV *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.

74.2.6 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.

74.2.7 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 lowlevel "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.

74.2.8 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".)

74.2.9 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,
                        AV *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* AV 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.

Having said all that translation-only layers do not need to provide `Open()` at all, but rather leave the opening to a lower level layer and wait to be "pushed". If a layer does provide `Open()` it should normally call the `Open()` method of next layer down (if any) and then push itself on top if that succeeds.

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 `Dup`ed.

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.)

74.2.10 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.

74.2.11 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 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
```

74.2.12 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.

74.2.13 Extension Layers

Layers can 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.

74.3 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).

Chapter 75

perlapio

Perl's IO abstraction interface.

75.1 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);
    int     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);
    int     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);
    int     PerlIO_get_cnt(PerlIO *f);
    char   *PerlIO_get_ptr(PerlIO *f);
    void    PerlIO_set_ptrcnt(PerlIO *f, char *ptr, int 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);
    int     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,...)
```

75.2 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_SFIO

   A "legacy" implementation in terms of the "sfio" library. Used for some specialist applications on Unix machines ("sfio" is not widely ported away from Unix). Most of above are #define'd to the sfio functions. PerlIO * is in this case Sfio_t *.

3. 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 *perliol*.

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 implementattions). 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 desciptor 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.)

75.2.1 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()`.

75.2.2 "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.

75.2.3 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,Nullch);
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.

Chapter 76

perlhack

How to hack at the Perl internals

76.1 DESCRIPTION

This document attempts to explain how Perl development takes place, and ends with some suggestions for people wanting to become bona fide porters.

The perl5-porters mailing list is where the Perl standard distribution is maintained and developed. The list can get anywhere from 10 to 150 messages a day, depending on the heatedness of the debate. Most days there are two or three patches, extensions, features, or bugs being discussed at a time.

A searchable archive of the list is at either:

```
    http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/
```

or

```
    http://archive.develooper.com/perl5-porters@perl.org/
```

List subscribers (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 the Perl language. Various releases of Perl are shepherded by a "pumpking", a porter responsible for gathering patches, deciding on a patch-by-patch feature-by-feature basis what will and will not go into the release. For instance, Gurusamy Sarathy was the pumpking for the 5.6 release of Perl, and Jarkko Hietaniemi is the pumpking for the 5.8 release, and Hugo van der Sanden will be the pumpking for the 5.10 release.

In addition, various people are pumpkings for different things. For instance, Andy Dougherty and Jarkko Hietaniemi share the *Configure* pumpkin.

Larry sees Perl development along the lines of the US government: there's the Legislature (the porters), the Executive branch (the pumpkings), 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.

New features and extensions to the language are contentious, because the criteria used by the pumpkings, Larry, and other porters to decide which features should be implemented and incorporated are not codified in a few small design goals as with some other languages. Instead, the heuristics are flexible and often difficult to fathom. Here is one person's list, roughly in decreasing order of importance, of heuristics that new features have to be weighed against:

Does concept match the general goals of Perl?

These haven't been written anywhere in stone, but one approximation is:

```
 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.
```

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.

Backwards compatibility

It's a cardinal sin to break existing Perl programs. New warnings are 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.

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. If it can be done in a module instead of in the core, it's highly unlikely to be added.

Is the feature generic enough?

Is this something that only the submitter wants added to the language, or would it be 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. For instance, instead of implementing a "delayed evaluation" feature, the porters are waiting for a macro system that would permit delayed evaluation and much more.

Does it potentially introduce new bugs?

Radical rewrites of large chunks of the Perl interpreter have the potential to introduce new bugs. The smaller and more localized the change, the better.

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.

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.

Is the implementation generic enough to be portable?

The worst patches make use of a system-specific features. It's highly unlikely that nonportable additions to the Perl language will be accepted.

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?

Is there enough documentation?

Patches without documentation are probably ill-thought out or incomplete. Nothing can be added without documentation, so submitting a patch for the appropriate manpages as well as the source code is always a good idea.

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.

Does it create too much work?

Work for the pumpking, work for Perl programmers, work for module authors, ... Perl is supposed to be easy.

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.

If you're on the list, you might hear the word "core" bandied around. It refers to the standard distribution. "Hacking on the core" means you're changing the C source code to the Perl interpreter. "A core module" is one that ships with Perl.

76.1.1 Keeping in sync

The source code to the Perl interpreter, in its different versions, is kept in a repository managed by a revision control system (which is currently the Perforce program, see http://perforce.com/). The pumpkings and a few others have access to the repository to check in changes. Periodically the pumpking for the development version of Perl will release a new version, so the rest of the porters can see what's changed. The current state of the main trunk of repository, and patches that describe the individual changes that have happened since the last public release are available at this location:

```
    http://public.activestate.com/gsar/APC/
    ftp://ftp.linux.activestate.com/pub/staff/gsar/APC/
```

If you're looking for a particular change, or a change that affected a particular set of files, you may find the **Perl Repository Browser** useful:

```
    http://public.activestate.com/cgi-bin/perlbrowse
```

You may also want to subscribe to the perl5-changes mailing list to receive a copy of each patch that gets submitted to the maintenance and development "branches" of the perl repository. See http://lists.perl.org/ for subscription information.

If you are a member of the perl5-porters mailing list, it is a good thing to keep in touch with the most recent changes. If not only to verify if what you would have posted as a bug report isn't already solved in the most recent available perl development branch, also known as perl-current, bleading edge perl, bleedperl or bleadperl.

Needless to say, the source code in perl-current is usually in a perpetual state of evolution. You should expect it to be very buggy. Do **not** use it for any purpose other than testing and development.

Keeping in sync with the most recent branch can be done in several ways, but the most convenient and reliable way is using **rsync**, available at ftp://rsync.samba.org/pub/rsync/ . (You can also get the most recent branch by FTP.)

If you choose to keep in sync using rsync, there are two approaches to doing so:

rsync'ing the source tree

Presuming you are in the directory where your perl source resides and you have rsync installed and available, you can 'upgrade' to the bleadperl using:

```
# rsync -avz rsync://ftp.linux.activestate.com/perl-current/ .
```

This takes care of updating every single item in the source tree to the latest applied patch level, creating files that are new (to your distribution) and setting date/time stamps of existing files to reflect the bleadperl status.

Note that this will not delete any files that were in '.' before the rsync. Once you are sure that the rsync is running correctly, run it with the –delete and the –dry-run options like this:

```
# rsync -avz --delete --dry-run rsync://ftp.linux.activestate.com/perl-current/ .
```

This will *simulate* an rsync run that also deletes files not present in the bleadperl master copy. Observe the results from this run closely. If you are sure that the actual run would delete no files precious to you, you could remove the '–dry-run' option.

You can than check what patch was the latest that was applied by looking in the file **.patch**, which will show the number of the latest patch.

If you have more than one machine to keep in sync, and not all of them have access to the WAN (so you are not able to rsync all the source trees to the real source), there are some ways to get around this problem.

Using rsync over the LAN

Set up a local rsync server which makes the rsynced source tree available to the LAN and sync the other machines against this directory.

From http://rsync.samba.org/README.html :

```
"Rsync uses rsh or ssh for communication. It does not need to be
 setuid and requires no special privileges for installation.  It
 does not require an inetd entry or a daemon.  You must, however,
 have a working rsh or ssh system.  Using ssh is recommended for
 its security features."
```

Using pushing over the NFS

Having the other systems mounted over the NFS, you can take an active pushing approach by checking the just updated tree against the other not-yet synced trees. An example would be

```
#!/usr/bin/perl -w

use strict;
use File::Copy;

my %MF = map {
    m/(\S+)/;
    $1 => [ (stat $1)[2, 7, 9] ];     # mode, size, mtime
    } `cat MANIFEST`;

my %remote = map { $_ => "/$_/pro/3gl/CPAN/perl-5.7.1" } qw(host1 host2);
```

```
        foreach my $host (keys %remote) {
            unless (-d $remote{$host}) {
                print STDERR "Cannot Xsync for host $host\n";
                next;
                }
            foreach my $file (keys %MF) {
                my $rfile = "$remote{$host}/$file";
                my ($mode, $size, $mtime) = (stat $rfile)[2, 7, 9];
                defined $size or ($mode, $size, $mtime) = (0, 0, 0);
                $size == $MF{$file}[1] && $mtime == $MF{$file}[2] and next;
                printf "%4s %-34s %8d %9d  %8d %9d\n",
                    $host, $file, $MF{$file}[1], $MF{$file}[2], $size, $mtime;
                unlink $rfile;
                copy ($file, $rfile);
                utime time, $MF{$file}[2], $rfile;
                chmod $MF{$file}[0], $rfile;
                }
            }
```

though this is not perfect. It could be improved with checking file checksums before updating. Not all NFS systems support reliable utime support (when used over the NFS).

rsync'ing the patches

The source tree is maintained by the pumpking who applies patches to the files in the tree. These patches are either created by the pumpking himself using `diff -c` after updating the file manually or by applying patches sent in by posters on the perl5-porters list. These patches are also saved and rsync'able, so you can apply them yourself to the source files.

Presuming you are in a directory where your patches reside, you can get them in sync with

```
 # rsync -avz rsync://ftp.linux.activestate.com/perl-current-diffs/ .
```

This makes sure the latest available patch is downloaded to your patch directory.

It's then up to you to apply these patches, using something like

```
 # last=`ls -t *.gz | sed q`
 # rsync -avz rsync://ftp.linux.activestate.com/perl-current-diffs/ .
 # find . -name '*.gz' -newer $last -exec gzcat {} \; >blead.patch
 # cd ../perl-current
 # patch -p1 -N <../perl-current-diffs/blead.patch
```

or, since this is only a hint towards how it works, use CPAN-patchaperl from Andreas König to have better control over the patching process.

76.1.2 Why rsync the source tree

It's easier to rsync the source tree

Since you don't have to apply the patches yourself, you are sure all files in the source tree are in the right state.

It's more reliable

While both the rsync-able source and patch areas are automatically updated every few minutes, keep in mind that applying patches may sometimes mean careful hand-holding, especially if your version of the `patch` program does not understand how to deal with new files, files with 8-bit characters, or files without trailing newlines.

76.1.3 Why rsync the patches

It's easier to rsync the patches

If you have more than one machine that you want to keep in track with bleadperl, it's easier to rsync the patches only once and then apply them to all the source trees on the different machines.

In case you try to keep in pace on 5 different machines, for which only one of them has access to the WAN, rsync'ing all the source trees should than be done 5 times over the NFS. Having rsync'ed the patches only once, I can apply them to all the source trees automatically. Need you say more ;-)

It's a good reference

If you do not only like to have the most recent development branch, but also like to **fix** bugs, or extend features, you want to dive into the sources. If you are a seasoned perl core diver, you don't need no manuals, tips, roadmaps, perlguts.pod or other aids to find your way around. But if you are a starter, the patches may help you in finding where you should start and how to change the bits that bug you.

The file **Changes** is updated on occasions the pumpking sees as his own little sync points. On those occasions, he releases a tar-ball of the current source tree (i.e. perl@7582.tar.gz), which will be an excellent point to start with when choosing to use the 'rsync the patches' scheme. Starting with perl@7582, which means a set of source files on which the latest applied patch is number 7582, you apply all succeeding patches available from then on (7583, 7584, ...).

You can use the patches later as a kind of search archive.

Finding a start point

If you want to fix/change the behaviour of function/feature Foo, just scan the patches for patches that mention Foo either in the subject, the comments, or the body of the fix. A good chance the patch shows you the files that are affected by that patch which are very likely to be the starting point of your journey into the guts of perl.

Finding how to fix a bug

If you've found *where* the function/feature Foo misbehaves, but you don't know how to fix it (but you do know the change you want to make), you can, again, peruse the patches for similar changes and look how others apply the fix.

Finding the source of misbehaviour

When you keep in sync with bleadperl, the pumpking would love to *see* that the community efforts really work. So after each of his sync points, you are to 'make test' to check if everything is still in working order. If it is, you do 'make ok', which will send an OK report to perlbug@perl.org. (If you do not have access to a mailer from the system you just finished successfully 'make test', you can do 'make okfile', which creates the file `perl.ok`, which you can than take to your favourite mailer and mail yourself).

But of course, as always, things will not always lead to a success path, and one or more test do not pass the 'make test'. Before sending in a bug report (using 'make nok' or 'make nokfile'), check the mailing list if someone else has reported the bug already and if so, confirm it by replying to that message. If not, you might want to trace the source of that misbehaviour **before** sending in the bug, which will help all the other porters in finding the solution.

Here the saved patches come in very handy. You can check the list of patches to see which patch changed what file and what change caused the misbehaviour. If you note that in the bug report, it saves the one trying to solve it, looking for that point.

If searching the patches is too bothersome, you might consider using perl's bugtron to find more information about discussions and ramblings on posted bugs.

If you want to get the best of both worlds, rsync both the source tree for convenience, reliability and ease and rsync the patches for reference.

76.1.4 Working with the source

Because you cannot use the Perforce client, you cannot easily generate diffs against the repository, nor will merges occur when you update via rsync. If you edit a file locally and then rsync against the latest source, changes made in the remote copy will *overwrite* your local versions!

The best way to deal with this is to maintain a tree of symlinks to the rsync'd source. Then, when you want to edit a file, you remove the symlink, copy the real file into the other tree, and edit it. You can then diff your edited file against the original to generate a patch, and you can safely update the original tree.

Perl's *Configure* script can generate this tree of symlinks for you. The following example assumes that you have used rsync to pull a copy of the Perl source into the *perl-rsync* directory. In the directory above that one, you can execute the following commands:

```
mkdir perl-dev
cd perl-dev
../perl-rsync/Configure -Dmksymlinks -Dusedevel -D"optimize=-g"
```

This will start the Perl configuration process. After a few prompts, you should see something like this:

```
Symbolic links are supported.

Checking how to test for symbolic links...
Your builtin 'test -h' may be broken.
Trying external '/usr/bin/test -h'.
You can test for symbolic links with '/usr/bin/test -h'.

Creating the symbolic links...
(First creating the subdirectories...)
(Then creating the symlinks...)
```

The specifics may vary based on your operating system, of course. After you see this, you can abort the *Configure* script, and you will see that the directory you are in has a tree of symlinks to the *perl-rsync* directories and files.

If you plan to do a lot of work with the Perl source, here are some Bourne shell script functions that can make your life easier:

```
    function edit {
        if [ -L $1 ]; then
            mv $1 $1.orig
                cp $1.orig $1
                vi $1
        else
            /bin/vi $1
                fi
    }

    function unedit {
        if [ -L $1.orig ]; then
            rm $1
                mv $1.orig $1
                fi
    }
```

Replace "vi" with your favorite flavor of editor.

Here is another function which will quickly generate a patch for the files which have been edited in your symlink tree:

```
    mkpatchorig() {
        local diffopts
            for f in ‘find . -name ’*.orig’ | sed s,^\./,,‘
                do
                    case ‘echo $f | sed ’s,.orig$,,;s,.*\.,,’‘ in
                        c)   diffopts=-p ;;
                pod) diffopts=’-F^=’ ;;
                *)   diffopts= ;;
                esac
                    diff -du $diffopts $f ‘echo $f | sed ’s,.orig$,,’‘
                    done
    }
```

This function produces patches which include enough context to make your changes obvious. This makes it easier for the Perl pumpking(s) to review them when you send them to the perl5-porters list, and that means they're more likely to get applied.

This function assumed a GNU diff, and may require some tweaking for other diff variants.

76.1.5 Perlbug administration

There is a single remote administrative interface for modifying bug status, category, open issues etc. using the **RT** *bugtracker* system, maintained by *Robert Spier*. Become an administrator, and close any bugs you can get your sticky mitts on:

```
        http://rt.perl.org
```

The bugtracker mechanism for **perl5** bugs in particular is at:

```
        http://bugs6.perl.org/perlbug
```

To email the bug system administrators:

```
        "perlbug-admin" <perlbug-admin@perl.org>
```

76.1.6 Submitting patches

Always submit patches to *perl5-porters@perl.org*. If you're patching a core module and there's an author listed, send the author a copy (see Patching a core module). This lets other porters review your patch, which catches a surprising number of errors in patches. Either use the diff program (available in source code form from ftp://ftp.gnu.org/pub/gnu/ , or use Johan Vromans' *makepatch* (available from *CPAN/authors/id/JV/*). Unified diffs are preferred, but context diffs are accepted. Do not send RCS-style diffs or diffs without context lines. More information is given in the *Porting/patching.pod* file in the Perl source distribution. Please patch against the latest **development** version (e.g., if you're fixing a bug in the 5.005 track, patch against the latest 5.005_5x version). Only patches that survive the heat of the development branch get applied to maintenance versions.

Your patch should update the documentation and test suite. See Writing a test.

To report a bug in Perl, use the program *perlbug* which comes with Perl (if you can't get Perl to work, send mail to the address *perlbug@perl.org* or *perlbug@perl.com*). Reporting bugs through *perlbug* feeds into the automated bug-tracking system, access to which is provided through the web at http://bugs.perl.org/ . It often pays to check the archives of the perl5-porters mailing list to see whether the bug you're reporting has been reported before, and if so whether it was considered a bug. See above for the location of the searchable archives.

The CPAN testers (http://testers.cpan.org/) are a group of volunteers who test CPAN modules on a variety of platforms. Perl Smokers (http://archives.develooper.com/daily-build@perl.org/) automatically tests Perl source releases on platforms with various configurations. Both efforts welcome volunteers.

It's a good idea to read and lurk for a while before chipping in. That way you'll get to see the dynamic of the conversations, learn the personalities of the players, and hopefully be better prepared to make a useful contribution when do you speak up.

If after all this you still think you want to join the perl5-porters mailing list, send mail to *perl5-porters-subscribe@perl.org*. To unsubscribe, send mail to *perl5-porters-unsubscribe@perl.org*.

To hack on the Perl guts, you'll need to read the following things:

perlguts

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.

You might also want to look at Gisle Aas's illustrated perlguts - there's no guarantee that this will be absolutely up-to-date with the latest documentation in the Perl core, but the fundamentals will be right. (http://gisle.aas.no/perl/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.

The perl5-porters FAQ

This should be available from http://simon-cozens.org/writings/p5p-faq ; alternatively, you can get the FAQ emailed to you by sending mail to `perl5-porters-faq@perl.org`. It contains hints on reading perl5-porters, information on how perl5-porters works and how Perl development in general works.

76.1.7 Finding Your Way Around

Perl maintenance can be split into a number of areas, and certain people (pumpkins) will have responsibility for each area. These areas sometimes correspond to files or directories in the source kit. Among the areas are:

Core modules

Modules shipped as part of the Perl core live in the *lib/* and *ext/* subdirectories: *lib/* is for the pure-Perl modules, and *ext/* contains the core XS modules.

Tests

There are tests for nearly all the modules, built-ins and major bits of functionality. Test files all have a .t suffix. Module tests live in the *lib/* and *ext/* directories next to the module being tested. Others live in *t/*. See Writing a test

Documentation

Documentation maintenance includes looking after everything in the *pod/* directory, (as well as contributing new documentation) and the documentation to the modules in core.

Configure

The configure process is the way we make Perl portable across the myriad of operating systems it supports. Responsibility for the configure, build and installation process, as well as the overall portability of the core code rests with the configure pumpkin - others help out with individual operating systems.

The files involved are the operating system directories, (*win32/*, *os2/*, *vms/* and so on) the shell scripts which generate *config.h* and *Makefile*, as well as the metaconfig files which generate *Configure*. (metaconfig isn't included in the core distribution.)

Interpreter

And of course, there's the core of the Perl interpreter itself. Let's have a look at that in a little more detail.

Before we leave looking at the layout, though, don't forget that *MANIFEST* contains not only the file names in the Perl distribution, but short descriptions of what's in them, too. For an overview of the important files, try this:

```
perl -lne 'print if /^[^\/]+\.[ch]\s+/' MANIFEST
```

76.1.8 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. Compiled code in *perlguts* explains exactly how the compilation stage happens.

Here is a short breakdown of perl's operation:

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 *perlembed*; most of the real action takes place in *perl.c*

First, *perlmain.c* allocates some memory and constructs a Perl interpreter:

```
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; 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) {
    exitstatus = perl_run(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.

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.

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.

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 = CALL_FPTR(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.

76.1.9 Internal Variable Types

You should by now have had a look at *perlguts*, 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 *perlguts*.

Let's take an example of manipulating a PV, from sv_catpvn, in *sv.c*

```
 1  void
 2  Perl_sv_catpvn(pTHX_ register SV *sv, register const char *ptr, register 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.

76.1.10 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 Running.

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 Optimization, 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.

76.1.11 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.

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 *perlguts* for a longer description of the macros used in stack manipulation.

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 how the tied `push` is implemented; see `av_push` in *av.c*:

```
1  PUSHMARK(SP);
2  EXTEND(SP,2);
3  PUSHs(SvTIED_obj((SV*)av, mg));
4  PUSHs(val);
5  PUTBACK;
6  ENTER;
7  call_method("PUSH", G_SCALAR|G_DISCARD);
8  LEAVE;
9  POPSTACK;
```

The lines which concern the mark stack are the first, fifth and last lines: they save away, restore and remove the current position of the argument stack.

Let's examine the whole implementation, for practice:

```
1  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 make the change to the global stack pointer: `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 *perlcall*. We call the `PUSH` method in scalar context, and we're going to discard its return value.

```
POPSTACK;
```

Finally, we remove the value we placed on the mark stack, since we don't need it any more.

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 Localising Changes in *perlguts* for how to use the save stack.

76.1.12 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:

```
PP(pp_add)
{
    dSP; dATARGET; tryAMAGICbin(add,opASSIGN);
    {
      dPOPTOPnnrl_ul;
      SETn( left + right );
      RETURN;
    }
}
```

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 Background and PERL_IMPLICIT_CONTEXT in *perlguts* for information on the `[pad]THX_?` macros.

76.1.13 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.

76.1.14 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. *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: *perlrun* 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'
```

76.1.15 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, but check the manual of the one you're using.

To fire up the debugger, type

```
gdb ./perl
```

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 Internal Functions in *perlguts*!) 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.

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 §76.1.16). You'll have to substitute them yourself, or to invoke cpp on the source code files (see §76.1.13) 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.

76.1.16 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).

76.1.17 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 Internal Functions in *perlguts*. 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:

```
    Perl_sv_2nv (sv=0xa0675d0) at sv.c:1669
    1669        if (!sv)
    (gdb)
```

We can now use `Perl_sv_dump` to investigate the 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.

```
{
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 #

76.1.18 Patching

All right, we've now had a look at how to navigate the Perl sources and some things you'll need to know when fiddling with them. Let's now get on and create a simple patch. 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.

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;
register char *pat = SvPVx(*++MARK, fromlen);
register char *patend = pat + fromlen;
register I32 len;
I32 datumtype;
SV *fromstr;
```

We'll have another string pointer in there:

```
   STRLEN fromlen;
   register char *pat = SvPVx(*++MARK, fromlen);
   register char *patend = pat + fromlen;
 + char *patcopy;
   register 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.

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!

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.
```

All done. Now let's create the patch. *Porting/patching.pod* tells us that if we're making major changes, we should copy the entire directory to somewhere safe before we begin fiddling, and then do

```
    diff -ruN old new > patch
```

However, we know which files we've changed, and we can simply do this:

```
    diff -u pp.c~             pp.c             >  patch
    diff -u t/op/pack.t~      t/op/pack.t      >> patch
    diff -u pod/perlfunc.pod~ pod/perlfunc.pod >> patch
```

We end up with a patch looking a little like this:

```
    --- pp.c~       Fri Jun 02 04:34:10 2000
    +++ pp.c        Fri Jun 16 11:37:25 2000
    @@ -4375,6 +4375,7 @@
         register I32 items;
         STRLEN fromlen;
         register char *pat = SvPVx(*++MARK, fromlen);
    +    char *patcopy;
         register char *patend = pat + fromlen;
         register I32 len;
         I32 datumtype;
    @@ -4405,6 +4406,7 @@
    ...
```

And finally, we submit it, with our rationale, to perl5-porters. Job done!

76.1.19 Patching a core module

This works just like patching anything else, with an extra consideration. Many core modules also live on CPAN. If this is so, patch the CPAN version instead of the core and send the patch off to the module maintainer (with a copy to p5p). This will help the module maintainer keep the CPAN version in sync with the core version without constantly scanning p5p.

76.1.20 Adding a new function to the core

If, as part of a patch to fix a bug, or just because you have an especially good idea, you decide to add a new function to the core, discuss your ideas on p5p well before you start work. It may be that someone else has already attempted to do what you are considering and can give lots of good advice or even provide you with bits of code that they already started (but never finished).

You have to follow all of the advice given above for patching. It is extremely important to test any addition thoroughly and add new tests to explore all boundary conditions that your new function is expected to handle. If your new function is used only by one module (e.g. toke), then it should probably be named S_your_function (for static); on the other hand, if you expect it to accessible from other functions in Perl, you should name it Perl_your_function. See Internal Functions in *perlguts* for more details.

The location of any new code is also an important consideration. Don't just create a new top level .c file and put your code there; you would have to make changes to Configure (so the Makefile is created properly), as well as possibly lots of include files. This is strictly pumpking business.

It is better to add your function to one of the existing top level source code files, but your choice is complicated by the nature of the Perl distribution. Only the files that are marked as compiled static are located in the perl executable. Everything else is located in the shared library (or DLL if you are running under WIN32). So, for example, if a function was only used by functions located in toke.c, then your code can go in toke.c. If, however, you want to call the function from universal.c, then you should put your code in another location, for example util.c.

In addition to writing your c-code, you will need to create an appropriate entry in embed.pl describing your function, then run 'make regen_headers' to create the entries in the numerous header files that perl needs to compile correctly. See Internal Functions in *perlguts* for information on the various options that you can set in embed.pl. You will forget to do this a few (or many) times and you will get warnings during the compilation phase. Make sure that you mention this when you post your patch to P5P; the pumpking needs to know this.

When you write your new code, please be conscious of existing code conventions used in the perl source files. See *perlstyle* for details. Although most of the guidelines discussed seem to focus on Perl code, rather than c, they all apply (except when they don't ;). See also *Porting/patching.pod* file in the Perl source distribution for lots of details about both formatting and submitting patches of your changes.

Lastly, TEST TEST TEST TEST TEST any code before posting to p5p. Test on as many platforms as you can find. Test as many perl Configure options as you can (e.g. MULTIPLICITY). If you have profiling or memory tools, see EXTERNAL TOOLS FOR DEBUGGING PERL below for how to use them to further test your code. Remember that most of the people on P5P are doing this on their own time and don't have the time to debug your code.

76.1.21 Writing a test

Every module and built-in function has an associated test file (or should...). If you add or change functionality, you have to write a test. If you fix a bug, you have to write a test so that bug never comes back. If you alter the docs, it would be nice to test what the new documentation says.

In short, if you submit a patch you probably also have to patch the tests.

For modules, the test file is right next to the module itself. *lib/strict.t* tests *lib/strict.pm*. This is a recent innovation, so there are some snags (and it would be wonderful for you to brush them out), but it basically works that way. Everything else lives in *t/*.

t/base/

Testing of the absolute basic functionality of Perl. Things like `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/

These test the basic control structures, `if/else`, `while`, subroutines, etc.

t/comp/

Tests basic issues of how Perl parses and compiles itself.

t/io/

Tests for built-in IO functions, including command line arguments.

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!

t/op/

Tests for perl's built in functions that don't fit into any of the other directories.

t/pod/

Tests for POD directives. There are still some tests for the Pod modules hanging around in here that need to be moved out into *lib/*.

t/run/

Testing 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/x2p

A test suite for the s2p converter.

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.

t/base t/comp

Since we don't know if require works, or even subroutines, use ad hoc tests for these two. Step carefully to avoid using the feature being tested.

t/cmd t/run t/io t/op

Now that basic require() and subroutines are tested, you can use the t/test.pl library which emulates the important features of Test::More while using a minimum of core features.

You can also conditionally use certain libraries like Config, but be sure to skip the test gracefully if it's not there.

t/lib ext lib

Now that the core of Perl is tested, Test::More can 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. 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 and avoiding things like `fork()` and `system()` unless absolutely necessary.

76.1.22 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.

coretest

Run *perl* on all core tests (*t/** and *lib/[a-z]** pragma tests).

test.deparse

Run all the tests through B::Deparse. Not all tests will succeed.

test.taintwarn

Run all tests with the **-t** command-line switch. Not all tests are expected to succeed (until they're specifically fixed, of course).

minitest

Run *miniperl* on *t/base*, *t/comp*, *t/cmd*, *t/run*, *t/io*, *t/op*, and *t/uni* tests.

test.valgrind check.valgrind utest.valgrind ucheck.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.third check.third utest.third ucheck.third

(Only in Tru64) Run all the tests using the memory leak + naughty memory access tool "Third Degree". The log files will be named *perl3.log.testname*.

test.torture torturetest

Run all the usual tests and some extra tests. As of Perl 5.8.0 the only extra tests are Abigail's JAPHs, *t/japh/abigail.t*.

You can also run the torture test with *t/harness* by giving `-torture` argument to *t/harness*.

utest ucheck test.utf8 check.utf8

Run all the tests with -Mutf8. Not all tests will succeed.

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.

76.1.23 Running tests by hand

You can run part of the test suite by hand by using one 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.)

You can run an individual test by a command similar to

```
./perl -I../lib patho/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 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 PERL_DESTRUCT_LEVEL)

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'.

76.2 EXTERNAL TOOLS FOR DEBUGGING PERL

Sometimes it helps to use external tools while debugging and testing Perl. This section tries to guide you through using some common testing and debugging tools with Perl. This is meant as a guide to interfacing these tools with Perl, not as any kind of guide to the use of the tools themselves.

NOTE 1: Running under 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.

NOTE 2: To minimize the number of memory leak false alarms (see PERL_DESTRUCT_LEVEL for more information), you have to have environment variable PERL_DESTRUCT_LEVEL set to 2. The *TEST* and harness scripts do that automatically. But if you are running some of the tests manually– for csh-like shells:

```
setenv PERL_DESTRUCT_LEVEL 2
```

and for Bourne-type shells:

```
PERL_DESTRUCT_LEVEL=2
export PERL_DESTRUCT_LEVEL
```

or in UNIXy environments you can also use the `env` command:

```
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.

76.2.1 Rational Software's Purify

Purify is a commercial tool that is helpful in identifying memory overruns, wild pointers, memory leaks and other such badness. Perl must be compiled in a specific way for optimal testing with Purify. Purify is available under Windows NT, Solaris, HP-UX, SGI, and Siemens Unix.

76.2.2 Purify on Unix

On Unix, Purify creates a new Perl binary. To get the most benefit out of Purify, you should create the perl to Purify using:

```
sh Configure -Accflags=-DPURIFY -Doptimize='-g' \
 -Uusemymalloc -Dusemultiplicity
```

where these arguments mean:

-Accflags=-DPURIFY

Disables Perl's arena memory allocation functions, as well as forcing use of memory allocation functions derived from the system malloc.

-Doptimize='-g'

Adds debugging information so that you see the exact source statements where the problem occurs. Without this flag, all you will see is the source filename of where the error occurred.

-Uusemymalloc

Disable Perl's malloc so that Purify can more closely monitor allocations and leaks. Using Perl's malloc will make Purify report most leaks in the "potential" leaks category.

-Dusemultiplicity

Enabling the multiplicity option allows perl to clean up thoroughly when the interpreter shuts down, which reduces the number of bogus leak reports from Purify.

Once you've compiled a perl suitable for Purify'ing, then you can just:

```
make pureperl
```

which creates a binary named 'pureperl' that has been Purify'ed. This binary is used in place of the standard 'perl' binary when you want to debug Perl memory problems.

As an example, to show any memory leaks produced during the standard Perl testset you would create and run the Purify'ed perl as:

```
make pureperl
cd t
../pureperl -I../lib harness
```

which would run Perl on test.pl and report any memory problems.

Purify outputs messages in "Viewer" windows by default. If you don't have a windowing environment or if you simply want the Purify output to unobtrusively go to a log file instead of to the interactive window, use these following options to output to the log file "perl.log":

```
setenv PURIFYOPTIONS "-chain-length=25 -windows=no \
 -log-file=perl.log -append-logfile=yes"
```

If you plan to use the "Viewer" windows, then you only need this option:

```
setenv PURIFYOPTIONS "-chain-length=25"
```

In Bourne-type shells:

```
PURIFYOPTIONS="..."
export PURIFYOPTIONS
```

or if you have the "env" utility:

```
env PURIFYOPTIONS="..." ../pureperl ...
```

76.2.3 Purify on NT

Purify on Windows NT instruments the Perl binary 'perl.exe' on the fly. There are several options in the makefile you should change to get the most use out of Purify:

DEFINES

You should add -DPURIFY to the DEFINES line so the DEFINES line looks something like:

```
DEFINES = -DWIN32 -D_CONSOLE -DNO_STRICT $(CRYPT_FLAG) -DPURIFY=1
```

to disable Perl's arena memory allocation functions, as well as to force use of memory allocation functions derived from the system malloc.

USE_MULTI = define

Enabling the multiplicity option allows perl to clean up thoroughly when the interpreter shuts down, which reduces the number of bogus leak reports from Purify.

#PERL_MALLOC = define

Disable Perl's malloc so that Purify can more closely monitor allocations and leaks. Using Perl's malloc will make Purify report most leaks in the "potential" leaks category.

CFG = Debug

Adds debugging information so that you see the exact source statements where the problem occurs. Without this flag, all you will see is the source filename of where the error occurred.

As an example, to show any memory leaks produced during the standard Perl testset you would create and run Purify as:

```
cd win32
make
cd ../t
purify ../perl -I../lib harness
```

which would instrument Perl in memory, run Perl on test.pl, then finally report any memory problems.

76.2.4 valgrind

The excellent valgrind tool can be used to find out both memory leaks and illegal memory accesses. As of August 2003 it unfortunately works only on x86 (ELF) Linux. The special "test.valgrind" target can be used to run the tests under valgrind. Found errors and memory leaks are logged in files named *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://developer.kde.org/~sewardj/
```

76.2.5 Compaq's/Digital's/HP's Third Degree

Third Degree is a tool for memory leak detection and memory access checks. It is one of the many tools in the ATOM toolkit. The toolkit is only available on Tru64 (formerly known as Digital UNIX formerly known as DEC OSF/1).

When building Perl, you must first run Configure with -Doptimize=-g and -Uusemymalloc flags, after that you can use the make targets "perl.third" and "test.third". (What is required is that Perl must be compiled using the `-g` flag, you may need to re-Configure.)

The short story is that with "atom" you can instrument the Perl executable to create a new executable called *perl.third*. When the instrumented executable is run, it creates a log of dubious memory traffic in file called *perl.3log*. See the manual pages of atom and third for more information. The most extensive Third Degree documentation is available in the Compaq "Tru64 UNIX Programmer's Guide", chapter "Debugging Programs with Third Degree".

The "test.third" leaves a lot of files named *foo_bar.3log* in the t/ subdirectory. There is a problem with these files: Third Degree is so effective that it finds problems also in the system libraries. Therefore you should used the Porting/thirdclean script to cleanup the **.3log* files.

There are also leaks that for given certain definition of a leak, aren't. See PERL_DESTRUCT_LEVEL for more information.

76.2.6 PERL_DESTRUCT_LEVEL

If you want to run any of the tests yourself manually using e.g. valgrind, or the pureperl or perl.third executables, 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 "third-degreed" Perl:

```
    env PERL_DESTRUCT_LEVEL=2 ./perl.third -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; 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.

76.2.7 Profiling

Depending on your platform there are various 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.

76.2.8 Gprof Profiling

gprof is a profiling tool available in many UNIX platforms, it uses *statistical time-sampling*.

You can build a profiled version of perl called "perl.gprof" by invoking the make target "perl.gprof" (What is required is that Perl must be compiled using the `-pg` flag, you may need to re-Configure). Running the profiled version of Perl will create an output file called *gmon.out* is created which contains the profiling data collected during the execution.

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.

76.2.9 GCC gcov Profiling

Starting from GCC 3.0 *basic block profiling* is officially available for the GNU CC.

You can build a profiled version of perl called *perl.gcov* by invoking the make target "perl.gcov" (what is required that Perl must be compiled using gcc with the flags `-fprofile-arcs -ftest-coverage`, you may need to re-Configure).

Running the profiled version of Perl will cause profile output to be generated. For each source file an accompanying ".da" 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.

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 3.0 see

```
    http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc.html
```

and its section titled "8. gcov: a Test Coverage Program"

```
    http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html#SEC132
```

76.2.10 Pixie Profiling

Pixie is a profiling tool available on IRIX and Tru64 (aka Digital UNIX aka DEC OSF/1) platforms. Pixie does its profiling using *basic-block counting*.

You can build a profiled version of perl called *perl.pixie* by invoking the make target "perl.pixie" (what is required is that Perl must be compiled using the `-g` flag, you may need to re-Configure).

In Tru64 a file called *perl.Addrs* will also be silently created, this file contains the addresses of the basic blocks. Running the profiled version of Perl will create a new file called "perl.Counts" which contains the counts for the basic block for that particular program execution.

To display the results you use the *prof* utility. The exact incantation depends on your operating system, "prof perl.Counts" in IRIX, and "prof -pixie -all -L. perl" in Tru64.

In IRIX the following prof options are available:

-h

Reports the most heavily used lines in descending order of use. Useful for finding the hotspot lines.

-l

Groups lines by procedure, with procedures sorted in descending order of use. Within a procedure, lines are listed in source order. Useful for finding the hotspots of procedures.

In Tru64 the following options are available:

-p[rocedures]

Procedures sorted in descending order by the number of cycles executed in each procedure. Useful for finding the hotspot procedures. (This is the default option.)

-h[eavy]

Lines sorted in descending order by the number of cycles executed in each line. Useful for finding the hotspot lines.

-i[nvocations]

The called procedures are sorted in descending order by number of calls made to the procedures. Useful for finding the most used procedures.

-l[ines]

Grouped by procedure, sorted by cycles executed per procedure. Useful for finding the hotspots of procedures.

-testcoverage

The compiler emitted code for these lines, but the code was unexecuted.

-z[ero]

Unexecuted procedures.

For further information, see your system's manual pages for pixie and prof.

76.2.11 Miscellaneous tricks

- 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.

- If you see in a debugger a memory area mysteriously full of 0xabababab, you may be seeing the effect of the Poison() macro, see *perlclib*.

76.2.12 CONCLUSION

We've had a brief look around the Perl source, an overview of the stages *perl* goes through when it's running your code, and how to use a debugger to poke at the Perl guts. We took a very simple problem and demonstrated how to solve it fully - with documentation, regression tests, and finally a patch for submission to p5p. Finally, we talked about how to use external tools to debug and test Perl.

I'd now suggest you read over those references again, and then, as soon as possible, get your hands dirty. The best way to learn is by doing, so:

- 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...
- Keep up to date with the bleeding edge Perl distributions and get familiar with the changes. Try and get an idea of what areas people are working on and the changes they're making.
- 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.

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!

76.3 AUTHOR

This document was written by Nathan Torkington, and is maintained by the perl5-porters mailing list.

Part V

Miscellaneous

Chapter 77

perlbook

Perl book information

77.1 DESCRIPTION

The Camel Book, officially known as *Programming Perl, Third Edition*, by Larry Wall et al, is the definitive reference work covering nearly all of Perl. You can order it and other Perl books from O'Reilly & Associates, 1-800-998-9938. Local/overseas is +1 707 829 0515. If you can locate an O'Reilly order form, you can also fax to +1 707 829 0104. If you're web-connected, you can even mosey on over to http://www.oreilly.com/ for an online order form.

Other Perl books from various publishers and authors can be found listed in *perlfaq2*.

Chapter 78

perltodo

Perl TO-DO List

78.1 DESCRIPTION

This is a list of wishes for Perl. Send updates to *perl5-porters@perl.org*. If you want to work on any of these projects, be sure to check the perl5-porters archives for past ideas, flames, and propaganda. This will save you time and also prevent you from implementing something that Larry has already vetoed. One set of archives may be found at:

```
    http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/
```

78.2 assertions

Clean up and finish support for assertions. See *assertions*.

78.3 iCOW

Sarathy and Arthur have a proposal for an improved Copy On Write which specifically will be able to COW new ithreads. If this can be implemented it would be a good thing.

78.4 (?{...}) closures in regexps

Fix (or rewrite) the implementation of the `/(?{...})/` closures.

78.5 A re-entrant regexp engine

This will allow the use of a regex from inside (?{ }), (??{ }) and (?(?{ })|) constructs.

78.6 pragmata

78.6.1 lexical pragmas

Reimplement the mechanism of lexical pragmas to be more extensible. Fix current pragmas that don't work well (or at all) with lexical scopes or in run-time eval(STRING) (`sort`, `re`, `encoding` for example). MJD has a preliminary patch that implements this.

78.6.2 use less 'memory'

Investigate trade offs to switch out perl's choices on memory usage. Particularly perl should be able to give memory back.

78.7 prototypes and functions

78.7.1 _ prototype character

Study the possibility of adding a new prototype character, _, meaning "this argument defaults to $_".

78.7.2 inlining autoloaded constants

Currently the optimiser can inline constants when expressed as subroutines with prototype ($) that return a constant. Likewise, many packages wrapping C libraries export lots of constants as subroutines which are AUTOLOADed on demand. However, these have no prototypes, so can't be seen as constants by the optimiser. Some way of cheaply (low syntax, low memory overhead) to the perl compiler that a name is a constant would be great, so that it knows to call the AUTOLOAD routine at compile time, and then inline the constant.

78.7.3 Finish off lvalue functions

The old perltodo notes "They don't work in the debugger, and they don't work for list or hash slices."

78.8 Unicode and UTF8

78.8.1 Implicit Latin 1 => Unicode translation

Conversions from byte strings to UTF-8 currently map high bit characters to Unicode without translation (or, depending on how you look at it, by implicitly assuming that the byte strings are in Latin-1). As perl assumes the C locale by default, upgrading a string to UTF-8 may change the meaning of its contents regarding character classes, case mapping, etc. This should probably emit a warning (at least).

78.8.2 UTF8 caching code

The string position/offset cache is not optional. It should be.

78.8.3 Unicode in Filenames

chdir, chmod, chown, chroot, exec, glob, link, lstat, mkdir, open, opendir, qx, readdir, readlink, rename, rmdir, stat, symlink, sysopen, system, truncate, unlink, utime, -X. All these could potentially accept Unicode filenames either as input or output (and in the case of system and qx Unicode in general, as input or output to/from the shell). Whether a filesystem - an operating system pair understands Unicode in filenames varies.

Known combinations that have some level of understanding include Microsoft NTFS, Apple HFS+ (In Mac OS 9 and X) and Apple UFS (in Mac OS X), NFS v4 is rumored to be Unicode, and of course Plan 9. How to create Unicode filenames, what forms of Unicode are accepted and used (UCS-2, UTF-16, UTF-8), what (if any) is the normalization form used, and so on, varies. Finding the right level of interfacing to Perl requires some thought. Remember that an OS does not implicate a filesystem.

(The Windows -C command flag "wide API support" has been at least temporarily retired in 5.8.1, and the -C has been repurposed, see *perlrun*.)

78.8.4 Unicode in %ENV

Currently the %ENV entries are always byte strings.

78.9 Regexps

78.9.1 regexp optimiser optional

The regexp optimiser is not optional. It should configurable to be, to allow its performance to be measured, and its bugs to be easily demonstrated.

78.9.2 common suffices/prefices in regexps (trie optimization)

Currently, the user has to optimize `foo|far` and `foo|goo` into `f(?:oo|ar)` and `[fg]oo` by hand; this could be done automatically.

78.10 POD

78.10.1 POD -> HTML conversion still sucks

Which is crazy given just how simple POD purports to be, and how simple HTML can be.

78.11 Misc medium sized projects

78.11.1 UNITCHECK

Introduce a new special block, UNITCHECK, which is run at the end of a compilation unit (module, file, eval(STRING) block). This will correspond to the Perl 6 CHECK. Perl 5's CHECK cannot be changed or removed because the O.pm/B.pm backend framework depends on it.

78.11.2 optional optimizer

Make the peephole optimizer optional.

78.11.3 You WANT *how* many

Currently contexts are void, scalar and list. split has a special mechanism in place to pass in the number of return values wanted. It would be useful to have a general mechanism for this, backwards compatible and little speed hit. This would allow proposals such as short circuiting sort to be implemented as a module on CPAN.

78.11.4 lexical aliases

Allow lexical aliases (maybe via the syntax `my \$alias = \$foo`.

78.11.5 no 6

Make `no 6` and `no v6` work (opposite of `use 5.005`, etc.).

78.11.6 IPv6

Clean this up. Check everything in core works

78.11.7 entersub XS vs Perl

At the moment pp_entersub is huge, and has code to deal with entering both perl and and XS subroutines. Subroutine implementations rarely change between perl and XS at run time, so investigate using 2 ops to enter subs (one for XS, one for perl) and swap between if a sub is redefined.

78.11.8 @INC source filter to Filter::Simple

The second return value from a sub in @INC can be a source filter. This isn't documented. It should be changed to use Filter::Simple, tested and documented.

78.11.9 bincompat functions

There are lots of functions which are retained for binary compatibility. Clean these up. Move them to mathom.c, and don't compile for blead?

78.11.10 Use fchown/fchmod internally

The old perltodo notes "This has been done in places, but needs a thorough code review. Also fchdir is available in some platforms."

78.12 Tests

78.12.1 Make Schwern poorer

Tests for everything, At which point Schwern coughs up $500 to TPF.

78.12.2 test B

A test suite for the B module would be nice.

78.12.3 Improve tests for Config.pm

Config.pm doesn't appear to be well tested.

78.12.4 common test code for timed bailout

Write portable self destruct code for tests to stop them burning CPU in infinite loops. Needs to avoid using alarm, as some of the tests are testing alarm/sleep or timers.

78.13 Installation

78.13.1 compressed man pages

Be able to install them

78.13.2 Make Config.pm cope with differences between build and installed perl

78.13.3 Relocatable perl

Make it possible to create a relocatable perl binary. Will need some collusion with Config.pm. We could use a syntax of ... for location of current binary?

78.13.4 make HTML install work

78.13.5 put patchlevel in -v

Currently perl from p4/rsync ships with a patchlevel.h file that usually defines one local patch, of the form "MAINT12345" or "RC1". The output of perl -v doesn't report that a perl isn't an official release, and this information can get lost in bugs reports. Because of this, the minor version isn't bumped up util RC time, to minimise the possibility of versions of perl escaping that believe themselves to be newer than they actually are.

It would be useful to find an elegant way to have the "this is an interim maintenance release" or "this is a release candidate" in the terse -v output, and have it so that it's easy for the pumpking to remove this just as the release tarball is rolled up. This way the version pulled out of rsync would always say "I'm a development release" and it would be safe to bump the reported minor version as soon as a release ships, which would aid perl developers.

78.14 Incremental things

Some tasks that don't need to get done in one big hit.

78.14.1 autovivification

Make all autovivification consistent w.r.t LVALUE/RVALUE and strict/no strict;

78.14.2 fix tainting bugs

Fix the bugs revealed by running the test suite with the `-t` switch (via `make test.taintwarn`).

78.14.3 Make tainting consistent

Tainting would be easier to use if it didn't take documented shortcuts and allow taint to "leak" everywhere within an expression.

78.14.4 Dual life everything

As part of the "dists" plan, anything that doesn't belong in the smallest perl distribution needs to be dual lifed. Anything else can be too.

78.15 Vague things

Some more nebulous ideas

78.15.1 threads

Make threads more robust.

78.15.2 POSIX memory footprint

Ilya observed that use POSIX; eats memory like there's no tomorrow, and at various times worked to cut it down. There is probably still fat to cut out - for example POSIX passes Exporter some very memory hungry data structures.

78.15.3 Optimize away @_

The old perltodo notes "Look at the "reification" code in `av.c`"

78.15.4 switch ops

The old perltodo notes "Although we have `Switch.pm` in core, Larry points to the dormant `nswitch` and `cswitch` ops in *pp.c*; using these opcodes would be much faster."

78.15.5 Attach/detach debugger from running program

The old perltodo notes "With `gdb`, you can attach the debugger to a running program if you pass the process ID. It would be good to do this with the Perl debugger on a running Perl program, although I'm not sure how it would be done." ssh and screen do this with named pipes in tmp. Maybe we can too.

78.15.6 A decent benchmark

perlbench seems impervious to any recent changes made to the perl core. It would be useful to have a reasonable general benchmarking suite that roughly represented what current perl programs do, and measurably reported whether tweaks to the core improve, degrade or don't really affect performance, to guide people attempting to optimise the guts of perl.

Chapter 79

perldoc

Look up Perl documentation in Pod format.

79.1 SYNOPSIS

perldoc [**-h**] [**-v**] [**-t**] [**-u**] [**-m**] [**-l**] [**-F**] [**-i**] [**-V**] [**-T**] [**-r**] [**-d*destination_file***] [**-o*formatname***] [**-M*FormatterClassName***] [**-w*formatteroption:value***] [**-n***nroff-replacement*] [**-X**] PageName|ModuleName|ProgramName

perldoc -f BuiltinFunction

perldoc -q FAQ Keyword

See below for more description of the switches.

79.2 DESCRIPTION

perldoc looks up a piece of documentation in .pod format that is embedded in the perl installation tree or in a perl script, and displays it via `pod2man | nroff -man | $PAGER`. (In addition, if running under HP-UX, `col -x` will be used.) This is primarily used for the documentation for the perl library modules.

Your system may also have man pages installed for those modules, in which case you can probably just use the man(1) command.

If you are looking for a table of contents to the Perl library modules documentation, see the *perltoc* page.

79.3 OPTIONS

-h

Prints out a brief **h**elp message.

-v

Describes search for the item in detail (**v**erbosely).

-t

Display docs using plain **t**ext converter, instead of nroff. This may be faster, but it probably won't look as nice.

-u

Skip the real Pod formatting, and just show the raw Pod source (**U**nformatted)

-m *module*

Display the entire module: both code and unformatted pod documentation. This may be useful if the docs don't explain a function in the detail you need, and you'd like to inspect the code directly; perldoc will find the file for you and simply hand it off for display.

-l

Display on**l**y the file name of the module found.

-F

Consider arguments as file names; no search in directories will be performed.

-f *perlfunc*

The **-f** option followed by the name of a perl built in function will extract the documentation of this function from *perlfunc*.

Example:

```
    perldoc -f sprintf
```

-q *perlfaq-search-regexp*

The **-q** option takes a regular expression as an argument. It will search the **q**uestion headings in perlfaq[1-9] and print the entries matching the regular expression. Example: `perldoc -q shuffle`

-T

This specifies that the output is not to be sent to a pager, but is to be sent right to STDOUT.

-d *destination-filename*

This specifies that the output is to be sent neither to a pager nor to STDOUT, but is to be saved to the specified filename. Example: `perldoc -oLaTeX -dtextwrapdocs.tex Text::Wrap`

-o *output-formatname*

This specifies that you want Perldoc to try using a Pod-formatting class for the output format that you specify. For example: `-oman`. This is actually just a wrapper around the `-M` switch; using `-o`*`formatname`* just looks for a loadable class by adding that format name (with different capitalizations) to the end of different classname prefixes.

For example, `-oLaTeX` currently tries all of the following classes: Pod::Perldoc::ToLaTeX Pod::Perldoc::Tolatex Pod::Perldoc::ToLatex Pod::Perldoc::ToLATEX Pod::Simple::LaTeX Pod::Simple::latex Pod::Simple::Latex Pod::Simple::LATEX Pod::LaTeX Pod::latex Pod::Latex Pod::LATEX.

-M *module-name*

This specifies the module that you want to try using for formatting the pod. The class must must at least provide a `parse_from_file` method. For example: `perldoc -MPod::Perldoc::ToChecker`.

You can specify several classes to try by joining them with commas or semicolons, as in `-MTk::SuperPod;Tk::Pod`.

-w *option:value* or -w *option*

This specifies an option to call the formatter **w**ith. For example, `-w textsize:15` will call `$formatter->textsize(15)` on the formatter object before it is used to format the object. For this to be valid, the formatter class must provide such a method, and the value you pass should be valid. (So if `textsize` expects an integer, and you do `-w textsize:big`, expect trouble.)

You can use `-w optionname` (without a value) as shorthand for `-w optionname:`*`TRUE`*. This is presumably useful in cases of on/off features like: `-w page_numbering`.

You can use a "=" instead of the ":", as in: `-w textsize=15`. This might be more (or less) convenient, depending on what shell you use.

-X

Use an index if it is present – the **-X** option looks for an entry whose basename matches the name given on the command line in the file `$Config{archlib}/pod.idx`. The *pod.idx* file should contain fully qualified filenames, one per line.

PageName|ModuleName|ProgramName

The item you want to look up. Nested modules (such as `File::Basename`) are specified either as `File::Basename` or `File/Basename`. You may also give a descriptive name of a page, such as `perlfunc`.

-n *some-formatter*

Specify replacement for nroff

-r

Recursive search.

-i

Ignore case.

-V

Displays the version of perldoc you're running.

79.4 SECURITY

Because **perldoc** does not run properly tainted, and is known to have security issues, when run as the superuser it will attempt to drop privileges by setting the effective and real IDs to nobody's or nouser's account, or -2 if unavailable. If it cannot relinquish its privileges, it will not run.

79.5 ENVIRONMENT

Any switches in the `PERLDOC` environment variable will be used before the command line arguments.

Useful values for `PERLDOC` include `-oman`, `-otext`, `-otk`, `-ortf`, `-oxml`, and so on, depending on what modules you have on hand; or exactly specify the formatter class with `-MPod::Perldoc::ToMan` or the like.

`perldoc` also searches directories specified by the `PERL5LIB` (or `PERLLIB` if `PERL5LIB` is not defined) and `PATH` environment variables. (The latter is so that embedded pods for executables, such as `perldoc` itself, are available.)

`perldoc` will use, in order of preference, the pager defined in `PERLDOC_PAGER`, `MANPAGER`, or `PAGER` before trying to find a pager on its own. (`MANPAGER` is not used if `perldoc` was told to display plain text or unformatted pod.)

One useful value for `PERLDOC_PAGER` is `less -+C -E`.

Having PERLDOCDEBUG set to a positive integer will make perldoc emit even more descriptive output than the `-v` switch does – the higher the number, the more it emits.

79.6 AUTHOR

Current maintainer: Sean M. Burke, <sburke@cpan.org>

Past contributors are: Kenneth Albanowski <kjahds@kjahds.com>, Andy Dougherty <doughera@lafcol.lafayette.edu>, and many others.

Chapter 80

perlhist

The Perl history records

80.1 DESCRIPTION

This document aims to record the Perl source code releases.

80.2 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.
```

80.3 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.

80.3.1 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 once 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?).

80.4 THE RECORDS

```
Pump-  Release         Date            Notes
king                                   (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.
          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

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.
         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
         5.8.0         2002-Jul-18
         5.8.1-RC1     2003-Jul-10
         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

 Hugo     5.9.0         2003-Oct-27
 Rafael   5.9.1         2004-Mar-16
```

80.4.1 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          2930  80   2626 364   1096 129    868 281   2841  93
5.7.0          2977  80   2801 425   1250 132    975 307   3206 100
5.6.1          3049  80   3764 484   1924 159   1025 304   3593 119
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          3415  87   5367 630  14448 410   2205 640   4491 148
```

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}
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   4.000   4.019   4.036

atarist      -  -    -  -    -  -    -  -    -  -  113 31
Configure   31  1   37  1   62  1   73  1   83  1   86  1
eg           -  -   34 28   47 39   47 39   47 39   47 39
emacs        -  -    -  -    -  -   67  4   67  4   67  4
h2pl         -  -    -  -   12 12   12 12   12 12   12 12
hints        -  -    -  -    -  -    -  -    5 42   11 56
msdos        -  -    -  -   41 13   57 15   58 15   60 15
os2          -  -    -  -   63 22   81 29   81 29  113 31
usub         -  -    -  -   21 16   25  7   43  8   43  8
x2p        103 17  104 17  137 17  147 18  152 19  154 19

======================================================================

           5.000a2 5.000a12h 5.000b3h 5.000  5.001m  5.002   5.003

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  201  1  201  1
Doc          -  -   26  1    75  7     11  1   11  1    -  -    -  -
eg          79 58   53 44    51 43     54 44   54 44   54 44   54 44
emacs       67  4  104  6   104  6    104  1  104  6  108  1  108  1
h2pl        12 12   12 12    12 12     12 12   12 12   12 12   12 12
hints       11 56   12 46    18 48     18 48   44 56   73 59   77 60
msdos       60 15   60 15     -  -      -  -    -  -    -  -    -  -
os2        113 31  113 31     -  -      -  -    -  -   84 17   56 10
U            -  -   62  8   112 42      -  -    -  -    -  -    -  -
usub        43  8    -  -     -  -      -  -    -  -    -  -    -  -
utils        -  -    -  -     -  -      -  -    -  -   87  7   88  7
vms          -  -   80  7   123  9    184 15  304 20  500 24  475 26
x2p        171 22  171 21   162 20    162 20  279 20  280 20  280 20
```

```
======================================================================

           5.003_07 5.004   5.004_04 5.004_62 5.004_65 5.004_68

beos         -  -     -  -    -  -     -  -     1   1    1   1
Configure  217  1   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          54 44    81 62   81 62    81 62    81  62   81  62
emacs      143  1   194  1  204  1   212  2   212   2  212   2
h2pl        12 12    12 12   12 12    12 12    12  12   12  12
hints       90 62   129 69  132 71   144 72   151  74  155  74
os2        117 42   121 42  127 42   127 44   129  44  129  44
plan9       79 15    82 15   82 15    82 15    82  15   82  15
Porting     51  1    94  2  109  4   203  6   234   8  241   9
qnx          -  -     1  2    1  2     1  2     1   2    1   2
utils       97  7   112  8  118  8   124  8   156   9  159   9
vms        505 27   518 34  524 34   538 34   569  34  569  34
win32        -  -   285 33  378 36   470 39   493  39  575  41
x2p        280 19   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
```

80.4.2 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-  Release         Date           diff lines kB
king                                  -------------
                                         +   -   !
===========================================================================
 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
```

80.5 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.

Chapter 81

perldelta

What is new for perl v5.8.5

81.1 DESCRIPTION

This document describes differences between the 5.8.4 release and the 5.8.5 release.

81.2 Incompatible Changes

There are no changes incompatible with 5.8.4.

81.3 Core Enhancements

Perl's regular expression engine now contains support for matching on the intersection of two Unicode character classes. You can also now refer to user-defined character classes from within other user defined character classes.

81.4 Modules and Pragmata

- Carp improved to work nicely with Safe. Carp's message reporting should now be anomaly free - it will always print out line number information.
- CGI upgraded to version 3.05
- charnames now avoids clobbering $_
- Digest upgraded to version 1.08
- Encode upgraded to version 2.01
- FileCache upgraded to version 1.04
- libnet upgraded to version 1.19
- Pod::Parser upgraded to version 1.28
- Pod::Perldoc upgraded to version 3.13
- Pod::LaTeX upgraded to version 0.57
- Safe now works properly with Carp

- Scalar-List-Utils upgraded to version 1.14
- Shell's documentation has been re-written, and its historical partial auto-quoting of command arguments can now be disabled.
- Test upgraded to version 1.25
- Test::Harness upgraded to version 2.42
- Time::Local upgraded to version 1.10
- Unicode::Collate upgraded to version 0.40
- Unicode::Normalize upgraded to version 0.30

81.5 Utility Changes

81.5.1 Perl's debugger

The debugger can now emulate stepping backwards, by restarting and rerunning all bar the last command from a saved command history.

81.5.2 h2ph

h2ph is now able to understand a very limited set of C inline functions – basically, the inline functions that look like CPP macros. This has been introduced to deal with some of the headers of the newest versions of the glibc. The standard warning still applies; to quote *h2ph*'s documentation, *you may need to dicker with the files produced*.

81.6 Installation and Configuration Improvements

Perl 5.8.5 should build cleanly from source on LynxOS.

81.7 Selected Bug Fixes

- The in-place sort optimisation introduced in 5.8.4 had a bug. For example, in code such as

  ```
  @a = sort ($b, @a)
  ```

  the result would omit the value $b. This is now fixed.
- The optimisation for unnecessary assignments introduced in 5.8.4 could give spurious warnings. This has been fixed.
- Perl should now correctly detect and read BOM-marked and (BOMless) UTF-16 scripts of either endianness.
- Creating a new thread when weak references exist was buggy, and would often cause warnings at interpreter destruction time. The known bug is now fixed.
- Several obscure bugs involving manipulating Unicode strings with `substr` have been fixed.
- Previously if Perl's file globbing function encountered a directory that it did not have permission to open it would return immediately, leading to unexpected truncation of the list of results. This has been fixed, to be consistent with Unix shells' globbing behaviour.
- Thread creation time could vary wildly between identical runs. This was caused by a poor hashing algorithm in the thread cloning routines, which has now been fixed.
- The internals of the ithreads implementation were not checking if OS-level thread creation had failed. threads->create() now returns `undef` in if thead creation fails instead of crashing perl.

81.8 New or Changed Diagnostics

- Perl -V has several improvements
  - correctly outputs local patch names that contain embedded code snippets or other characters that used to confuse it.
  - arguments to -V that look like regexps will give multiple lines of output.
  - a trailing colon suppresses the linefeed and ';' terminator, allowing embedding of queries into shell commands.
  - a leading colon removes the 'name=' part of the response, allowing mapping to any name.
- When perl fails to find the specified script, it now outputs a second line suggesting that the user use the `-S` flag:

```
$ perl5.8.5 missing.pl
Can't open perl script "missing.pl": No such file or directory.
Use -S to search $PATH for it.
```

81.9 Changed Internals

The Unicode character class files used by the regular expression engine are now built at build time from the supplied Unicode consortium data files, instead of being shipped prebuilt. This makes the compressed Perl source tarball about 200K smaller. A side effect is that the layout of files inside lib/unicore has changed.

81.10 Known Problems

The regression test *t/uni/class.t* is now performing considerably more tests, and can take several minutes to run even on a fast machine.

81.11 Platform Specific Problems

This release is known not to build on Windows 95.

81.12 Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup and the perl bug database at http://bugs.perl.org. There may also be information at http://www.perl.org, the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to perlbug@perl.org to be analysed by the Perl porting team. You can browse and search the Perl 5 bugs at http://bugs.perl.org/

81.13 SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

Chapter 82

perl584delta

What is new for perl v5.8.4

82.1 DESCRIPTION

This document describes differences between the 5.8.3 release and the 5.8.4 release.

82.2 Incompatible Changes

Many minor bugs have been fixed. Scripts which happen to rely on previously erroneous behaviour will consider these fixes as incompatible changes :-) You are advised to perform sufficient acceptance testing on this release to satisfy yourself that this does not affect you, before putting this release into production.

The diagnostic output of Carp has been changed slightly, to add a space after the comma between arguments. This makes it much easier for tools such as web browsers to wrap it, but might confuse any automatic tools which perform detailed parsing of Carp output.

The internal dump output has been improved, so that non-printable characters such as newline and backspace are output in `\x` notation, rather than octal. This might just confuse non-robust tools which parse the output of modules such as Devel::Peek.

82.3 Core Enhancements

82.3.1 Malloc wrapping

Perl can now be built to detect attempts to assign pathologically large chunks of memory. Previously such assignments would suffer from integer wrap-around during size calculations causing a misallocation, which would crash perl, and could theoretically be used for "stack smashing" attacks. The wrapping defaults to enabled on platforms where we know it works (most AIX configurations, BSDi, Darwin, DEC OSF/1, FreeBSD, HP/UX, GNU Linux, OpenBSD, Solaris, VMS and most Win32 compilers) and defaults to disabled on other platforms.

82.3.2 Unicode Character Database 4.0.1

The copy of the Unicode Character Database included in Perl 5.8 has been updated to 4.0.1 from 4.0.0.

82.3.3 suidperl less insecure

Paul Szabo has analysed and patched `suidperl` to remove existing known insecurities. Currently there are no known holes in `suidperl`, but previous experience shows that we cannot be confident that these were the last. You may no longer invoke the set uid perl directly, so to preserve backwards compatibility with scripts that invoke #!/usr/bin/suidperl the only set uid binary is now `sperl5.8.`*n* (`sperl5.8.4` for this release). `suidperl` is installed as a hard link to `perl`; both `suidperl` and `perl` will invoke `sperl5.8.4` automatically the set uid binary, so this change should be completely transparent.

For new projects the core perl team would strongly recommend that you use dedicated, single purpose security tools such as `sudo` in preference to `suidperl`.

82.3.4 format

In addition to bug fixes, `format`'s features have been enhanced. See *perlform*

82.4 Modules and Pragmata

The (mis)use of `/tmp` in core modules and documentation has been tidied up. Some modules available both within the perl core and independently from CPAN ("dual-life modules") have not yet had these changes applied; the changes will be integrated into future stable perl releases as the modules are updated on CPAN.

82.4.1 Updated modules

Attribute::Handlers

B

Benchmark

CGI

Carp

Cwd

Exporter

File::Find

IO

IPC::Open3

Local::Maketext

Math::BigFloat

Math::BigInt

Math::BigRat

MIME::Base64

ODBM_File

POSIX

Shell

Socket

There is experimental support for Linux abstract Unix domain sockets.

Storable

Switch

Synced with its CPAN version 2.10

Sys::Syslog

`syslog()` can now use numeric constants for facility names and priorities, in addition to strings.

Term::ANSIColor

Time::HiRes

Unicode::UCD

Win32

Win32.pm/Win32.xs has moved from the libwin32 module to core Perl

base

open

threads

Detached threads are now also supported on Windows.

utf8

82.5 Performance Enhancements

- Accelerated Unicode case mappings (`/i`, `lc`, `uc`, etc).
- In place sort optimised (eg `@a = sort @a`)
- Unnecessary assignment optimised away in

```
my $s = undef;
my @a = ();
my %h = ();
```

- Optimised `map` in scalar context

82.6 Utility Changes

The Perl debugger (*lib/perl5db.pl*) can now save all debugger commands for sourcing later, and can display the parent inheritance tree of a given class.

82.7 Installation and Configuration Improvements

The build process on both VMS and Windows has had several minor improvements made. On Windows Borland's C compiler can now compile perl with PerlIO and/or USE_LARGE_FILES enabled.

`perl.exe` on Windows now has a "Camel" logo icon. The use of a camel with the topic of Perl is a trademark of O'Reilly and Associates Inc., and is used with their permission (ie distribution of the source, compiling a Windows executable from it, and using that executable locally). Use of the supplied camel for anything other than a perl executable's icon is specifically not covered, and anyone wishing to redistribute perl binaries *with* the icon should check directly with O'Reilly beforehand.

Perl should build cleanly on Stratus VOS once more.

82.8 Selected Bug Fixes

More utf8 bugs fixed, notably in how `chomp`, `chop`, `send`, and `syswrite` and interact with utf8 data. Concatenation now works correctly when `use bytes;` is in scope.

Pragmata are now correctly propagated into (?{...}) constructions in regexps. Code such as

```
   my $x = qr{ ... (??{ $x }) ... };
```

will now (correctly) fail under use strict. (As the inner `$x` is and has always referred to `$::x`)

The "const in void context" warning has been suppressed for a constant in an optimised-away boolean expression such as `5 || print;`

`perl -i` could `fchmod(stdin)` by mistake. This is serious if stdin is attached to a terminal, and perl is running as root. Now fixed.

82.9 New or Changed Diagnostics

`Carp` and the internal diagnostic routines used by `Devel::Peek` have been made clearer, as described in Incompatible Changes

82.10 Changed Internals

Some bugs have been fixed in the hash internals. Restricted hashes and their place holders are now allocated and deleted at slightly different times, but this should not be visible to user code.

82.11 Future Directions

Code freeze for the next maintenance release (5.8.5) will be on 30th June 2004, with release by mid July.

82.12 Platform Specific Problems

This release is known not to build on Windows 95.

82.13 Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup and the perl bug database at http://bugs.perl.org. There may also be information at http://www.perl.org, the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to perlbug@perl.org to be analysed by the Perl porting team. You can browse and search the Perl 5 bugs at http://bugs.perl.org/

82.14 SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

Chapter 83

perl583delta

What is new for perl v5.8.3

83.1 DESCRIPTION

This document describes differences between the 5.8.2 release and the 5.8.3 release.

If you are upgrading from an earlier release such as 5.6.1, first read the *perl58delta*, which describes differences between 5.6.0 and 5.8.0, and the *perl581delta* and *perl582delta*, which describe differences between 5.8.0, 5.8.1 and 5.8.2

83.2 Incompatible Changes

There are no changes incompatible with 5.8.2.

83.3 Core Enhancements

A `SCALAR` method is now available for tied hashes. This is called when a tied hash is used in scalar context, such as

```
    if (%tied_hash) {
        ...
    }
```

The old behaviour was that %tied_hash would return whatever would have been returned for that hash before the hash was tied (so usually 0). The new behaviour in the absence of a SCALAR method is to return TRUE if in the middle of an `each` iteration, and otherwise call FIRSTKEY to check if the hash is empty (making sure that a subsequent `each` will also begin by calling FIRSTKEY). Please see SCALAR in *perltie* for the full details and caveats.

83.4 Modules and Pragmata

CGI

Cwd

Digest

Digest::MD5

Encode

File::Spec

FindBin

A function `again` is provided to resolve problems where modules in different directories wish to use FindBin.

List::Util

You can now weaken references to read only values.

Math::BigInt

PodParser

Pod::Perldoc

POSIX

Unicode::Collate

Unicode::Normalize

Test::Harness

threads::shared

`cond_wait` has a new two argument form. `cond_timedwait` has been added.

83.5 Utility Changes

`find2perl` now assumes `-print` as a default action. Previously, it needed to be specified explicitly.

A new utility, `prove`, makes it easy to run an individual regression test at the command line. `prove` is part of Test::Harness, which users of earlier Perl versions can install from CPAN.

83.6 New Documentation

The documentation has been revised in places to produce more standard manpages.

The documentation for the special code blocks (BEGIN, CHECK, INIT, END) has been improved.

83.7 Installation and Configuration Improvements

Perl now builds on OpenVMS I64

83.8 Selected Bug Fixes

Using substr() on a UTF8 string could cause subsequent accesses on that string to return garbage. This was due to incorrect UTF8 offsets being cached, and is now fixed.

join() could return garbage when the same join() statement was used to process 8 bit data having earlier processed UTF8 data, due to the flags on that statement's temporary workspace not being reset correctly. This is now fixed.

`$a .. $b` will now work as expected when either $a or $b is `undef`

Using Unicode keys with tied hashes should now work correctly.

Reading $ˆE now preserves $!. Previously, the C code implementing $ˆE did not preserve `errno`, so reading $ˆE could cause `errno` and therefore `$!` to change unexpectedly.

Reentrant functions will (once more) work with C++. 5.8.2 introduced a bugfix which accidentally broke the compilation of Perl extensions written in C++

83.9 New or Changed Diagnostics

The fatal error "DESTROY created new reference to dead object" is now documented in *perldiag*.

83.10 Changed Internals

The hash code has been refactored to reduce source duplication. The external interface is unchanged, and aside from the bug fixes described above, there should be no change in behaviour.

`hv_clear_placeholders` is now part of the perl API

Some C macros have been tidied. In particular macros which create temporary local variables now name these variables more defensively, which should avoid bugs where names clash.

<signal.h> is now always included.

83.11 Configuration and Building

`Configure` now invokes callbacks regardless of the value of the variable they are called for. Previously callbacks were only invoked in the `case $variable $define)` branch. This change should only affect platform maintainers writing configuration hints files.

83.12 Platform Specific Problems

The regression test ext/threads/shared/t/wait.t fails on early RedHat 9 and HP-UX 10.20 due to bugs in their threading implementations. RedHat users should see https://rhn.redhat.com/errata/RHBA-2003-136.html and consider upgrading their glibc.

83.13 Known Problems

Detached threads aren't supported on Windows yet, as they may lead to memory access violation problems.

There is a known race condition opening scripts in `suidperl`. `suidperl` is neither built nor installed by default, and has been deprecated since perl 5.8.0. You are advised to replace use of suidperl with tools such as sudo (http://www.courtesan.com/sudo/)

We have a backlog of unresolved bugs. Dealing with bugs and bug reports is unglamorous work; not something ideally suited to volunteer labour, but that is all that we have.

The perl5 development team are implementing changes to help address this problem, which should go live in early 2004.

83.14 Future Directions

Code freeze for the next maintenance release (5.8.4) is on March 31st 2004, with release expected by mid April. Similarly 5.8.5's freeze will be at the end of June, with release by mid July.

83.15 Obituary

Iain 'Spoon' Truskett, Perl hacker, author of *perlreref* and contributor to CPAN, died suddenly on 29th December 2003, aged 24. He will be missed.

83.16 Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup and the perl bug database at http://bugs.perl.org. There may also be information at http://www.perl.org, the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to perlbug@perl.org to be analysed by the Perl porting team. You can browse and search the Perl 5 bugs at http://bugs.perl.org/

83.17 SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

Chapter 84

perl582delta

What is new for perl v5.8.2

84.1 DESCRIPTION

This document describes differences between the 5.8.1 release and the 5.8.2 release.

If you are upgrading from an earlier release such as 5.6.1, first read the *perl58delta*, which describes differences between 5.6.0 and 5.8.0, and the *perl581delta*, which describes differences between 5.8.0 and 5.8.1.

84.2 Incompatible Changes

For threaded builds for modules calling certain re-entrant system calls, binary compatibility was accidentally lost between 5.8.0 and 5.8.1. Binary compatibility with 5.8.0 has been restored in 5.8.2, which necessitates breaking compatibility with 5.8.1. We see this as the lesser of two evils.

This will only affect people who have a threaded perl 5.8.1, and compiled modules which use these calls, and now attempt to run the compiled modules with 5.8.2. The fix is to re-compile and re-install the modules using 5.8.2.

84.3 Core Enhancements

84.3.1 Hash Randomisation

The hash randomisation introduced with 5.8.1 has been amended. It transpired that although the implementation introduced in 5.8.1 was source compatible with 5.8.0, it was not binary compatible in certain cases. 5.8.2 contains an improved implementation which is both source and binary compatible with both 5.8.0 and 5.8.1, and remains robust against the form of attack which prompted the change for 5.8.1.

We are grateful to the Debian project for their input in this area. See Algorithmic Complexity Attacks in *perlsec* for the original rationale behind this change.

84.3.2 Threading

Several memory leaks associated with variables shared between threads have been fixed.

84.4 Modules and Pragmata

84.4.1 Updated Modules And Pragmata

The following modules and pragmata have been updated since Perl 5.8.1:

Devel::PPPort

Digest::MD5

I18N::LangTags

libnet

MIME::Base64

Pod::Perldoc

strict

Documentation improved

Tie::Hash

Documentation improved

Time::HiRes

Unicode::Collate

Unicode::Normalize

UNIVERSAL

Documentation improved

84.5 Selected Bug Fixes

Some syntax errors involving unrecognized filetest operators are now handled correctly by the parser.

84.6 Changed Internals

Interpreter initialization is more complete when -DMULTIPLICITY is off. This should resolve problems with initializing and destroying the Perl interpreter more than once in a single process.

84.7 Platform Specific Problems

Dynamic linker flags have been tweaked for Solaris and OS X, which should solve problems seen while building some XS modules.

Bugs in OS/2 sockets and tmpfile have been fixed.

In OS X `setreuid` and friends are troublesome - perl will now work around their problems as best possible.

84.8 Future Directions

Starting with 5.8.3 we intend to make more frequent maintenance releases, with a smaller number of changes in each. The intent is to propagate bug fixes out to stable releases more rapidly and make upgrading stable releases less of an upheaval. This should give end users more flexibility in their choice of upgrade timing, and allow them easier assessment of the impact of upgrades. The current plan is for code freezes as follows

- 5.8.3 23:59:59 GMT, Wednesday December 31st 2003
- 5.8.4 23:59:59 GMT, Wednesday March 31st 2004
- 5.8.5 23:59:59 GMT, Wednesday June 30th 2004

with the release following soon after, when testing is complete.

See Future Directions in *perl581delta* for more soothsaying.

84.9 Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup and the perl bug database at http://bugs.perl.org/. There may also be information at http://www.perl.com/, the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to perlbug@perl.org to be analysed by the Perl porting team. You can browse and search the Perl 5 bugs at http://bugs.perl.org/

84.10 SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

Chapter 85

perl581delta

What is new for perl v5.8.1

85.1 DESCRIPTION

This document describes differences between the 5.8.0 release and the 5.8.1 release.

If you are upgrading from an earlier release such as 5.6.1, first read the *perl58delta*, which describes differences between 5.6.0 and 5.8.0.

In case you are wondering about 5.6.1, it was bug-fix-wise rather identical to the development release 5.7.1. Confused? This timeline hopefully helps a bit: it lists the new major releases, their maintenance releases, and the development releases.

```
          New     Maintenance  Development

          5.6.0                             2000-Mar-22
                               5.7.0        2000-Sep-02
                  5.6.1                     2001-Apr-08
                               5.7.1        2001-Apr-09
                               5.7.2        2001-Jul-13
                               5.7.3        2002-Mar-05
          5.8.0                             2002-Jul-18
                  5.8.1                     2003-Sep-25
```

85.2 Incompatible Changes

85.2.1 Hash Randomisation

Mainly due to security reasons, the "random ordering" of hashes has been made even more random. Previously while the order of hash elements from keys(), values(), and each() was essentially random, it was still repeatable. Now, however, the order varies between different runs of Perl.

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.

The added randomness may affect applications.

One possible scenario is when output of an application has included hash data. For example, if you have used the Data::Dumper module to dump data into different files, and then compared the files to see whether the data has changed, now you will have false positives since the order in which hashes are dumped will vary. In general the cure is to sort the keys (or the values); in particular for Data::Dumper to use the `Sortkeys` option. If some particular order is really important, use tied hashes: for example the Tie::IxHash module which by default preserves the order in which the hash elements were added.

More subtle problem is reliance on the order of "global destruction". That is what happens at the end of execution: Perl destroys all data structures, including user data. If your destructors (the DESTROY subroutines) have assumed any particular ordering to the global destruction, there might be problems ahead. For example, in a destructor of one object you cannot assume that objects of any other class are still available, unless you hold a reference to them. If the environment variable PERL_DESTRUCT_LEVEL is set to a non-zero value, or if Perl is exiting a spawned thread, it will also destruct the ordinary references and the symbol tables that are no longer in use. You can't call a class method or an ordinary function on a class that has been collected that way.

The hash randomisation is certain to reveal hidden assumptions about some particular ordering of hash elements, and outright bugs: it revealed a few bugs in the Perl core and core modules.

To disable the hash randomisation in runtime, set the environment variable PERL_HASH_SEED to 0 (zero) before running Perl (for more information see PERL_HASH_SEED in *perlrun*), or to disable the feature completely in compile time, compile with `-DNO_HASH_SEED` (see *INSTALL*).

See Algorithmic Complexity Attacks in *perlsec* for the original rationale behind this change.

85.2.2 UTF-8 On Filehandles No Longer Activated By Locale

In Perl 5.8.0 all filehandles, including the standard filehandles, were implicitly set to be in Unicode UTF-8 if the locale settings indicated the use of UTF-8. This feature caused too many problems, so the feature was turned off and redesigned: see §85.3.

85.2.3 Single-number v-strings are no longer v-strings before "=>"

The version strings or v-strings (see Version Strings in *perldata*) feature introduced in Perl 5.6.0 has been a source of some confusion– especially when the user did not want to use it, but Perl thought it knew better. Especially troublesome has been the feature that before a "=>" a version string (a "v" followed by digits) has been interpreted as a v-string instead of a string literal. In other words:

```
    %h = ( v65 => 42 );
```

has meant since Perl 5.6.0

```
    %h = ( 'A' => 42 );
```

(at least in platforms of ASCII progeny) Perl 5.8.1 restores the more natural interpretation

```
    %h = ( 'v65' => 42 );
```

The multi-number v-strings like v65.66 and 65.66.67 still continue to be v-strings in Perl 5.8.

85.2.4 (Win32) The -C Switch Has Been Repurposed

The -C switch has changed in an incompatible way. The old semantics of this switch only made sense in Win32 and only in the "use utf8" universe in 5.6.x releases, and do not make sense for the Unicode implementation in 5.8.0. Since this switch could not have been used by anyone, it has been repurposed. The behavior that this switch enabled in 5.6.x releases may be supported in a transparent, data-dependent fashion in a future release.

For the new life of this switch, see §85.3.1, and -C in *perlrun*.

85.2.5 (Win32) The /d Switch Of cmd.exe

Perl 5.8.1 uses the /d switch when running the cmd.exe shell internally for system(), backticks, and when opening pipes to external programs. The extra switch disables the execution of AutoRun commands from the registry, which is generally considered undesirable when running external programs. If you wish to retain compatibility with the older behavior, set PERL5SHELL in your environment to `cmd /x/c`.

85.3 Core Enhancements

85.3.1 UTF-8 no longer default under UTF-8 locales

In Perl 5.8.0 many Unicode features were introduced. One of them was found to be of more nuisance than benefit: the automagic (and silent) "UTF-8-ification" of filehandles, including the standard filehandles, if the user's locale settings indicated use of UTF-8.

For example, if you had `en_US.UTF-8` as your locale, your STDIN and STDOUT were automatically "UTF-8", in other words an implicit binmode(..., ":utf8") was made. This meant that trying to print, say, chr(0xff), ended up printing the bytes 0xc3 0xbf. Hardly what you had in mind unless you were aware of this feature of Perl 5.8.0. The problem is that the vast majority of people weren't: for example in RedHat releases 8 and 9 the **default** locale setting is UTF-8, so all RedHat users got UTF-8 filehandles, whether they wanted it or not. The pain was intensified by the Unicode implementation of Perl 5.8.0 (still) having nasty bugs, especially related to the use of s/// and tr///. (Bugs that have been fixed in 5.8.1)

Therefore a decision was made to backtrack the feature and change it from implicit silent default to explicit conscious option. The new Perl command line option `-C` and its counterpart environment variable PERL_UNICODE can now be used to control how Perl and Unicode interact at interfaces like I/O and for example the command line arguments. See -C in *perlrun* and PERL_UNICODE in *perlrun* for more information.

85.3.2 Unsafe signals again available

In Perl 5.8.0 the so-called "safe signals" were introduced. This means that Perl no longer handles signals immediately but instead "between opcodes", when it is safe to do so. The earlier immediate handling easily could corrupt the internal state of Perl, resulting in mysterious crashes.

However, the new safer model has its problems too. Because now an opcode, a basic unit of Perl execution, is never interrupted but instead let to run to completion, certain operations that can take a long time now really do take a long time. For example, certain network operations have their own blocking and timeout mechanisms, and being able to interrupt them immediately would be nice.

Therefore perl 5.8.1 introduces a "backdoor" to restore the pre-5.8.0 (pre-5.7.3, really) signal behaviour. Just set the environment variable PERL_SIGNALS to `unsafe`, and the old immediate (and unsafe) signal handling behaviour returns. See PERL_SIGNALS in *perlrun* and Deferred Signals (Safe Signals) in *perlipc*.

In completely unrelated news, you can now use safe signals with POSIX::SigAction. See POSIX::SigAction in *POSIX*.

85.3.3 Tied Arrays with Negative Array Indices

Formerly, the indices passed to `FETCH`, `STORE`, `EXISTS`, and `DELETE` methods in tied array class were always non-negative. If the actual argument was negative, Perl would call FETCHSIZE implicitly and add the result to the index before passing the result to the tied array method. This behaviour is now optional. If the tied array class contains a package variable named `$NEGATIVE_INDICES` which is set to a true value, negative values will be passed to `FETCH`, `STORE`, `EXISTS`, and `DELETE` unchanged.

85.3.4 local $ {$ x}

The syntaxes

```
        local ${$x}
        local @{$x}
        local %{$x}
```

now do localise variables, given that the $x is a valid variable name.

85.3.5 Unicode Character Database 4.0.0

The copy of the Unicode Character Database included in Perl 5.8 has been updated to 4.0.0 from 3.2.0. This means for example that the Unicode character properties are as in Unicode 4.0.0.

85.3.6 Deprecation Warnings

There is one new feature deprecation. Perl 5.8.0 forgot to add some deprecation warnings, these warnings have now been added. Finally, a reminder of an impending feature removal.

(Reminder) Pseudo-hashes are deprecated (really)

Pseudo-hashes were deprecated in Perl 5.8.0 and will be removed in Perl 5.10.0, see *perl58delta* for details. Each attempt to access pseudo-hashes will trigger the warning `Pseudo-hashes are deprecated`. If you really want to continue using pseudo-hashes but not to see the deprecation warnings, use:

```
    no warnings 'deprecated';
```

Or you can continue to use the *fields* pragma, but please don't expect the data structures to be pseudohashes any more.

(Reminder) 5.005-style threads are deprecated (really)

5.005-style threads (activated by `use Thread;`) were deprecated in Perl 5.8.0 and will be removed after Perl 5.8, see *perl58delta* for details. Each 5.005-style thread creation will trigger the warning `5.005 threads are deprecated`. If you really want to continue using the 5.005 threads but not to see the deprecation warnings, use:

```
    no warnings 'deprecated';
```

(Reminder) The $ * variable is deprecated (really)

The `$*` variable controlling multi-line matching has been deprecated and will be removed after 5.8. The variable has been deprecated for a long time, and a deprecation warning `Use of $* is deprecated` is given, now the variable will just finally be removed. The functionality has been supplanted by the `/s` and `/m` modifiers on pattern matching. If you really want to continue using the `$*`-variable but not to see the deprecation warnings, use:

```
    no warnings 'deprecated';
```

85.3.7 Miscellaneous Enhancements

`map` in void context is no longer expensive. `map` is now context aware, and will not construct a list if called in void context.

If a socket gets closed by the server while printing to it, the client now gets a SIGPIPE. While this new feature was not planned, it fell naturally out of PerlIO changes, and is to be considered an accidental feature.

PerlIO::get_layers(FH) returns the names of the PerlIO layers active on a filehandle.

PerlIO::via layers can now have an optional UTF8 method to indicate whether the layer wants to "auto-:utf8" the stream.

utf8::is_utf8() has been added as a quick way to test whether a scalar is encoded internally in UTF-8 (Unicode).

85.4 Modules and Pragmata

85.4.1 Updated Modules And Pragmata

The following modules and pragmata have been updated since Perl 5.8.0:

base

B::Bytecode

In much better shape than it used to be. Still far from perfect, but maybe worth a try.

B::Concise

B::Deparse

Benchmark

An optional feature, `:hireswallclock`, now allows for high resolution wall clock times (uses Time::HiRes).

ByteLoader

See B::Bytecode.

bytes

Now has bytes::substr.

CGI

charnames

One can now have custom character name aliases.

CPAN

There is now a simple command line frontend to the CPAN.pm module called *cpan*.

Data::Dumper

A new option, Pair, allows choosing the separator between hash keys and values.

DB_File

Devel::PPPort

Digest::MD5

Encode

Significant updates on the encoding pragma functionality (tr/// and the DATA filehandle, formats).

If a filehandle has been marked as to have an encoding, unmappable characters are detected already during input, not later (when the corrupted data is being used).

The ISO 8859-6 conversion table has been corrected (the 0x30..0x39 erroneously mapped to U+0660..U+0669, instead of U+0030..U+0039). The GSM 03.38 conversion did not handle escape sequences correctly. The UTF-7 encoding has been added (making Encode feature-complete with Unicode::String).

fields

libnet

Math::BigInt

A lot of bugs have been fixed since v1.60, the version included in Perl v5.8.0. Especially noteworthy are the bug in Calc that caused div and mod to fail for some large values, and the fixes to the handling of bad inputs.

Some new features were added, e.g. the broot() method, you can now pass parameters to config() to change some settings at runtime, and it is now possible to trap the creation of NaN and infinity.

As usual, some optimizations took place and made the math overall a tad faster. In some cases, quite a lot faster, actually. Especially alternative libraries like Math::BigInt::GMP benefit from this. In addition, a lot of the quite clunky routines like fsqrt() and flog() are now much much faster.

MIME::Base64

NEXT

Diamond inheritance now works.

Net::Ping

PerlIO::scalar

Reading from non-string scalars (like the special variables, see *perlvar*) now works.

podlators

Pod::LaTeX

PodParsers

Pod::Perldoc

Complete rewrite. As a side-effect, no longer refuses to startup when run by root.

Scalar::Util

New utilities: refaddr, isvstring, looks_like_number, set_prototype.

Storable

Can now store code references (via B::Deparse, so not foolproof).

strict

Earlier versions of the strict pragma did not check the parameters implicitly passed to its "import" (use) and "unimport" (no) routine. This caused the false idiom such as:

```
    use strict qw(@ISA);
    @ISA = qw(Foo);
```

This however (probably) raised the false expectation that the strict refs, vars and subs were being enforced (and that @ISA was somehow "declared"). But the strict refs, vars, and subs are **not** enforced when using this false idiom.

Starting from Perl 5.8.1, the above **will** cause an error to be raised. This may cause programs which used to execute seemingly correctly without warnings and errors to fail when run under 5.8.1. This happens because

```
    use strict qw(@ISA);
```

will now fail with the error:

```
    Unknown 'strict' tag(s) '@ISA'
```

The remedy to this problem is to replace this code with the correct idiom:

```
    use strict;
    use vars qw(@ISA);
    @ISA = qw(Foo);
```

Term::ANSIcolor

Test::Harness

Now much more picky about extra or missing output from test scripts.

Test::More

Test::Simple

Text::Balanced

Time::HiRes

Use of nanosleep(), if available, allows mixing subsecond sleeps with alarms.

threads

Several fixes, for example for join() problems and memory leaks. In some platforms (like Linux) that use glibc the minimum memory footprint of one ithread has been reduced by several hundred kilobytes.

threads::shared

Many memory leaks have been fixed.

Unicode::Collate

Unicode::Normalize

Win32::GetFolderPath

Win32::GetOSVersion

Now returns extra information.

85.5 Utility Changes

The `h2xs` utility now produces a more modern layout: *Foo-Bar/lib/Foo/Bar.pm* instead of *Foo/Bar/Bar.pm*. Also, the boilerplate test is now called *t/Foo-Bar.t* instead of *t/1.t*.

The Perl debugger (*lib/perl5db.pl*) has now been extensively documented and bugs found while documenting have been fixed.

`perldoc` has been rewritten from scratch to be more robust and featureful.

`perlcc -B` works now at least somewhat better, while `perlcc -c` is rather more broken. (The Perl compiler suite as a whole continues to be experimental.)

85.6 New Documentation

perl573delta has been added to list the differences between the (now quite obsolete) development releases 5.7.2 and 5.7.3.

perl58delta has been added: it is the perldelta of 5.8.0, detailing the differences between 5.6.0 and 5.8.0.

perlartistic has been added: it is the Artistic License in pod format, making it easier for modules to refer to it.

perlcheat has been added: it is a Perl cheat sheet.

perlgpl has been added: it is the GNU General Public License in pod format, making it easier for modules to refer to it.

perlmacosx has been added to tell about the installation and use of Perl in Mac OS X.

perlos400 has been added to tell about the installation and use of Perl in OS/400 PASE.

perlreref has been added: it is a regular expressions quick reference.

85.7 Installation and Configuration Improvements

The UNIX standard Perl location, */usr/bin/perl*, is no longer overwritten by default if it exists. This change was very prudent because so many UNIX vendors already provide a */usr/bin/perl*, but simultaneously many system utilities may depend on that exact version of Perl, so better not to overwrite it.

One can now specify installation directories for site and vendor man and HTML pages, and site and vendor scripts. See *INSTALL*.

One can now specify a destination directory for Perl installation by specifying the DESTDIR variable for `make install`. (This feature is slightly different from the previous `Configure -Dinstallprefix=...`.) See *INSTALL*.

gcc versions 3.x introduced a new warning that caused a lot of noise during Perl compilation: `gcc -Ialreadyknowndirectory (warning:  changing search order)`. This warning has now been avoided by Configure weeding out such directories before the compilation.

One can now build subsets of Perl core modules by using the Configure flags `-Dnoextensions=...` and `-Donlyextensions=...`, see *INSTALL*.

85.7.1 Platform-specific enhancements

In Cygwin Perl can now be built with threads (`Configure -Duseithreads`). This works with both Cygwin 1.3.22 and Cygwin 1.5.3.

In newer FreeBSD releases Perl 5.8.0 compilation failed because of trying to use *malloc.h*, which in FreeBSD is just a dummy file, and a fatal error to even try to use. Now *malloc.h* is not used.

Perl is now known to build also in Hitachi HI-UXMPP.

Perl is now known to build again in LynxOS.

Mac OS X now installs with Perl version number embedded in installation directory names for easier upgrading of user-compiled Perl, and the installation directories in general are more standard. In other words, the default installation no longer breaks the Apple-provided Perl. On the other hand, with `Configure -Dprefix=/usr` you can now really replace the Apple-supplied Perl (**please be careful**).

Mac OS X now builds Perl statically by default. This change was done mainly for faster startup times. The Apple-provided Perl is still dynamically linked and shared, and you can enable the sharedness for your own Perl builds by `Configure -Duseshrplib`.

Perl has been ported to IBM's OS/400 PASE environment. The best way to build a Perl for PASE is to use an AIX host as a cross-compilation environment. See README.os400.

Yet another cross-compilation option has been added: now Perl builds on OpenZaurus, an Linux distribution based on Mandrake + Embedix for the Sharp Zaurus PDA. See the Cross/README file.

Tru64 when using gcc 3 drops the optimisation for *toke.c* to `-O2` because of gigantic memory use with the default `-O3`.

Tru64 can now build Perl with the newer Berkeley DBs.

Building Perl on WinCE has been much enhanced, see *README.ce* and *README.perlce*.

85.8 Selected Bug Fixes

85.8.1 Closures, eval and lexicals

There have been many fixes in the area of anonymous subs, lexicals and closures. Although this means that Perl is now more "correct", it is possible that some existing code will break that happens to rely on the faulty behaviour. In practice this is unlikely unless your code contains a very complex nesting of anonymous subs, evals and lexicals.

85.8.2 Generic fixes

If an input filehandle is marked `:utf8` and Perl sees illegal UTF-8 coming in when doing `<FH>`, if warnings are enabled a warning is immediately given - instead of being silent about it and Perl being unhappy about the broken data later. (The `:encoding(utf8)` layer also works the same way.)

binmode(SOCKET, ":utf8") only worked on the input side, not on the output side of the socket. Now it works both ways.

For threaded Perls certain system database functions like getpwent() and getgrent() now grow their result buffer dynamically, instead of failing. This means that at sites with lots of users and groups the functions no longer fail by returning only partial results.

Perl 5.8.0 had accidentally broken the capability for users to define their own uppercase<->lowercase Unicode mappings (as advertised by the Camel). This feature has been fixed and is also documented better.

In 5.8.0 this

```
    $some_unicode .= <FH>;
```

didn't work correctly but instead corrupted the data. This has now been fixed.

Tied methods like FETCH etc. may now safely access tied values, i.e. resulting in a recursive call to FETCH etc. Remember to break the recursion, though.

At startup Perl blocks the SIGFPE signal away since there isn't much Perl can do about it. Previously this blocking was in effect also for programs executed from within Perl. Now Perl restores the original SIGFPE handling routine, whatever it was, before running external programs.

Linenumbers in Perl scripts may now be greater than 65536, or 2**16. (Perl scripts have always been able to be larger than that, it's just that the linenumber for reported errors and warnings have "wrapped around".) While scripts that large usually indicate a need to rethink your code a bit, such Perl scripts do exist, for example as results from generated code. Now linenumbers can go all the way to 4294967296, or 2**32.

85.8.3 Platform-specific fixes

Linux

- Setting $0 works again (with certain limitations that Perl cannot do much about: see `$0` in *perlvar*)

HP-UX

- Setting $0 now works.

VMS

- Configuration now tests for the presence of `poll()`, and IO::Poll now uses the vendor-supplied function if detected.
- A rare access violation at Perl start-up could occur if the Perl image was installed with privileges or if there was an identifier with the subsystem attribute set in the process's rightslist. Either of these circumstances triggered tainting code that contained a pointer bug. The faulty pointer arithmetic has been fixed.
- The length limit on values (not keys) in the %ENV hash has been raised from 255 bytes to 32640 bytes (except when the PERL_ENV_TABLES setting overrides the default use of logical names for %ENV). If it is necessary to access these long values from outside Perl, be aware that they are implemented using search list logical names that store the value in pieces, each 255-byte piece (up to 128 of them) being an element in the search list. When doing a lookup in %ENV from within Perl, the elements are combined into a single value. The existing VMS-specific ability to access individual elements of a search list logical name via the $ENV{'foo;N'} syntax (where N is the search list index) is unimpaired.
- The piping implementation now uses local rather than global DCL symbols for inter-process communication.

- File::Find could become confused when navigating to a relative directory whose name collided with a logical name. This problem has been corrected by adding directory syntax to relative path names, thus preventing logical name translation.

Win32

- A memory leak in the fork() emulation has been fixed.
- The return value of the ioctl() built-in function was accidentally broken in 5.8.0. This has been corrected.
- The internal message loop executed by perl during blocking operations sometimes interfered with messages that were external to Perl. This often resulted in blocking operations terminating prematurely or returning incorrect results, when Perl was executing under environments that could generate Windows messages. This has been corrected.
- Pipes and sockets are now automatically in binary mode.
- The four-argument form of select() did not preserve $! (errno) properly when there were errors in the underlying call. This is now fixed.
- The "CR CR LF" problem of has been fixed, binmode(FH, ":crlf") is now effectively a no-op.

85.9 New or Changed Diagnostics

All the warnings related to pack() and unpack() were made more informative and consistent.

85.9.1 Changed "A thread exited while %d threads were running"

The old version

```
    A thread exited while %d other threads were still running
```

was misleading because the "other" included also the thread giving the warning.

85.9.2 Removed "Attempt to clear a restricted hash"

It is not illegal to clear a restricted hash, so the warning was removed.

85.9.3 New "Illegal declaration of anonymous subroutine"

You must specify the block of code for `sub`.

85.9.4 Changed "Invalid range "%s" in transliteration operator"

The old version

```
    Invalid [] range "%s" in transliteration operator
```

was simply wrong because there are no "[] ranges" in tr///.

85.9.5 New "Missing control char name in \c"

Self-explanatory.

85.9.6 New "Newline in left-justified string for %s"

The padding spaces would appear after the newline, which is probably not what you had in mind.

85.9.7 New "Possible precedence problem on bitwise %c operator"

If you think this

```
    $x & $y == 0
```

tests whether the bitwise AND of $x and $y is zero, you will like this warning.

85.9.8 New "Pseudo-hashes are deprecated"

This warning should have been already in 5.8.0, since they are.

85.9.9 New "read() on %s filehandle %s"

You cannot read() (or sysread()) from a closed or unopened filehandle.

85.9.10 New "5.005 threads are deprecated"

This warning should have been already in 5.8.0, since they are.

85.9.11 New "Tied variable freed while still in use"

Something pulled the plug on a live tied variable, Perl plays safe by bailing out.

85.9.12 New "To%s: illegal mapping '%s'"

An illegal user-defined Unicode casemapping was specified.

85.9.13 New "Use of freed value in iteration"

Something modified the values being iterated over. This is not good.

85.10 Changed Internals

These news matter to you only if you either write XS code or like to know about or hack Perl internals (using Devel::Peek or any of the `B::` modules counts), or like to run Perl with the `-D` option.

The embedding examples of *perlembed* have been reviewed to be uptodate and consistent: for example, the correct use of PERL_SYS_INIT3() and PERL_SYS_TERM().

Extensive reworking of the pad code (the code responsible for lexical variables) has been conducted by Dave Mitchell.

Extensive work on the v-strings by John Peacock.

UTF-8 length and position cache: to speed up the handling of Unicode (UTF-8) scalars, a cache was introduced. Potential problems exist if an extension bypasses the official APIs and directly modifies the PV of an SV: the UTF-8 cache does not get cleared as it should.

APIs obsoleted in Perl 5.8.0, like sv_2pv, sv_catpvn, sv_catsv, sv_setsv, are again available.

Certain Perl core C APIs like cxinc and regatom are no longer available at all to code outside the Perl core of the Perl core extensions. This is intentional. They never should have been available with the shorter names, and if you application depends on them, you should (be ashamed and) contact perl5-porters to discuss what are the proper APIs.

Certain Perl core C APIs like `Perl_list` are no longer available without their `Perl_` prefix. If your XS module stops working because some functions cannot be found, in many cases a simple fix is to add the `Perl_` prefix to the function and the thread context `aTHX_` as the first argument of the function call. This is also how it should always have been done: letting the Perl_-less forms to leak from the core was an accident. For cleaner embedding you can also force this for all APIs by defining at compile time the cpp define PERL_NO_SHORT_NAMES.

Perl_save_bool() has been added.

Regexp objects (those created with `qr`) now have S-magic rather than R-magic. This fixed regexps of the form /...(??{...;$x})/ to no longer ignore changes made to $x. The S-magic avoids dropping the caching optimization and making (??{...}) constructs obscenely slow (and consequently useless). See also Magic Variables in *perlguts*. Regexp::Copy was affected by this change.

The Perl internal debugging macros DEBUG() and DEB() have been renamed to PERL_DEBUG() and PERL_DEB() to avoid namespace conflicts.

`-DL` removed (the leaktest had been broken and unsupported for years, use alternative debugging mallocs or tools like valgrind and Purify).

Verbose modifier `v` added for `-DXv` and `-Dsv`, see *perlrun*.

85.11 New Tests

In Perl 5.8.0 there were about 69000 separate tests in about 700 test files, in Perl 5.8.1 there are about 77000 separate tests in about 780 test files. The exact numbers depend on the Perl configuration and on the operating system platform.

85.12 Known Problems

The hash randomisation mentioned in Incompatible Changes is definitely problematic: it will wake dormant bugs and shake out bad assumptions.

If you want to use mod_perl 2.x with Perl 5.8.1, you will need mod_perl-1.99_10 or higher. Earlier versions of mod_perl 2.x do not work with the randomised hashes. (mod_perl 1.x works fine.) You will also need Apache::Test 1.04 or higher.

Many of the rarer platforms that worked 100% or pretty close to it with perl 5.8.0 have been left a little bit untended since their maintainers have been otherwise busy lately, and therefore there will be more failures on those platforms. Such platforms include Mac OS Classic, IBM z/OS (and other EBCDIC platforms), and NetWare. The most common Perl platforms (Unix and Unix-like, Microsoft platforms, and VMS) have large enough testing and expert population that they are doing well.

85.12.1 Tied hashes in scalar context

Tied hashes do not currently return anything useful in scalar context, for example when used as boolean tests:

```
        if (%tied_hash) { ... }
```

The current nonsensical behaviour is always to return false, regardless of whether the hash is empty or has elements.

The root cause is that there is no interface for the implementors of tied hashes to implement the behaviour of a hash in scalar context.

85.12.2 Net::Ping 450_service and 510_ping_udp failures

The subtests 9 and 18 of lib/Net/Ping/t/450_service.t, and the subtest 2 of lib/Net/Ping/t/510_ping_udp.t might fail if you have an unusual networking setup. For example in the latter case the test is trying to send a UDP ping to the IP address 127.0.0.1.

85.12.3 B::C

The C-generating compiler backend B::C (the frontend being `perlcc -c`) is even more broken than it used to be because of the extensive lexical variable changes. (The good news is that B::Bytecode and ByteLoader are better than they used to be.)

85.13 Platform Specific Problems

85.13.1 EBCDIC Platforms

IBM z/OS and other EBCDIC platforms continue to be problematic regarding Unicode support. Many Unicode tests are skipped when they really should be fixed.

85.13.2 Cygwin 1.5 problems

In Cygwin 1.5 the *io/tell* and *op/sysio* tests have failures for some yet unknown reason. In 1.5.5 the threads tests stress_cv, stress_re, and stress_string are failing unless the environment variable PERLIO is set to "perlio" (which makes also the io/tell failure go away).

Perl 5.8.1 does build and work well with Cygwin 1.3: with (uname -a) `CYGWIN_NT-5.0 ... 1.3.22(0.78/3/2) 2003-03-18 09:20 i686 ...` a 100% "make test" was achieved with `Configure -des -Duseithreads`.

85.13.3 HP-UX: HP cc warnings about sendfile and sendpath

With certain HP C compiler releases (e.g. B.11.11.02) you will get many warnings like this (lines wrapped for easier reading):

```
  cc: "/usr/include/sys/socket.h", line 504: warning 562:
    Redeclaration of "sendfile" with a different storage class specifier:
      "sendfile" will have internal linkage.
  cc: "/usr/include/sys/socket.h", line 505: warning 562:
    Redeclaration of "sendpath" with a different storage class specifier:
      "sendpath" will have internal linkage.
```

The warnings show up both during the build of Perl and during certain lib/ExtUtils tests that invoke the C compiler. The warning, however, is not serious and can be ignored.

85.13.4 IRIX: t/uni/tr_7jis.t falsely failing

The test t/uni/tr_7jis.t is known to report failure under 'make test' or the test harness with certain releases of IRIX (at least IRIX 6.5 and MIPSpro Compilers Version 7.3.1.1m), but if run manually the test fully passes.

85.13.5 Mac OS X: no usemymalloc

The Perl malloc (`-Dusemymalloc`) does not work at all in Mac OS X. This is not that serious, though, since the native malloc works just fine.

85.13.6 Tru64: No threaded builds with GNU cc (gcc)

In the latest Tru64 releases (e.g. v5.1B or later) gcc cannot be used to compile a threaded Perl (-Duseithreads) because the system <`pthread.h`> file doesn't know about gcc.

85.13.7 Win32: sysopen, sysread, syswrite

As of the 5.8.0 release, sysopen()/sysread()/syswrite() do not behave like they used to in 5.6.1 and earlier with respect to "text" mode. These built-ins now always operate in "binary" mode (even if sysopen() was passed the O_TEXT flag, or if binmode() was used on the file handle). Note that this issue should only make a difference for disk files, as sockets and pipes have always been in "binary" mode in the Windows port. As this behavior is currently considered a bug, compatible behavior may be re-introduced in a future release. Until then, the use of sysopen(), sysread() and syswrite() is not supported for "text" mode operations.

85.14 Future Directions

The following things **might** happen in future. The first publicly available releases having these characteristics will be the developer releases Perl 5.9.x, culminating in the Perl 5.10.0 release. These are our best guesses at the moment: we reserve the right to rethink.

- PerlIO will become The Default. Currently (in Perl 5.8.x) the stdio library is still used if Perl thinks it can use certain tricks to make stdio go **really** fast. For future releases our goal is to make PerlIO go even faster.
- A new feature called *assertions* will be available. This means that one can have code called assertions sprinkled in the code: usually they are optimised away, but they can be enabled with the `-A` option.
- A new operator `//` (defined-or) will be available. This means that one will be able to say

  ```
   $a // $b
  ```

  instead of

  ```
  defined $a ? $a : $b
  ```

  and

  ```
  $c //= $d;
  ```

  instead of

  ```
  $c = $d unless defined $c;
  ```

  The operator will have the same precedence and associativity as ||. A source code patch against the Perl 5.8.1 sources will be available in CPAN as *authors/id/H/HM/HMBRAND/dor-5.8.1.diff*.
- `unpack()` will default to unpacking the `$_`.
- Various Copy-On-Write techniques will be investigated in hopes of speeding up Perl.
- CPANPLUS, Inline, and Module::Build will become core modules.
- The ability to write true lexically scoped pragmas will be introduced.
- Work will continue on the bytecompiler and byteloader.
- v-strings as they currently exist are scheduled to be deprecated. The v-less form (1.2.3) will become a "version object" when used with `use`, `require`, and `$VERSION`. $^V will also be a "version object" so the printf("%vd",...) construct will no longer be needed. The v-ful version (v1.2.3) will become obsolete. The equivalence of strings and v-strings (e.g. that currently 5.8.0 is equal to "\5\8\0") will go away. **There may be no deprecation warning for v-strings**, though: it is quite hard to detect when v-strings are being used safely, and when they are not.
- 5.005 Threads Will Be Removed
- The `$*` Variable Will Be Removed (it was deprecated a long time ago)
- Pseudohashes Will Be Removed

85.15 Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup and the perl bug database at http://bugs.perl.org/ . There may also be information at http://www.perl.com/ , the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to perlbug@perl.org to be analysed by the Perl porting team. You can browse and search the Perl 5 bugs at http://bugs.perl.org/

85.16 SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

Chapter 86

perl58delta

What is new for perl v5.8.0

86.1 DESCRIPTION

This document describes differences between the 5.6.0 release and the 5.8.0 release.

Many of the bug fixes in 5.8.0 were already seen in the 5.6.1 maintenance release since the two releases were kept closely coordinated (while 5.8.0 was still called 5.7.something).

Changes that were integrated into the 5.6.1 release are marked `[561]`. Many of these changes have been further developed since 5.6.1 was released, those are marked `[561+]`.

You can see the list of changes in the 5.6.1 release (both from the 5.005_03 release and the 5.6.0 release) by reading *perl561delta*.

86.2 Highlights In 5.8.0

- Better Unicode support
- New IO Implementation
- New Thread Implementation
- Better Numeric Accuracy
- Safe Signals
- Many New Modules
- More Extensive Regression Testing

86.3 Incompatible Changes

86.3.1 Binary Incompatibility

Perl 5.8 is not binary compatible with earlier releases of Perl.

You have to recompile your XS modules.

(Pure Perl modules should continue to work.)

The major reason for the discontinuity is the new IO architecture called PerlIO. PerlIO is the default configuration because without it many new features of Perl 5.8 cannot be used. In other words: you just have to recompile your modules containing XS code, sorry about that.

In future releases of Perl, non-PerlIO aware XS modules may become completely unsupported. This shouldn't be too difficult for module authors, however: PerlIO has been designed as a drop-in replacement (at the source code level) for the stdio interface.

Depending on your platform, there are also other reasons why we decided to break binary compatibility, please read on.

86.3.2 64-bit platforms and malloc

If your pointers are 64 bits wide, the Perl malloc is no longer being used because it does not work well with 8-byte pointers. Also, usually the system mallocs on such platforms are much better optimized for such large memory models than the Perl malloc. Some memory-hungry Perl applications like the PDL don't work well with Perl's malloc. Finally, other applications than Perl (such as mod_perl) tend to prefer the system malloc. Such platforms include Alpha and 64-bit HPPA, MIPS, PPC, and Sparc.

86.3.3 AIX Dynaloading

The AIX dynaloading now uses in AIX releases 4.3 and newer the native dlopen interface of AIX instead of the old emulated interface. This change will probably break backward compatibility with compiled modules. The change was made to make Perl more compliant with other applications like mod_perl which are using the AIX native interface.

86.3.4 Attributes for `my` variables now handled at run-time

The `my EXPR : ATTRS` syntax now applies variable attributes at run-time. (Subroutine and `our` variables still get attributes applied at compile-time.) See *attributes* for additional details. In particular, however, this allows variable attributes to be useful for `tie` interfaces, which was a deficiency of earlier releases. Note that the new semantics doesn't work with the Attribute::Handlers module (as of version 0.76).

86.3.5 Socket Extension Dynamic in VMS

The Socket extension is now dynamically loaded instead of being statically built in. This may or may not be a problem with ancient TCP/IP stacks of VMS: we do not know since we weren't able to test Perl in such configurations.

86.3.6 IEEE-format Floating Point Default on OpenVMS Alpha

Perl now uses IEEE format (T_FLOAT) as the default internal floating point format on OpenVMS Alpha, potentially breaking binary compatibility with external libraries or existing data. G_FLOAT is still available as a configuration option. The default on VAX (D_FLOAT) has not changed.

86.3.7 New Unicode Semantics (no more `use utf8`, almost)

Previously in Perl 5.6 to use Unicode one would say "use utf8" and then the operations (like string concatenation) were Unicode-aware in that lexical scope.

This was found to be an inconvenient interface, and in Perl 5.8 the Unicode model has completely changed: now the "Unicodeness" is bound to the data itself, and for most of the time "use utf8" is not needed at all. The only remaining use of "use utf8" is when the Perl script itself has been written in the UTF-8 encoding of Unicode. (UTF-8 has not been made the default since there are many Perl scripts out there that are using various national eight-bit character sets, which would be illegal in UTF-8.)

See *perluniintro* for the explanation of the current model, and *utf8* for the current use of the utf8 pragma.

86.3.8 New Unicode Properties

Unicode *scripts* are now supported. Scripts are similar to (and superior to) Unicode *blocks*. The difference between scripts and blocks is that scripts are the glyphs used by a language or a group of languages, while the blocks are more artificial groupings of (mostly) 256 characters based on the Unicode numbering.

In general, scripts are more inclusive, but not universally so. For example, while the script `Latin` includes all the Latin characters and their various diacritic-adorned versions, it does not include the various punctuation or digits (since they are not solely `Latin`).

A number of other properties are now supported, including `\p{L&}`, `\p{Any}` `\p{Assigned}`, `\p{Unassigned}`, `\p{Blank}` [561] and `\p{SpacePerl}` [561] (along with their `\P{...}` versions, of course). See *perlunicode* for details, and more additions.

The `In` or `Is` prefix to names used with the `\p{...}` and `\P{...}` are now almost always optional. The only exception is that a `In` prefix is required to signify a Unicode block when a block name conflicts with a script name. For example, `\p{Tibetan}` refers to the script, while `\p{InTibetan}` refers to the block. When there is no name conflict, you can omit the `In` from the block name (e.g. `\p{BraillePatterns}`), but to be safe, it's probably best to always use the `In`).

86.3.9 REF(...) Instead Of SCALAR(...)

A reference to a reference now stringifies as "REF(0x81485ec)" instead of "SCALAR(0x81485ec)" in order to be more consistent with the return value of ref().

86.3.10 pack/unpack D/F recycled

The undocumented pack/unpack template letters D/F have been recycled for better use: now they stand for long double (if supported by the platform) and NV (Perl internal floating point type). (They used to be aliases for d/f, but you never knew that.)

86.3.11 glob() now returns filenames in alphabetical order

The list of filenames from glob() (or <...>) is now by default sorted alphabetically to be csh-compliant (which is what happened before in most UNIX platforms). (bsd_glob() does still sort platform natively, ASCII or EBCDIC, unless GLOB_ALPHASORT is specified.) [561]

86.3.12 Deprecations

- The semantics of bless(REF, REF) were unclear and until someone proves it to make some sense, it is forbidden.
- The obsolete chat2 library that should never have been allowed to escape the laboratory has been decommissioned.
- Using chdir("") or chdir(undef) instead of explicit chdir() is doubtful. A failure (think chdir(some_function()) can lead into unintended chdir() to the home directory, therefore this behaviour is deprecated.
- The builtin dump() function has probably outlived most of its usefulness. The core-dumping functionality will remain in future available as an explicit call to `CORE::dump()`, but in future releases the behaviour of an unqualified `dump()` call may change.
- The very dusty examples in the eg/ directory have been removed. Suggestions for new shiny examples welcome but the main issue is that the examples need to be documented, tested and (most importantly) maintained.
- The (bogus) escape sequences \8 and \9 now give an optional warning ("Unrecognized escape passed through"). There is no need to \-escape any `\w` character.
- The *glob{FILEHANDLE} is deprecated, use *glob{IO} instead.

- The `package;` syntax (`package` without an argument) has been deprecated. Its semantics were never that clear and its implementation even less so. If you have used that feature to disallow all but fully qualified variables, `use strict;` instead.

- The unimplemented POSIX regex features [[.cc.]] and [[=c=]] are still recognised but now cause fatal errors. The previous behaviour of ignoring them by default and warning if requested was unacceptable since it, in a way, falsely promised that the features could be used.

- In future releases, non-PerlIO aware XS modules may become completely unsupported. Since PerlIO is a drop-in replacement for stdio at the source code level, this shouldn't be that drastic a change.

- Previous versions of perl and some readings of some sections of Camel III implied that the `:raw` "discipline" was the inverse of `:crlf`. Turning off "clrfness" is no longer enough to make a stream truly binary. So the PerlIO `:raw` layer (or "discipline", to use the Camel book's older terminology) is now formally defined as being equivalent to binmode(FH) - which is in turn defined as doing whatever is necessary to pass each byte as-is without any translation. In particular binmode(FH) - and hence `:raw` - will now turn off both CRLF and UTF-8 translation and remove other layers (e.g. :encoding()) which would modify byte stream.

- The current user-visible implementation of pseudo-hashes (the weird use of the first array element) is deprecated starting from Perl 5.8.0 and will be removed in Perl 5.10.0, and the feature will be implemented differently. Not only is the current interface rather ugly, but the current implementation slows down normal array and hash use quite noticeably. The `fields` pragma interface will remain available. The *restricted hashes* interface is expected to be the replacement interface (see *Hash::Util*). If your existing programs depends on the underlying implementation, consider using *Class::PseudoHash* from CPAN.

- The syntaxes `@a->[...]` and `%h->{...}` have now been deprecated.

- After years of trying, suidperl is considered to be too complex to ever be considered truly secure. The suidperl functionality is likely to be removed in a future release.

- The 5.005 threads model (module `Thread`) is deprecated and expected to be removed in Perl 5.10. Multithreaded code should be migrated to the new ithreads model (see *threads*, *threads::shared* and *perlthrtut*).

- The long deprecated uppercase aliases for the string comparison operators (EQ, NE, LT, LE, GE, GT) have now been removed.

- The tr///C and tr///U features have been removed and will not return; the interface was a mistake. Sorry about that. For similar functionality, see pack('U0', ...) and pack('C0', ...). [561]

- Earlier Perls treated "sub foo (@bar)" as equivalent to "sub foo (@)". The prototypes are now checked better at compile-time for invalid syntax. An optional warning is generated ("Illegal character in prototype...") but this may be upgraded to a fatal error in a future release.

- The `exec LIST` and `system LIST` operations now produce warnings on tainted data and in some future release they will produce fatal errors.

- The existing behaviour when localising tied arrays and hashes is wrong, and will be changed in a future release, so do not rely on the existing behaviour. See §86.15.2.

86.4 Core Enhancements

86.4.1 Unicode Overhaul

Unicode in general should be now much more usable than in Perl 5.6.0 (or even in 5.6.1). Unicode can be used in hash keys, Unicode in regular expressions should work now, Unicode in tr/// should work now, Unicode in I/O should work now. See *perluniintro* for introduction and *perlunicode* for details.

- The Unicode Character Database coming with Perl has been upgraded to Unicode 3.2.0. For more information, see http://www.unicode.org/ . [561+] (5.6.1 has UCD 3.0.1.)

- For developers interested in enhancing Perl's Unicode capabilities: almost all the UCD files are included with the Perl distribution in the *lib/unicore* subdirectory. The most notable omission, for space considerations, is the Unihan database.
- The properties \p{Blank} and \p{SpacePerl} have been added. "Blank" is like C isblank(), that is, it contains only "horizontal whitespace" (the space character is, the newline isn't), and the "SpacePerl" is the Unicode equivalent of `\s` (`\p{Space}` isn't, since that includes the vertical tabulator character, whereas `\s` doesn't.)

  See "New Unicode Properties" earlier in this document for additional information on changes with Unicode properties.

86.4.2 PerlIO is Now The Default

- IO is now by default done via PerlIO rather than system's "stdio". PerlIO allows "layers" to be "pushed" onto a file handle to alter the handle's behaviour. Layers can be specified at open time via 3-arg form of open:

  ```
  open($fh,'>:crlf :utf8', $path) || ...
  ```

  or on already opened handles via extended `binmode`:

  ```
  binmode($fh,':encoding(iso-8859-7)');
  ```

  The built-in layers are: unix (low level read/write), stdio (as in previous Perls), perlio (re-implementation of stdio buffering in a portable manner), crlf (does CRLF <=> "\n" translation as on Win32, but available on any platform). A mmap layer may be available if platform supports it (mostly UNIXes).

  Layers to be applied by default may be specified via the 'open' pragma.

  See §86.9 for the effects of PerlIO on your architecture name.
- If your platform supports fork(), you can use the list form of `open` for pipes. For example:

  ```
   open KID_PS, "-|", "ps", "aux" or die $!;
  ```

  forks the ps(1) command (without spawning a shell, as there are more than three arguments to open()), and reads its standard output via the `KID_PS` filehandle. See *perlipc*.
- File handles can be marked as accepting Perl's internal encoding of Unicode (UTF-8 or UTF-EBCDIC depending on platform) by a pseudo layer ":utf8" :

  ```
  open($fh,">:utf8","Uni.txt");
  ```

  Note for EBCDIC users: the pseudo layer ":utf8" is erroneously named for you since it's not UTF-8 what you will be getting but instead UTF-EBCDIC. See *perlunicode*, *utf8*, and http://www.unicode.org/unicode/reports/tr16/ for more information. In future releases this naming may change. See *perluniintro* for more information about UTF-8.
- If your environment variables (LC_ALL, LC_CTYPE, LANG) look like you want to use UTF-8 (any of the the variables match `/utf-?8/i`), your STDIN, STDOUT, STDERR handles and the default open layer (see *open*) are marked as UTF-8. (This feature, like other new features that combine Unicode and I/O, work only if you are using PerlIO, but that's the default.)

  Note that after this Perl really does assume that everything is UTF-8: for example if some input handle is not, Perl will probably very soon complain about the input data like this "Malformed UTF-8 ..." since any old eight-bit data is not legal UTF-8.

  Note for code authors: if you want to enable your users to use UTF-8 as their default encoding but in your code still have eight-bit I/O streams (such as images or zip files), you need to explicitly open() or binmode() with `:bytes` (see open in *perlfunc* and binmode in *perlfunc*), or you can just use `binmode(FH)` (nice for pre-5.8.0 backward compatibility).

- File handles can translate character encodings from/to Perl's internal Unicode form on read/write via the ":encoding()" layer.
- File handles can be opened to "in memory" files held in Perl scalars via:

  ```
  open($fh,'>', \$variable) || ...
  ```

- Anonymous temporary files are available without need to 'use FileHandle' or other module via

  ```
  open($fh,"+>", undef) || ...
  ```

  That is a literal undef, not an undefined value.

86.4.3 ithreads

The new interpreter threads ("ithreads" for short) implementation of multithreading, by Arthur Bergman, replaces the old "5.005 threads" implementation. In the ithreads model any data sharing between threads must be explicit, as opposed to the model where data sharing was implicit. See *threads* and *threads::shared*, and *perlthrtut*.

As a part of the ithreads implementation Perl will also use any necessary and detectable reentrant libc interfaces.

86.4.4 Restricted Hashes

A restricted hash is restricted to a certain set of keys, no keys outside the set can be added. Also individual keys can be restricted so that the key cannot be deleted and the value cannot be changed. No new syntax is involved: the Hash::Util module is the interface.

86.4.5 Safe Signals

Perl used to be fragile in that signals arriving at inopportune moments could corrupt Perl's internal state. Now Perl postpones handling of signals until it's safe (between opcodes).

This change may have surprising side effects because signals no longer interrupt Perl instantly. Perl will now first finish whatever it was doing, like finishing an internal operation (like sort()) or an external operation (like an I/O operation), and only then look at any arrived signals (and before starting the next operation). No more corrupt internal state since the current operation is always finished first, but the signal may take more time to get heard. Note that breaking out from potentially blocking operations should still work, though.

86.4.6 Understanding of Numbers

In general a lot of fixing has happened in the area of Perl's understanding of numbers, both integer and floating point. Since in many systems the standard number parsing functions like `strtoul()` and `atof()` seem to have bugs, Perl tries to work around their deficiencies. This results hopefully in more accurate numbers.

Perl now tries internally to use integer values in numeric conversions and basic arithmetics (+ - * /) if the arguments are integers, and tries also to keep the results stored internally as integers. This change leads to often slightly faster and always less lossy arithmetics. (Previously Perl always preferred floating point numbers in its math.)

86.4.7 Arrays now always interpolate into double-quoted strings [561]

In double-quoted strings, arrays now interpolate, no matter what. The behavior in earlier versions of perl 5 was that arrays would interpolate into strings if the array had been mentioned before the string was compiled, and otherwise Perl would raise a fatal compile-time error. In versions 5.000 through 5.003, the error was

```
        Literal @example now requires backslash
```

In versions 5.004_01 through 5.6.0, the error was

```
        In string, @example now must be written as \@example
```

The idea here was to get people into the habit of writing `"fred\@example.com"` when they wanted a literal `@` sign, just as they have always written `"Give me back my \$5"` when they wanted a literal `$` sign.

Starting with 5.6.1, when Perl now sees an `@` sign in a double-quoted string, it *always* attempts to interpolate an array, regardless of whether or not the array has been used or declared already. The fatal error has been downgraded to an optional warning:

```
        Possible unintended interpolation of @example in string
```

This warns you that `"fred@example.com"` is going to turn into `fred.com` if you don't backslash the `@`. See http://www.plover.com/˜mjd/perl/at-error.html for more details about the history here.

86.4.8 Miscellaneous Changes

- AUTOLOAD is now lvaluable, meaning that you can add the :lvalue attribute to AUTOLOAD subroutines and you can assign to the AUTOLOAD return value.
- The $Config{byteorder} (and corresponding BYTEORDER in config.h) was previously wrong in platforms if sizeof(long) was 4, but sizeof(IV) was 8. The byteorder was only sizeof(long) bytes long (1234 or 4321), but now it is correctly sizeof(IV) bytes long, (12345678 or 87654321). (This problem didn't affect Windows platforms.)

  Also, $Config{byteorder} is now computed dynamically–this is more robust with "fat binaries" where an executable image contains binaries for more than one binary platform, and when cross-compiling.
- `perl -d:Module=arg,arg,arg` now works (previously one couldn't pass in multiple arguments.)
- do followed by a bareword now ensures that this bareword isn't a keyword (to avoid a bug where `do q(foo.pl)` tried to call a subroutine called `q`). This means that for example instead of `do format()` you must write `do &format()`.
- The builtin dump() now gives an optional warning `dump() better written as CORE::dump()`, meaning that by default `dump(...)` is resolved as the builtin dump() which dumps core and aborts, not as (possibly) user-defined `sub dump`. To call the latter, qualify the call as `&dump(...)`. (The whole dump() feature is to considered deprecated, and possibly removed/changed in future releases.)
- chomp() and chop() are now overridable. Note, however, that their prototype (as given by `prototype("CORE::chomp")` is undefined, because it cannot be expressed and therefore one cannot really write replacements to override these builtins.
- END blocks are now run even if you exit/die in a BEGIN block. Internally, the execution of END blocks is now controlled by PL_exit_flags & PERL_EXIT_DESTRUCT_END. This enables the new behaviour for Perl embedders. This will default in 5.10. See *perlembed*.
- Formats now support zero-padded decimal fields.
- Although "you shouldn't do that", it was possible to write code that depends on Perl's hashed key order (Data::Dumper does this). The new algorithm "One-at-a-Time" produces a different hashed key order. More details are in §86.8.

- lstat(FILEHANDLE) now gives a warning because the operation makes no sense. In future releases this may become a fatal error.
- Spurious syntax errors generated in certain situations, when glob() caused File::Glob to be loaded for the first time, have been fixed. [561]
- Lvalue subroutines can now return `undef` in list context. However, the lvalue subroutine feature still remains experimental. [561+]
- A lost warning "Can't declare ... dereference in my" has been restored (Perl had it earlier but it became lost in later releases.)
- A new special regular expression variable has been introduced: `$^N`, which contains the most-recently closed group (submatch).
- `no Module;` does not produce an error even if Module does not have an unimport() method. This parallels the behavior of `use` vis-a-vis `import`. [561]
- The numerical comparison operators return `undef` if either operand is a NaN. Previously the behaviour was unspecified.
- `our` can now have an experimental optional attribute `unique` that affects how global variables are shared among multiple interpreters, see `our` in *perlfunc*.
- The following builtin functions are now overridable: each(), keys(), pop(), push(), shift(), splice(), unshift(). [561]
- `pack() / unpack()` can now group template letters with `()` and then apply repetition/count modifiers on the groups.
- `pack() / unpack()` can now process the Perl internal numeric types: IVs, UVs, NVs– and also long doubles, if supported by the platform. The template letters are `j`, `J`, `F`, and `D`.
- `pack('U0a*', ...)` can now be used to force a string to UTF-8.
- my __PACKAGE__ $obj now works. [561]
- POSIX::sleep() now returns the number of *unslept* seconds (as the POSIX standard says), as opposed to CORE::sleep() which returns the number of slept seconds.
- printf() and sprintf() now support parameter reordering using the `%\d+\$` and `*\d+\$` syntaxes. For example

  ```
  printf "%2\$s %1\$s\n", "foo", "bar";
  ```

  will print "bar foo\n". This feature helps in writing internationalised software, and in general when the order of the parameters can vary.
- The (\&) prototype now works properly. [561]
- prototype(\[$@%&]) is now available to implicitly create references (useful for example if you want to emulate the tie() interface).
- A new command-line option, `-t` is available. It is the little brother of `-T`: instead of dying on taint violations, lexical warnings are given. **This is only meant as a temporary debugging aid while securing the code of old legacy applications. This is not a substitute for -T.**
- In other taint news, the `exec LIST` and `system LIST` have now been considered too risky (think `exec @ARGV`: it can start any program with any arguments), and now the said forms cause a warning under lexical warnings. You should carefully launder the arguments to guarantee their validity. In future releases of Perl the forms will become fatal errors so consider starting laundering now.
- Tied hash interfaces are now required to have the EXISTS and DELETE methods (either own or inherited).
- If tr/// is just counting characters, it doesn't attempt to modify its target.

- untie() will now call an UNTIE() hook if it exists. See *perltie* for details. [561]
- *utime* now supports `utime undef, undef, @files` to change the file timestamps to the current time.
- The rules for allowing underscores (underbars) in numeric constants have been relaxed and simplified: now you can have an underscore simply **between digits**.
- Rather than relying on C's argv[0] (which may not contain a full pathname) where possible $ˆX is now set by asking the operating system. (eg by reading */proc/self/exe* on Linux, */proc/curproc/file* on FreeBSD)
- A new variable, `${ˆTAINT}`, indicates whether taint mode is enabled.
- You can now override the readline() builtin, and this overrides also the <FILEHANDLE> angle bracket operator.
- The command-line options -s and -F are now recognized on the shebang (#!) line.
- Use of the `/c` match modifier without an accompanying `/g` modifier elicits a new warning: `Use of /c modifier is meaningless without /g`.

  Use of `/c` in substitutions, even with `/g`, elicits `Use of /c modifier is meaningless in s///`.

  Use of `/g` with `split` elicits `Use of /g modifier is meaningless in split`.
- Support for the `CLONE` special subroutine had been added. With ithreads, when a new thread is created, all Perl data is cloned, however non-Perl data cannot be cloned automatically. 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 executed once for every package that has it defined or inherited. It will be called in the context of the new thread, so all modifications are made in the new area.

  See *perlmod*

86.5 Modules and Pragmata

86.5.1 New Modules and Pragmata

- `Attribute::Handlers`, originally by Damian Conway and now maintained by Arthur Bergman, allows a class to define attribute handlers.

```
package MyPack;
use Attribute::Handlers;
sub Wolf :ATTR(SCALAR) { print "howl!\n" }

# later, in some package using or inheriting from MyPack...

my MyPack $Fluffy : Wolf; # the attribute handler Wolf will be called
```

  Both variables and routines can have attribute handlers. Handlers can be specific to type (SCALAR, ARRAY, HASH, or CODE), or specific to the exact compilation phase (BEGIN, CHECK, INIT, or END). See *Attribute::Handlers*.
- `B::Concise`, by Stephen McCamant, is a new compiler backend for walking the Perl syntax tree, printing concise info about ops. The output is highly customisable. See *B::Concise*. [561+]
- The new bignum, bigint, and bigrat pragmas, by Tels, implement transparent bignum support (using the Math::BigInt, Math::BigFloat, and Math::BigRat backends).
- `Class::ISA`, by Sean Burke, is a module for reporting the search path for a class's ISA tree. See *Class::ISA*.
- `Cwd` now has a split personality: if possible, an XS extension is used, (this will hopefully be faster, more secure, and more robust) but if not possible, the familiar Perl implementation is used.

- `Devel::PPPort`, originally by Kenneth Albanowski and now maintained by Paul Marquess, has been added. It is primarily used by `h2xs` to enhance portability of XS modules between different versions of Perl. See *Devel::PPPort*.
- `Digest`, frontend module for calculating digests (checksums), from Gisle Aas, has been added. See *Digest*.
- `Digest::MD5` for calculating MD5 digests (checksums) as defined in RFC 1321, from Gisle Aas, has been added. See *Digest::MD5*.

  ```
  use Digest::MD5 'md5_hex';

  $digest = md5_hex("Thirsty Camel");

  print $digest, "\n"; # 01d19d9d2045e005c3f1b80e8b164de1
  ```

  NOTE: the `MD5` backward compatibility module is deliberately not included since its further use is discouraged.

  See also *PerlIO::via::QuotedPrint*.
- `Encode`, originally by Nick Ing-Simmons and now maintained by Dan Kogai, provides a mechanism to translate between different character encodings. Support for Unicode, ISO-8859-1, and ASCII are compiled in to the module. Several other encodings (like the rest of the ISO-8859, CP*/Win*, Mac, KOI8-R, three variants EBCDIC, Chinese, Japanese, and Korean encodings) are included and can be loaded at runtime. (For space considerations, the largest Chinese encodings have been separated into their own CPAN module, Encode::HanExtra, which Encode will use if available). See *Encode*.

  Any encoding supported by Encode module is also available to the ":encoding()" layer if PerlIO is used.
- `Hash::Util` is the interface to the new *restricted hashes* feature. (Implemented by Jeffrey Friedl, Nick Ing-Simmons, and Michael Schwern.) See *Hash::Util*.
- `I18N::Langinfo` can be used to query locale information. See *I18N::Langinfo*.
- `I18N::LangTags`, by Sean Burke, has functions for dealing with RFC3066-style language tags. See *I18N::LangTags*.
- `ExtUtils::Constant`, by Nicholas Clark, is a new tool for extension writers for generating XS code to import C header constants. See *ExtUtils::Constant*.
- `Filter::Simple`, by Damian Conway, is an easy-to-use frontend to Filter::Util::Call. See *Filter::Simple*.

  ```
  # in MyFilter.pm:

  package MyFilter;

  use Filter::Simple sub {
      while (my ($from, $to) = splice @_, 0, 2) {
              s/$from/$to/g;
      }
  };

  1;

  # in user's code:

  use MyFilter qr/red/ => 'green';

  print "red\n";   # this code is filtered, will print "green\n"
  print "bored\n"; # this code is filtered, will print "bogreen\n"
  ```

```
    no MyFilter;

    print "red\n";   # this code is not filtered, will print "red\n"
```

- `File::Temp`, by Tim Jenness, allows one to create temporary files and directories in an easy, portable, and secure way. See *File::Temp*. [561+]
- `Filter::Util::Call`, by Paul Marquess, provides you with the framework to write *source filters* in Perl. For most uses, the frontend Filter::Simple is to be preferred. See *Filter::Util::Call*.
- `if`, by Ilya Zakharevich, is a new pragma for conditional inclusion of modules.
- *libnet*, by Graham Barr, is a collection of perl5 modules related to network programming. See *Net::FTP*, *Net::NNTP*, *Net::Ping* (not part of libnet, but related), *Net::POP3*, *Net::SMTP*, and *Net::Time*.

  Perl installation leaves libnet unconfigured; use *libnetcfg* to configure it.
- `List::Util`, by Graham Barr, is a selection of general-utility list subroutines, such as sum(), min(), first(), and shuffle(). See *List::Util*.
- `Locale::Constants`, `Locale::Country`, `Locale::Currency` `Locale::Language`, and *Locale::Script*, by Neil Bowers, have been added. They provide the codes for various locale standards, such as "fr" for France, "usd" for US Dollar, and "ja" for Japanese.

  ```
  use Locale::Country;

  $country = code2country('jp');               # $country gets 'Japan'
  $code    = country2code('Norway');           # $code gets 'no'
  ```

  See *Locale::Constants*, *Locale::Country*, *Locale::Currency*, and *Locale::Language*.
- `Locale::Maketext`, by Sean Burke, is a localization framework. See *Locale::Maketext*, and *Locale::Maketext::TPJ13*. The latter is an article about software localization, originally published in The Perl Journal #13, and republished here with kind permission.
- `Math::BigRat` for big rational numbers, to accompany Math::BigInt and Math::BigFloat, from Tels. See *Math::BigRat*.
- `Memoize` can make your functions faster by trading space for time, from Mark-Jason Dominus. See *Memoize*.
- `MIME::Base64`, by Gisle Aas, allows you to encode data in base64, as defined in RFC 2045 - *MIME (Multipurpose Internet Mail Extensions)*.

  ```
  use MIME::Base64;

  $encoded = encode_base64('Aladdin:open sesame');
  $decoded = decode_base64($encoded);

  print $encoded, "\n"; # "QWxhZGRpbjpvcGVuIHNlc2FtZQ=="
  ```

  See *MIME::Base64*.
- `MIME::QuotedPrint`, by Gisle Aas, allows you to encode data in quoted-printable encoding, as defined in RFC 2045 - *MIME (Multipurpose Internet Mail Extensions)*.

  ```
  use MIME::QuotedPrint;

  $encoded = encode_qp("\xDE\xAD\xBE\xEF");
  $decoded = decode_qp($encoded);
  ```

```
    print $encoded, "\n"; # "=DE=AD=BE=EF\n"
    print $decoded, "\n"; # "\xDE\xAD\xBE\xEF\n"
```

See also *PerlIO::via::QuotedPrint*.

- `NEXT`, by Damian Conway, is a pseudo-class for method redispatch. See *NEXT*.
- `open` is a new pragma for setting the default I/O layers for open().
- `PerlIO::scalar`, by Nick Ing-Simmons, provides the implementation of IO to "in memory" Perl scalars as discussed above. It also serves as an example of a loadable PerlIO layer. Other future possibilities include PerlIO::Array and PerlIO::Code. See *PerlIO::scalar*.
- `PerlIO::via`, by Nick Ing-Simmons, acts as a PerlIO layer and wraps PerlIO layer functionality provided by a class (typically implemented in Perl code).
- `PerlIO::via::QuotedPrint`, by Elizabeth Mattijsen, is an example of a `PerlIO::via` class:

```
    use PerlIO::via::QuotedPrint;
    open($fh,">:via(QuotedPrint)",$path);
```

  This will automatically convert everything output to `$fh` to Quoted-Printable. See *PerlIO::via* and *PerlIO::via::QuotedPrint*.

- `Pod::ParseLink`, by Russ Allbery, has been added, to parse L<> links in pods as described in the new perlpodspec.
- `Pod::Text::Overstrike`, by Joe Smith, has been added. It converts POD data to formatted overstrike text. See *Pod::Text::Overstrike*. [561+]
- `Scalar::Util` is a selection of general-utility scalar subroutines, such as blessed(), reftype(), and tainted(). See *Scalar::Util*.
- `sort` is a new pragma for controlling the behaviour of sort().
- `Storable` gives persistence to Perl data structures by allowing the storage and retrieval of Perl data to and from files in a fast and compact binary format. Because in effect Storable does serialisation of Perl data structures, with it you can also clone deep, hierarchical datastructures. Storable was originally created by Raphael Manfredi, but it is now maintained by Abhijit Menon-Sen. Storable has been enhanced to understand the two new hash features, Unicode keys and restricted hashes. See *Storable*.
- `Switch`, by Damian Conway, has been added. Just by saying

```
    use Switch;
```

  you have `switch` and `case` available in Perl.

```
    use Switch;

    switch ($val) {

                case 1          { print "number 1" }
                case "a"        { print "string a" }
                case [1..10,42] { print "number in list" }
                case (@array)   { print "number in list" }
                case /\w+/      { print "pattern" }
                case qr/\w+/    { print "pattern" }
                case (%hash)    { print "entry in hash" }
                case (\%hash)   { print "entry in hash" }
                case (\&sub)    { print "arg to subroutine" }
                else            { print "previous case not true" }
    }
```

See *Switch*.

- `Test::More`, by Michael Schwern, is yet another framework for writing test scripts, more extensive than Test::Simple. See *Test::More*.
- `Test::Simple`, by Michael Schwern, has basic utilities for writing tests. See *Test::Simple*.
- `Text::Balanced`, by Damian Conway, has been added, for extracting delimited text sequences from strings.

```
    use Text::Balanced 'extract_delimited';

    ($a, $b) = extract_delimited("'never say never', he never said", "'", '');
```

  $a will be "'never say never'", $b will be ', he never said'.

  In addition to extract_delimited(), there are also extract_bracketed(), extract_quotelike(), extract_codeblock(), extract_variable(), extract_tagged(), extract_multiple(), gen_delimited_pat(), and gen_extract_tagged(). With these, you can implement rather advanced parsing algorithms. See *Text::Balanced*.
- `threads`, by Arthur Bergman, is an interface to interpreter threads. Interpreter threads (ithreads) is the new thread model introduced in Perl 5.6 but only available as an internal interface for extension writers (and for Win32 Perl for `fork()` emulation). See *threads*, *threads::shared*, and *perlthrtut*.
- `threads::shared`, by Arthur Bergman, allows data sharing for interpreter threads. See *threads::shared*.
- `Tie::File`, by Mark-Jason Dominus, associates a Perl array with the lines of a file. See *Tie::File*.
- `Tie::Memoize`, by Ilya Zakharevich, provides on-demand loaded hashes. See *Tie::Memoize*.
- `Tie::RefHash::Nestable`, by Edward Avis, allows storing hash references (unlike the standard Tie::RefHash) The module is contained within Tie::RefHash. See *Tie::RefHash*.
- `Time::HiRes`, by Douglas E. Wegscheid, provides high resolution timing (ualarm, usleep, and gettimeofday). See *Time::HiRes*.
- `Unicode::UCD` offers a querying interface to the Unicode Character Database. See *Unicode::UCD*.
- `Unicode::Collate`, by SADAHIRO Tomoyuki, implements the UCA (Unicode Collation Algorithm) for sorting Unicode strings. See *Unicode::Collate*.
- `Unicode::Normalize`, by SADAHIRO Tomoyuki, implements the various Unicode normalization forms. See *Unicode::Normalize*.
- `XS::APItest`, by Tim Jenness, is a test extension that exercises XS APIs. Currently only `printf()` is tested: how to output various basic data types from XS.
- `XS::Typemap`, by Tim Jenness, is a test extension that exercises XS typemaps. Nothing gets installed, but the code is worth studying for extension writers.

86.5.2 Updated And Improved Modules and Pragmata

- The following independently supported modules have been updated to the newest versions from CPAN: CGI, CPAN, DB_File, File::Spec, File::Temp, Getopt::Long, Math::BigFloat, Math::BigInt, the podlators bundle (Pod::Man, Pod::Text), Pod::LaTeX [561+], Pod::Parser, Storable, Term::ANSIColor, Test, Text-Tabs+Wrap.
- attributes::reftype() now works on tied arguments.
- AutoLoader can now be disabled with `no AutoLoader;`.
- B::Deparse has been significantly enhanced by Robin Houston. It can now deparse almost all of the standard test suite (so that the tests still succeed). There is a make target "test.deparse" for trying this out.

- Carp now has better interface documentation, and the @CARP_NOT interface has been added to get optional control over where errors are reported independently of @ISA, by Ben Tilly.
- Class::Struct can now define the classes in compile time.
- Class::Struct now assigns the array/hash element if the accessor is called with an array/hash element as the **sole** argument.
- The return value of Cwd::fastcwd() is now tainted.
- Data::Dumper now has an option to sort hashes.
- Data::Dumper now has an option to dump code references using B::Deparse.
- DB_File now supports newer Berkeley DB versions, among other improvements.
- Devel::Peek now has an interface for the Perl memory statistics (this works only if you are using perl's malloc, and if you have compiled with debugging).
- The English module can now be used without the infamous performance hit by saying

  ```
          use English '-no_match_vars';
  ```

  (Assuming, of course, that you don't need the troublesome variables `` $` ``, `$&`, or `$'`.) Also, introduced `@LAST_MATCH_START` and `@LAST_MATCH_END` English aliases for `@-` and `@+`.
- ExtUtils::MakeMaker has been significantly cleaned up and fixed. The enhanced version has also been backported to earlier releases of Perl and submitted to CPAN so that the earlier releases can enjoy the fixes.
- The arguments of WriteMakefile() in Makefile.PL are now checked for sanity much more carefully than before. This may cause new warnings when modules are being installed. See *ExtUtils::MakeMaker* for more details.
- ExtUtils::MakeMaker now uses File::Spec internally, which hopefully leads to better portability.
- Fcntl, Socket, and Sys::Syslog have been rewritten by Nicholas Clark to use the new-style constant dispatch section (see *ExtUtils::Constant*). This means that they will be more robust and hopefully faster.
- File::Find now chdir()s correctly when chasing symbolic links. [561]
- File::Find now has pre- and post-processing callbacks. It also correctly changes directories when chasing symbolic links. Callbacks (naughtily) exiting with "next;" instead of "return;" now work.
- File::Find is now (again) reentrant. It also has been made more portable.
- The warnings issued by File::Find now belong to their own category. You can enable/disable them with `use/no warnings 'File::Find';`.
- File::Glob::glob() has been renamed to File::Glob::bsd_glob() because the name clashes with the builtin glob(). The older name is still available for compatibility, but is deprecated. [561]
- File::Glob now supports `GLOB_LIMIT` constant to limit the size of the returned list of filenames.
- IPC::Open3 now allows the use of numeric file descriptors.
- IO::Socket now has an atmark() method, which returns true if the socket is positioned at the out-of-band mark. The method is also exportable as a sockatmark() function.
- IO::Socket::INET failed to open the specified port if the service name was not known. It now correctly uses the supplied port number as is. [561]
- IO::Socket::INET has support for the ReusePort option (if your platform supports it). The Reuse option now has an alias, ReuseAddr. For clarity, you may want to prefer ReuseAddr.
- IO::Socket::INET now supports a value of zero for `LocalPort` (usually meaning that the operating system will make one up.)

- ’use lib’ now works identically to @INC. Removing directories with ’no lib’ now works.
- Math::BigFloat and Math::BigInt have undergone a full rewrite by Tels. They are now magnitudes faster, and they support various bignum libraries such as GMP and PARI as their backends.
- Math::Complex handles inf, NaN etc., better.
- Net::Ping has been considerably enhanced by Rob Brown: multihoming is now supported, Win32 functionality is better, there is now time measuring functionality (optionally high-resolution using Time::HiRes), and there is now "external" protocol which uses Net::Ping::External module which runs your external ping utility and parses the output. A version of Net::Ping::External is available in CPAN.

  Note that some of the Net::Ping tests are disabled when running under the Perl distribution since one cannot assume one or more of the following: enabled echo port at localhost, full Internet connectivity, or sympathetic firewalls. You can set the environment variable PERL_TEST_Net_Ping to "1" (one) before running the Perl test suite to enable all the Net::Ping tests.
- POSIX::sigaction() is now much more flexible and robust. You can now install coderef handlers, ’DEFAULT’, and ’IGNORE’ handlers, installing new handlers was not atomic.
- In Safe, `%INC` is now localised in a Safe compartment so that use/require work.
- In SDBM_File on dosish platforms, some keys went missing because of lack of support for files with "holes". A workaround for the problem has been added.
- In Search::Dict one can now have a pre-processing hook for the lines being searched.
- The Shell module now has an OO interface.
- In Sys::Syslog there is now a failover mechanism that will go through alternative connection mechanisms until the message is successfully logged.
- The Test module has been significantly enhanced.
- Time::Local::timelocal() does not handle fractional seconds anymore. The rationale is that neither does localtime(), and timelocal() and localtime() are supposed to be inverses of each other.
- The vars pragma now supports declaring fully qualified variables. (Something that `our()` does not and will not support.)
- The `utf8::` name space (as in the pragma) provides various Perl-callable functions to provide low level access to Perl’s internal Unicode representation. At the moment only length() has been implemented.

86.6 Utility Changes

- Emacs perl mode (emacs/cperl-mode.el) has been updated to version 4.31.
- *emacs/e2ctags.pl* is now much faster.
- `enc2xs` is a tool for people adding their own encodings to the Encode module.
- `h2ph` now supports C trigraphs.
- `h2xs` now produces a template README.
- `h2xs` now uses `Devel::PPPort` for better portability between different versions of Perl.
- `h2xs` uses the new ExtUtils::Constant module which will affect newly created extensions that define constants. Since the new code is more correct (if you have two constants where the first one is a prefix of the second one, the first constant **never** got defined), less lossy (it uses integers for integer constant, as opposed to the old code that used floating point numbers even for integer constants), and slightly faster, you might want to consider regenerating your extension code (the new scheme makes regenerating easy). *h2xs* now also supports C trigraphs.

- `libnetcfg` has been added to configure libnet.
- `perlbug` is now much more robust. It also sends the bug report to perl.org, not perl.com.
- `perlcc` has been rewritten and its user interface (that is, command line) is much more like that of the UNIX C compiler, cc. (The perlbc tools has been removed. Use `perlcc -B` instead.) **Note that perlcc is still considered very experimental and unsupported.** [561]
- `perlivp` is a new Installation Verification Procedure utility for running any time after installing Perl.
- `piconv` is an implementation of the character conversion utility `iconv`, demonstrating the new Encode module.
- `pod2html` now allows specifying a cache directory.
- `pod2html` now produces XHTML 1.0.
- `pod2html` now understands POD written using different line endings (PC-like CRLF versus UNIX-like LF versus MacClassic-like CR).
- `s2p` has been completely rewritten in Perl. (It is in fact a full implementation of sed in Perl: you can use the sed functionality by using the `psed` utility.)
- `xsubpp` now understands POD documentation embedded in the *.xs files. [561]
- `xsubpp` now supports the OUT keyword.

86.7 New Documentation

- perl56delta details the changes between the 5.005 release and the 5.6.0 release.
- perlclib documents the internal replacements for standard C library functions. (Interesting only for extension writers and Perl core hackers.) [561+]
- perldebtut is a Perl debugging tutorial. [561+]
- perlebcdic contains considerations for running Perl on EBCDIC platforms. [561+]
- perlintro is a gentle introduction to Perl.
- perliol documents the internals of PerlIO with layers.
- perlmodstyle is a style guide for writing modules.
- perlnewmod tells about writing and submitting a new module. [561+]
- perlpacktut is a pack() tutorial.
- perlpod has been rewritten to be clearer and to record the best practices gathered over the years.
- perlpodspec is a more formal specification of the pod format, mainly of interest for writers of pod applications, not to people writing in pod.
- perlretut is a regular expression tutorial. [561+]
- perlrequick is a regular expressions quick-start guide. Yes, much quicker than perlretut. [561]
- perltodo has been updated.
- perltootc has been renamed as perltooc (to not to conflict with perltoot in filesystems restricted to "8.3" names).
- perluniintro is an introduction to using Unicode in Perl. (perlunicode is more of a detailed reference and background information)
- perlutil explains the command line utilities packaged with the Perl distribution. [561+]

The following platform-specific documents are available before the installation as README.*platform*, and after the installation as perl*platform*:

```
    perlaix perlamiga perlapollo perlbeos perlbs2000
    perlce perlcygwin perldgux perldos perlepoc perlfreebsd perlhpux
    perlhurd perlirix perlmachten perlmacos perlmint perlmpeix
    perlnetware perlos2 perlos390 perlplan9 perlqnx perlsolaris
    perltru64 perluts perlvmesa perlvms perlvos perlwin32
```

These documents usually detail one or more of the following subjects: configuring, building, testing, installing, and sometimes also using Perl on the said platform.

Eastern Asian Perl users are now welcomed in their own languages: README.jp (Japanese), README.ko (Korean), README.cn (simplified Chinese) and README.tw (traditional Chinese), which are written in normal pod but encoded in EUC-JP, EUC-KR, EUC-CN and Big5. These will get installed as

```
   perljp perlko perlcn perltw
```

- The documentation for the POSIX-BC platform is called "BS2000", to avoid confusion with the Perl POSIX module.
- The documentation for the WinCE platform is called perlce (README.ce in the source code kit), to avoid confusion with the perlwin32 documentation on 8.3-restricted filesystems.

86.8 Performance Enhancements

- map() could get pathologically slow when the result list it generates is larger than the source list. The performance has been improved for common scenarios. [561]
- sort() is also fully reentrant, in the sense that the sort function can itself call sort(). This did not work reliably in previous releases. [561]
- sort() has been changed to use primarily mergesort internally as opposed to the earlier quicksort. For very small lists this may result in slightly slower sorting times, but in general the speedup should be at least 20%. Additional bonuses are that the worst case behaviour of sort() is now better (in computer science terms it now runs in time O(N log N), as opposed to quicksort's Theta(N**2) worst-case run time behaviour), and that sort() is now stable (meaning that elements with identical keys will stay ordered as they were before the sort). See the `sort` pragma for information.

  The story in more detail: suppose you want to serve yourself a little slice of Pi.

  ```
      @digits = ( 3,1,4,1,5,9 );
  ```

  A numerical sort of the digits will yield (1,1,3,4,5,9), as expected. Which 1 comes first is hard to know, since one 1 looks pretty much like any other. You can regard this as totally trivial, or somewhat profound. However, if you just want to sort the even digits ahead of the odd ones, then what will

  ```
      sort { ($a % 2) <=> ($b % 2) } @digits;
  ```

  yield? The only even digit, 4, will come first. But how about the odd numbers, which all compare equal? With the quicksort algorithm used to implement Perl 5.6 and earlier, the order of ties is left up to the sort. So, as you add more and more digits of Pi, the order in which the sorted even and odd digits appear will change. and, for sufficiently large slices of Pi, the quicksort algorithm in Perl 5.8 won't return the same results even if reinvoked with the same input. The justification for this rests with quicksort's worst case behavior. If you run

  ```
     sort { $a <=> $b } ( 1 .. $N , 1 .. $N );
  ```

(something you might approximate if you wanted to merge two sorted arrays using sort), doubling $N doesn't just double the quicksort time, it *quadruples* it. Quicksort has a worst case run time that can grow like N**2, so-called *quadratic* behaviour, and it can happen on patterns that may well arise in normal use. You won't notice this for small arrays, but you *will* notice it with larger arrays, and you may not live long enough for the sort to complete on arrays of a million elements. So the 5.8 quicksort scrambles large arrays before sorting them, as a statistical defence against quadratic behaviour. But that means if you sort the same large array twice, ties may be broken in different ways.

Because of the unpredictability of tie-breaking order, and the quadratic worst-case behaviour, quicksort was *almost* replaced completely with a stable mergesort. *Stable* means that ties are broken to preserve the original order of appearance in the input array. So

```
    sort { ($a % 2) <=> ($b % 2) } (3,1,4,1,5,9);
```

will yield (4,3,1,1,5,9), guaranteed. The even and odd numbers appear in the output in the same order they appeared in the input. Mergesort has worst case O(N log N) behaviour, the best value attainable. And, ironically, this mergesort does particularly well where quicksort goes quadratic: mergesort sorts (1..$N, 1..$N) in O(N) time. But quicksort was rescued at the last moment because it is faster than mergesort on certain inputs and platforms. For example, if you really *don't* care about the order of even and odd digits, quicksort will run in O(N) time; it's very good at sorting many repetitions of a small number of distinct elements. The quicksort divide and conquer strategy works well on platforms with relatively small, very fast, caches. Eventually, the problem gets whittled down to one that fits in the cache, from which point it benefits from the increased memory speed.

Quicksort was rescued by implementing a sort pragma to control aspects of the sort. The **stable** subpragma forces stable behaviour, regardless of algorithm. The **_quicksort** and **_mergesort** subpragmas are heavy-handed ways to select the underlying implementation. The leading **_** is a reminder that these subpragmas may not survive beyond 5.8. More appropriate mechanisms for selecting the implementation exist, but they wouldn't have arrived in time to save quicksort.

- Hashes now use Bob Jenkins "One-at-a-Time" hashing key algorithm (http://burtleburtle.net/bob/hash/doobs.html). This algorithm is reasonably fast while producing a much better spread of values than the old hashing algorithm (originally by Chris Torek, later tweaked by Ilya Zakharevich). Hash values output from the algorithm on a hash of all 3-char printable ASCII keys comes much closer to passing the DIEHARD random number generation tests. According to perlbench, this change has not affected the overall speed of Perl.

- unshift() should now be noticeably faster.

86.9 Installation and Configuration Improvements

86.9.1 Generic Improvements

- INSTALL now explains how you can configure Perl to use 64-bit integers even on non-64-bit platforms.

- Policy.sh policy change: if you are reusing a Policy.sh file (see INSTALL) and you use Configure -Dprefix=/foo/bar and in the old Policy $prefix eq $siteprefix and $prefix eq $vendorprefix, all of them will now be changed to the new prefix, /foo/bar. (Previously only $prefix changed.) If you do not like this new behaviour, specify prefix, siteprefix, and vendorprefix explicitly.

- A new optional location for Perl libraries, otherlibdirs, is available. It can be used for example for vendor add-ons without disturbing Perl's own library directories.

- In many platforms, the vendor-supplied 'cc' is too stripped-down to build Perl (basically, 'cc' doesn't do ANSI C). If this seems to be the case and 'cc' does not seem to be the GNU C compiler 'gcc', an automatic attempt is made to find and use 'gcc' instead.

- gcc needs to closely track the operating system release to avoid build problems. If Configure finds that gcc was built for a different operating system release than is running, it now gives a clearly visible warning that there may be trouble ahead.

- Since Perl 5.8 is not binary-compatible with previous releases of Perl, Configure no longer suggests including the 5.005 modules in @INC.
- Configure `-S` can now run non-interactively. [561]
- Configure support for pdp11-style memory models has been removed due to obsolescence. [561]
- configure.gnu now works with options with whitespace in them.
- installperl now outputs everything to STDERR.
- Because PerlIO is now the default on most platforms, "-perlio" doesn't get appended to the $Config{archname} (also known as $^O) anymore. Instead, if you explicitly choose not to use perlio (Configure command line option -Uuseperlio), you will get "-stdio" appended.
- Another change related to the architecture name is that "-64all" (-Duse64bitall, or "maximally 64-bit") is appended only if your pointers are 64 bits wide. (To be exact, the use64bitall is ignored.)
- In AFS installations, one can configure the root of the AFS to be somewhere else than the default */afs* by using the Configure parameter `-Dafsroot=/some/where/else`.
- APPLLIB_EXP, a lesser-known configuration-time definition, has been documented. It can be used to prepend site-specific directories to Perl's default search path (@INC); see INSTALL for information.
- The version of Berkeley DB used when the Perl (and, presumably, the DB_File extension) was built is now available as `@Config{qw(db_version_major db_version_minor db_version_patch)}` from Perl and as `DB_VERSION_MAJOR_CFG DB_VERSION_MINOR_CFG DB_VERSION_PATCH_CFG` from C.
- Building Berkeley DB3 for compatibility modes for DB, NDBM, and ODBM has been documented in INSTALL.
- If you have CPAN access (either network or a local copy such as a CD-ROM) you can during specify extra modules to Configure to build and install with Perl using the -Dextras=... option. See INSTALL for more details.
- In addition to config.over, a new override file, config.arch, is available. This file is supposed to be used by hints file writers for architecture-wide changes (as opposed to config.over which is for site-wide changes).
- If your file system supports symbolic links, you can build Perl outside of the source directory by

  ```
  mkdir perl/build/directory
  cd perl/build/directory
  sh /path/to/perl/source/Configure -Dmksymlinks ...
  ```

  This will create in perl/build/directory a tree of symbolic links pointing to files in /path/to/perl/source. The original files are left unaffected. After Configure has finished, you can just say

  ```
  make all test
  ```

  and Perl will be built and tested, all in perl/build/directory. [561]
- For Perl developers, several new make targets for profiling and debugging have been added; see *perlhack*.
  - Use of the *gprof* tool to profile Perl has been documented in *perlhack*. There is a make target called "perl.gprof" for generating a gprofiled Perl executable.
  - If you have GCC 3, there is a make target called "perl.gcov" for creating a gcoved Perl executable for coverage analysis. See *perlhack*.
  - If you are on IRIX or Tru64 platforms, new profiling/debugging options have been added; see *perlhack* for more information about pixie and Third Degree.
- Guidelines of how to construct minimal Perl installations have been added to INSTALL.

- The Thread extension is now not built at all under ithreads (`Configure -Duseithreads`) because it wouldn't work anyway (the Thread extension requires being Configured with `-Duse5005threads`).

  Note that the 5.005 threads are unsupported and deprecated: if you have code written for the old threads you should migrate it to the new ithreads model.

- The Gconvert macro ($Config{d_Gconvert}) used by perl for stringifying floating-point numbers is now more picky about using sprintf %.*g rules for the conversion. Some platforms that used to use gcvt may now resort to the slower sprintf.

- The obsolete method of making a special (e.g., debugging) flavor of perl by saying

  ```
      make LIBPERL=libperld.a
  ```

  has been removed. Use -DDEBUGGING instead.

86.9.2 New Or Improved Platforms

For the list of platforms known to support Perl, see Supported Platforms in *perlport*.

- AIX dynamic loading should be now better supported.
- AIX should now work better with gcc, threads, and 64-bitness. Also the long doubles support in AIX should be better now. See *perlaix*.
- AtheOS (http://www.atheos.cx/) is a new platform.
- BeOS has been reclaimed.
- The DG/UX platform now supports 5.005-style threads. See *perldgux*.
- The DYNIX/ptx platform (also known as dynixptx) is supported at or near osvers 4.5.2.
- EBCDIC platforms (z/OS (also known as OS/390), POSIX-BC, and VM/ESA) have been regained. Many test suite tests still fail and the co-existence of Unicode and EBCDIC isn't quite settled, but the situation is much better than with Perl 5.6. See *perlos390*, *perlbs2000* (for POSIX-BC), and *perlvmesa* for more information.
- Building perl with -Duseithreads or -Duse5005threads now works under HP-UX 10.20 (previously it only worked under 10.30 or later). You will need a thread library package installed. See README.hpux. [561]
- Mac OS Classic is now supported in the mainstream source package (MacPerl has of course been available since perl 5.004 but now the source code bases of standard Perl and MacPerl have been synchronised) [561]
- Mac OS X (or Darwin) should now be able to build Perl even on HFS+ filesystems. (The case-insensitivity used to confuse the Perl build process.)
- NCR MP-RAS is now supported. [561]
- All the NetBSD specific patches (except for the installation specific ones) have been merged back to the main distribution.
- NetWare from Novell is now supported. See *perlnetware*.
- NonStop-UX is now supported. [561]
- NEC SUPER-UX is now supported.
- All the OpenBSD specific patches (except for the installation specific ones) have been merged back to the main distribution.

- Perl has been tested with the GNU pth userlevel thread package (http://www.gnu.org/software/pth/pth.html). All thread tests of Perl now work, but not without adding some yield()s to the tests, so while pth (and other userlevel thread implementations) can be considered to be "working" with Perl ithreads, keep in mind the possible non-preemptability of the underlying thread implementation.
- Stratus VOS is now supported using Perl's native build method (Configure). This is the recommended method to build Perl on VOS. The older methods, which build miniperl, are still available. See *perlvos*. [561+]
- The Amdahl UTS UNIX mainframe platform is now supported. [561]
- WinCE is now supported. See *perlce*.
- z/OS (formerly known as OS/390, formerly known as MVS OE) now has support for dynamic loading. This is not selected by default, however, you must specify -Dusedl in the arguments of Configure. [561]

86.10 Selected Bug Fixes

Numerous memory leaks and uninitialized memory accesses have been hunted down. Most importantly, anonymous subs used to leak quite a bit. [561]

- The autouse pragma didn't work for Multi::Part::Function::Names.
- caller() could cause core dumps in certain situations. Carp was sometimes affected by this problem. In particular, caller() now returns a subroutine name of `(unknown)` for subroutines that have been removed from the symbol table.
- chop(@list) in list context returned the characters chopped in reverse order. This has been reversed to be in the right order. [561]
- Configure no longer includes the DBM libraries (dbm, gdbm, db, ndbm) when building the Perl binary. The only exception to this is SunOS 4.x, which needs them. [561]
- The behaviour of non-decimal but numeric string constants such as "0x23" was platform-dependent: in some platforms that was seen as 35, in some as 0, in some as a floating point number (don't ask). This was caused by Perl's using the operating system libraries in a situation where the result of the string to number conversion is undefined: now Perl consistently handles such strings as zero in numeric contexts.
- Several debugger fixes: exit code now reflects the script exit code, condition `"0"` now treated correctly, the `d` command now checks line number, `$.` no longer gets corrupted, and all debugger output now goes correctly to the socket if RemotePort is set. [561]
- The debugger (perl5db.pl) has been modified to present a more consistent commands interface, via (CommandSet=580). perl5db.t was also added to test the changes, and as a placeholder for further tests.

  See *perldebug*.
- The debugger has a new `dumpDepth` option to control the maximum depth to which nested structures are dumped. The `x` command has been extended so that `x N EXPR` dumps out the value of *EXPR* to a depth of at most *N* levels.
- The debugger can now show lexical variables if you have the CPAN module PadWalker installed.
- The order of DESTROYs has been made more predictable.
- Perl 5.6.0 could emit spurious warnings about redefinition of dl_error() when statically building extensions into perl. This has been corrected. [561]
- *dprofpp* -R didn't work.
- `*foo{FORMAT}` now works.
- Infinity is now recognized as a number.

- UNIVERSAL::isa no longer caches methods incorrectly. (This broke the Tk extension with 5.6.0.) [561]
- Lexicals I: lexicals outside an eval "" weren't resolved correctly inside a subroutine definition inside the eval "" if they were not already referenced in the top level of the eval""ed code.
- Lexicals II: lexicals leaked at file scope into subroutines that were declared before the lexicals.
- Lexical warnings now propagating correctly between scopes and into `eval "..."`.
- `use warnings qw(FATAL all)` did not work as intended. This has been corrected. [561]
- warnings::enabled() now reports the state of $ˆW correctly if the caller isn't using lexical warnings. [561]
- Line renumbering with eval and `#line` now works. [561]
- Fixed numerous memory leaks, especially in eval "".
- Localised tied variables no longer leak memory

  ```
  use Tie::Hash;
  tie my %tied_hash => 'Tie::StdHash';

  ...

  # Used to leak memory every time local() was called;
  # in a loop, this added up.
  local($tied_hash{Foo}) = 1;
  ```

- Localised hash elements (and %ENV) are correctly unlocalised to not exist, if they didn't before they were localised.

  ```
  use Tie::Hash;
  tie my %tied_hash => 'Tie::StdHash';

  ...

  # Nothing has set the FOO element so far

  { local $tied_hash{FOO} = 'Bar' }

  # This used to print, but not now.
  print "exists!\n" if exists $tied_hash{FOO};
  ```

  As a side effect of this fix, tied hash interfaces **must** define the EXISTS and DELETE methods.

- mkdir() now ignores trailing slashes in the directory name, as mandated by POSIX.
- Some versions of glibc have a broken modfl(). This affects builds with `-Duselongdouble`. This version of Perl detects this brokenness and has a workaround for it. The glibc release 2.2.2 is known to have fixed the modfl() bug.
- Modulus of unsigned numbers now works (4063328477 % 65535 used to return 27406, instead of 27047). [561]
- Some "not a number" warnings introduced in 5.6.0 eliminated to be more compatible with 5.005. Infinity is now recognised as a number. [561]
- Numeric conversions did not recognize changes in the string value properly in certain circumstances. [561]
- Attributes (such as :shared) didn't work with our().
- our() variables will not cause bogus "Variable will not stay shared" warnings. [561]

- "our" variables of the same name declared in two sibling blocks resulted in bogus warnings about "redeclaration" of the variables. The problem has been corrected. [561]
- pack "Z" now correctly terminates the string with "\0".
- Fix password routines which in some shadow password platforms (e.g. HP-UX) caused getpwent() to return every other entry.
- The PERL5OPT environment variable (for passing command line arguments to Perl) didn't work for more than a single group of options. [561]
- PERL5OPT with embedded spaces didn't work.
- printf() no longer resets the numeric locale to "C".
- `qw(a\\b)` now parses correctly as `'a\\b'`: that is, as three characters, not four. [561]
- pos() did not return the correct value within s///ge in earlier versions. This is now handled correctly. [561]
- Printing quads (64-bit integers) with printf/sprintf now works without the q L ll prefixes (assuming you are on a quad-capable platform).
- Regular expressions on references and overloaded scalars now work. [561+]
- Right-hand side magic (GMAGIC) could in many cases such as string concatenation be invoked too many times.
- scalar() now forces scalar context even when used in void context.
- SOCKS support is now much more robust.
- sort() arguments are now compiled in the right wantarray context (they were accidentally using the context of the sort() itself). The comparison block is now run in scalar context, and the arguments to be sorted are always provided list context. [561]
- Changed the POSIX character class `[[:space:]]` to include the (very rarely used) vertical tab character. Added a new POSIX-ish character class `[[:blank:]]` which stands for horizontal whitespace (currently, the space and the tab).
- The tainting behaviour of sprintf() has been rationalized. It does not taint the result of floating point formats anymore, making the behaviour consistent with that of string interpolation. [561]
- Some cases of inconsistent taint propagation (such as within hash values) have been fixed.
- The RE engine found in Perl 5.6.0 accidentally pessimised certain kinds of simple pattern matches. These are now handled better. [561]
- Regular expression debug output (whether through `use re 'debug'` or via `-Dr`) now looks better. [561]
- Multi-line matches like `"a\nxb\n" =~ /(?!\A)x/m` were flawed. The bug has been fixed. [561]
- Use of $& could trigger a core dump under some situations. This is now avoided. [561]
- The regular expression captured submatches ($1, $2, ...) are now more consistently unset if the match fails, instead of leaving false data lying around in them. [561]
- readline() on files opened in "slurp" mode could return an extra "" (blank line) at the end in certain situations. This has been corrected. [561]
- Autovivification of symbolic references of special variables described in *perlvar* (as in `${$num}`) was accidentally disabled. This works again now. [561]
- Sys::Syslog ignored the `LOG_AUTH` constant.
- $AUTOLOAD, sort(), lock(), and spawning subprocesses in multiple threads simultaneously are now thread-safe.

- Tie::Array's SPLICE method was broken.
- Allow a read-only string on the left-hand side of a non-modifying tr///.
- If `STDERR` is tied, warnings caused by `warn` and `die` now correctly pass to it.
- Several Unicode fixes.
  - BOMs (byte order marks) at the beginning of Perl files (scripts, modules) should now be transparently skipped. UTF-16 and UCS-2 encoded Perl files should now be read correctly.
  - The character tables have been updated to Unicode 3.2.0.
  - Comparing with utf8 data does not magically upgrade non-utf8 data into utf8. (This was a problem for example if you were mixing data from I/O and Unicode data: your output might have got magically encoded as UTF-8.)
  - Generating illegal Unicode code points such as U+FFFE, or the UTF-16 surrogates, now also generates an optional warning.
  - `IsAlnum`, `IsAlpha`, and `IsWord` now match titlecase.
  - Concatenation with the `.` operator or via variable interpolation, `eq`, `substr`, `reverse`, `quotemeta`, the `x` operator, substitution with `s///`, single-quoted UTF-8, should now work.
  - The `tr///` operator now works. Note that the `tr///CU` functionality has been removed (but see pack('U0', ...)).
  - `eval "v200"` now works.
  - Perl 5.6.0 parsed m/\x{ab}/ incorrectly, leading to spurious warnings. This has been corrected. [561]
  - Zero entries were missing from the Unicode classes such as `IsDigit`.
- Large unsigned numbers (those above 2**31) could sometimes lose their unsignedness, causing bogus results in arithmetic operations. [561]
- The Perl parser has been stress tested using both random input and Markov chain input and the few found crashes and lockups have been fixed.

86.10.1 Platform Specific Changes and Fixes

- BSDI 4.*

  Perl now works on post-4.0 BSD/OSes.
- All BSDs

  Setting `$0` now works (as much as possible; see *perlvar* for details).
- Cygwin

  Numerous updates; currently synchronised with Cygwin 1.3.10.
- Previously DYNIX/ptx had problems in its Configure probe for non-blocking I/O.
- EPOC

  EPOC now better supported. See README.epoc. [561]
- FreeBSD 3.*

  Perl now works on post-3.0 FreeBSDs.
- HP-UX

  README.hpux updated; `Configure -Duse64bitall` now works; now uses HP-UX malloc instead of Perl malloc.

- IRIX

  Numerous compilation flag and hint enhancements; accidental mixing of 32-bit and 64-bit libraries (a doomed attempt) made much harder.

- Linux

  - Long doubles should now work (see INSTALL). [561]
  - Linux previously had problems related to sockaddrlen when using accept(), recvfrom() (in Perl: recv()), getpeername(), and getsockname().

- Mac OS Classic

  Compilation of the standard Perl distribution in Mac OS Classic should now work if you have the Metrowerks development environment and the missing Mac-specific toolkit bits. Contact the macperl mailing list for details.

- MPE/iX

  MPE/iX update after Perl 5.6.0. See README.mpeix. [561]

- NetBSD/threads: try installing the GNU pth (should be in the packages collection, or http://www.gnu.org/software/pth/), and Configure with -Duseithreads.

- NetBSD/sparc

  Perl now works on NetBSD/sparc.

- OS/2

  Now works with usethreads (see INSTALL). [561]

- Solaris

  64-bitness using the Sun Workshop compiler now works.

- Stratus VOS

  The native build method requires at least VOS Release 14.5.0 and GNU C++/GNU Tools 2.0.1 or later. The Perl pack function now maps overflowed values to +infinity and underflowed values to -infinity.

- Tru64 (aka Digital UNIX, aka DEC OSF/1)

  The operating system version letter now recorded in $Config{osvers}. Allow compiling with gcc (previously explicitly forbidden). Compiling with gcc still not recommended because buggy code results, even with gcc 2.95.2.

- Unicos

  Fixed various alignment problems that lead into core dumps either during build or later; no longer dies on math errors at runtime; now using full quad integers (64 bits), previously was using only 46 bit integers for speed.

- VMS

  See §86.3.5 and §86.3.6 for important changes not otherwise listed here.

  chdir() now works better despite a CRT bug; now works with MULTIPLICITY (see INSTALL); now works with Perl's malloc.

  The tainting of `%ENV` elements via `keys` or `values` was previously unimplemented. It now works as documented.

  The `waitpid` emulation has been improved. The worst bug (now fixed) was that a pid of -1 would cause a wildcard search of all processes on the system.

  POSIX-style signals are now emulated much better on VMS versions prior to 7.0.

  The `system` function and backticks operator have improved functionality and better error handling. [561]

  File access tests now use current process privileges rather than the user's default privileges, which could sometimes result in a mismatch between reported access and actual access. This improvement is only available on VMS v6.0 and later.

There is a new `kill` implementation based on `sys$sigprc` that allows older VMS systems (pre-7.0) to use `kill` to send signals rather than simply force exit. This implementation also allows later systems to call `kill` from within a signal handler.

Iterative logical name translations are now limited to 10 iterations in imitation of SHOW LOGICAL and other OpenVMS facilities.

- Windows
  - Signal handling now works better than it used to. It is now implemented using a Windows message loop, and is therefore less prone to random crashes.
  - fork() emulation is now more robust, but still continues to have a few esoteric bugs and caveats. See *perlfork* for details. [561+]
  - A failed (pseudo)fork now returns undef and sets errno to EAGAIN. [561]
  - The following modules now work on Windows:

    ```
    ExtUtils::Embed         [561]
    IO::Pipe
    IO::Poll
    Net::Ping
    ```

  - IO::File::new_tmpfile() is no longer limited to 32767 invocations per-process.
  - Better chdir() return value for a non-existent directory.
  - Compiling perl using the 64-bit Platform SDK tools is now supported.
  - The Win32::SetChildShowWindow() builtin can be used to control the visibility of windows created by child processes. See *Win32* for details.
  - Non-blocking waits for child processes (or pseudo-processes) are supported via `waitpid($pid, &POSIX::WNOHANG)`.
  - The behavior of system() with multiple arguments has been rationalized. Each unquoted argument will be automatically quoted to protect whitespace, and any existing whitespace in the arguments will be preserved. This improves the portability of system(@args) by avoiding the need for Windows `cmd` shell specific quoting in perl programs.

    Note that this means that some scripts that may have relied on earlier buggy behavior may no longer work correctly. For example, `system("nmake /nologo", @args)` will now attempt to run the file `nmake /nologo` and will fail when such a file isn't found. On the other hand, perl will now execute code such as `system("c:/Program Files/MyApp/foo.exe", @args)` correctly.
  - The perl header files no longer suppress common warnings from the Microsoft Visual C++ compiler. This means that additional warnings may now show up when compiling XS code.
  - Borland C++ v5.5 is now a supported compiler that can build Perl. However, the generated binaries continue to be incompatible with those generated by the other supported compilers (GCC and Visual C++). [561]
  - Duping socket handles with open(F, ">&MYSOCK") now works under Windows 9x. [561]
  - Current directory entries in %ENV are now correctly propagated to child processes. [561]
  - New %ENV entries now propagate to subprocesses. [561]
  - Win32::GetCwd() correctly returns C:\ instead of C: when at the drive root. Other bugs in chdir() and Cwd::cwd() have also been fixed. [561]
  - The makefiles now default to the features enabled in ActiveState ActivePerl (a popular Win32 binary distribution). [561]
  - HTML files will now be installed in c:\perl\html instead of c:\perl\lib\pod\html
  - REG_EXPAND_SZ keys are now allowed in registry settings used by perl. [561]
  - Can now send() from all threads, not just the first one. [561]
  - ExtUtils::MakeMaker now uses $ENV{LIB} to search for libraries. [561]
  - Less stack reserved per thread so that more threads can run concurrently. (Still 16M per thread.) [561]

- `File::Spec->tmpdir()` now prefers C:/temp over /tmp (works better when perl is running as service).
- Better UNC path handling under ithreads. [561]
- wait(), waitpid(), and backticks now return the correct exit status under Windows 9x. [561]
- A socket handle leak in accept() has been fixed. [561]

86.11 New or Changed Diagnostics

Please see *perldiag* for more details.

- Ambiguous range in the transliteration operator (like a-z-9) now gives a warning.
- chdir("") and chdir(undef) now give a deprecation warning because they cause a possible unintentional chdir to the home directory. Say chdir() if you really mean that.
- Two new debugging options have been added: if you have compiled your Perl with debugging, you can use the -DT [561] and -DR options to trace tokenising and to add reference counts to displaying variables, respectively.
- The lexical warnings category "deprecated" is no longer a sub-category of the "syntax" category. It is now a top-level category in its own right.
- Unadorned dump() will now give a warning suggesting to use explicit CORE::dump() if that's what really is meant.
- The "Unrecognized escape" warning has been extended to include `\8`, `\9`, and `\_`. There is no need to escape any of the `\w` characters.
- All regular expression compilation error messages are now hopefully easier to understand both because the error message now comes before the failed regex and because the point of failure is now clearly marked by a `<- HERE` marker.
- Various I/O (and socket) functions like binmode(), close(), and so forth now more consistently warn if they are used illogically either on a yet unopened or on an already closed filehandle (or socket).
- Using lstat() on a filehandle now gives a warning. (It's a non-sensical thing to do.)
- The `-M` and `-m` options now warn if you didn't supply the module name.
- If you in `use` specify a required minimum version, modules matching the name and but not defining a $VERSION will cause a fatal failure.
- Using negative offset for vec() in lvalue context is now a warnable offense.
- Odd number of arguments to overload::constant now elicits a warning.
- Odd number of elements in anonymous hash now elicits a warning.
- The various "opened only for", "on closed", "never opened" warnings drop the `main::` prefix for filehandles in the `main` package, for example `STDIN` instead of `main::STDIN`.
- Subroutine prototypes are now checked more carefully, you may get warnings for example if you have used non-prototype characters.
- If an attempt to use a (non-blessed) reference as an array index is made, a warning is given.
- `push @a;` and `unshift @a;` (with no values to push or unshift) now give a warning. This may be a problem for generated and evaled code.
- If you try to `pack` in *perlfunc* a number less than 0 or larger than 255 using the `"C"` format you will get an optional warning. Similarly for the `"c"` format and a number less than -128 or more than 127.
- pack `P` format now demands an explicit size.

- unpack `w` now warns of unterminated compressed integers.
- Warnings relating to the use of PerlIO have been added.
- Certain regex modifiers such as `(?o)` make sense only if applied to the entire regex. You will get an optional warning if you try to do otherwise.
- Variable length lookbehind has not yet been implemented, trying to use it will tell that.
- Using arrays or hashes as references (e.g. `%foo->{bar}` has been deprecated for a while. Now you will get an optional warning.
- Warnings relating to the use of the new restricted hashes feature have been added.
- Self-ties of arrays and hashes are not supported and fatal errors will happen even at an attempt to do so.
- Using `sort` in scalar context now issues an optional warning. This didn't do anything useful, as the sort was not performed.
- Using the /g modifier in split() is meaningless and will cause a warning.
- Using splice() past the end of an array now causes a warning.
- Malformed Unicode encodings (UTF-8 and UTF-16) cause a lot of warnings, ad doestrying to use UTF-16 surrogates (which are unimplemented).
- Trying to use Unicode characters on an I/O stream without marking the stream's encoding (using open() or binmode()) will cause "Wide character" warnings.
- Use of v-strings in use/require causes a (backward) portability warning.
- Warnings relating to the use interpreter threads and their shared data have been added.

86.12 Changed Internals

- PerlIO is now the default.
- perlapi.pod (a companion to perlguts) now attempts to document the internal API.
- You can now build a really minimal perl called microperl. Building microperl does not require even running Configure; `make -f Makefile.micro` should be enough. Beware: microperl makes many assumptions, some of which may be too bold; the resulting executable may crash or otherwise misbehave in wondrous ways. For careful hackers only.
- Added rsignal(), whichsig(), do_join(), op_clear, op_null, ptr_table_clear(), ptr_table_free(), sv_setref_uv(), and several UTF-8 interfaces to the publicised API. For the full list of the available APIs see *perlapi*.
- Made possible to propagate customised exceptions via croak()ing.
- Now xsubs can have attributes just like subs. (Well, at least the built-in attributes.)
- dTHR and djSP have been obsoleted; the former removed (because it's a no-op) and the latter replaced with dSP.
- PERL_OBJECT has been completely removed.
- The MAGIC constants (e.g. `'P'`) have been macrofied (e.g. `PERL_MAGIC_TIED`) for better source code readability and maintainability.
- The regex compiler now maintains a structure that identifies nodes in the compiled bytecode with the corresponding syntactic features of the original regex expression. The information is attached to the new `offsets` member of the `struct regexp`. See *perldebguts* for more complete information.

- The C code has been made much more `gcc -Wall` clean. Some warning messages still remain in some platforms, so if you are compiling with gcc you may see some warnings about dubious practices. The warnings are being worked on.
- *perly.c*, *sv.c*, and *sv.h* have now been extensively commented.
- Documentation on how to use the Perl source repository has been added to *Porting/repository.pod*.
- There are now several profiling make targets.

86.13 Security Vulnerability Closed [561]

(This change was already made in 5.7.0 but bears repeating here.) (5.7.0 came out before 5.6.1: the development branch 5.7 released earlier than the maintenance branch 5.6)

A potential security vulnerability in the optional suidperl component of Perl was identified in August 2000. suidperl is neither built nor installed by default. As of November 2001 the only known vulnerable platform is Linux, most likely all Linux distributions. CERT and various vendors and distributors have been alerted about the vulnerability. See http://www.cpan.org/src/5.0/sperl-2000-08-05/sperl-2000-08-05.txt for more information.

The problem was caused by Perl trying to report a suspected security exploit attempt using an external program, /bin/mail. On Linux platforms the /bin/mail program had an undocumented feature which when combined with suidperl gave access to a root shell, resulting in a serious compromise instead of reporting the exploit attempt. If you don't have /bin/mail, or if you have 'safe setuid scripts', or if suidperl is not installed, you are safe.

The exploit attempt reporting feature has been completely removed from Perl 5.8.0 (and the maintenance release 5.6.1, and it was removed also from all the Perl 5.7 releases), so that particular vulnerability isn't there anymore. However, further security vulnerabilities are, unfortunately, always possible. The suidperl functionality is most probably going to be removed in Perl 5.10. In any case, suidperl should only be used by security experts who know exactly what they are doing and why they are using suidperl instead of some other solution such as sudo (see http://www.courtesan.com/sudo/).

86.14 New Tests

Several new tests have been added, especially for the *lib* and *ext* subsections. There are now about 69 000 individual tests (spread over about 700 test scripts), in the regression suite (5.6.1 has about 11 700 tests, in 258 test scripts) The exact numbers depend on the platform and Perl configuration used. Many of the new tests are of course introduced by the new modules, but still in general Perl is now more thoroughly tested.

Because of the large number of tests, running the regression suite will take considerably longer time than it used to: expect the suite to take up to 4-5 times longer to run than in perl 5.6. On a really fast machine you can hope to finish the suite in about 6-8 minutes (wallclock time).

The tests are now reported in a different order than in earlier Perls. (This happens because the test scripts from under t/lib have been moved to be closer to the library/extension they are testing.)

86.15 Known Problems

86.15.1 The Compiler Suite Is Still Very Experimental

The compiler suite is slowly getting better but it continues to be highly experimental. Use in production environments is discouraged.

86.15.2 Localising Tied Arrays and Hashes Is Broken

```
    local %tied_array;
```

doesn't work as one would expect: the old value is restored incorrectly. This will be changed in a future release, but we don't know yet what the new semantics will exactly be. In any case, the change will break existing code that relies on the current (ill-defined) semantics, so just avoid doing this in general.

86.15.3 Building Extensions Can Fail Because Of Largefiles

Some extensions like mod_perl are known to have issues with 'largefiles', a change brought by Perl 5.6.0 in which file offsets default to 64 bits wide, where supported. Modules may fail to compile at all, or they may compile and work incorrectly. Currently, there is no good solution for the problem, but Configure now provides appropriate non-largefile ccflags, ldflags, libswanted, and libs in the %Config hash (e.g., $Config{ccflags_nolargefiles}) so the extensions that are having problems can try configuring themselves without the largefileness. This is admittedly not a clean solution, and the solution may not even work at all. One potential failure is whether one can (or, if one can, whether it's a good idea to) link together at all binaries with different ideas about file offsets; all this is platform-dependent.

86.15.4 Modifying $ _ Inside for(..)

```
   for (1..5) { $_++ }
```

works without complaint. It shouldn't. (You should be able to modify only lvalue elements inside the loops.) You can see the correct behaviour by replacing the 1..5 with 1, 2, 3, 4, 5.

86.15.5 mod_perl 1.26 Doesn't Build With Threaded Perl

Use mod_perl 1.27 or higher.

86.15.6 lib/ftmp-security tests warn 'system possibly insecure'

Don't panic. Read the 'make test' section of INSTALL instead.

86.15.7 libwww-perl (LWP) fails base/date #51

Use libwww-perl 5.65 or later.

86.15.8 PDL failing some tests

Use PDL 2.3.4 or later.

86.15.9 Perl_get_sv

You may get errors like 'Undefined symbol "Perl_get_sv"' or "can't resolve symbol 'Perl_get_sv'", or the symbol may be "Perl_sv_2pv". This probably means that you are trying to use an older shared Perl library (or extensions linked with such) with Perl 5.8.0 executable. Perl used to have such a subroutine, but that is no more the case. Check your shared library path, and any shared Perl libraries in those directories.

Sometimes this problem may also indicate a partial Perl 5.8.0 installation, see §86.16.15 for an example and how to deal with it.

86.15.10 Self-tying Problems

Self-tying of arrays and hashes is broken in rather deep and hard-to-fix ways. As a stop-gap measure to avoid people from getting frustrated at the mysterious results (core dumps, most often), it is forbidden for now (you will get a fatal error even from an attempt).

A change to self-tying of globs has caused them to be recursively referenced (see: Two-Phased Garbage Collection in *perlobj*). You will now need an explicit untie to destroy a self-tied glob. This behaviour may be fixed at a later date.

Self-tying of scalars and IO thingies works.

86.15.11 ext/threads/t/libc

If this test fails, it indicates that your libc (C library) is not threadsafe. This particular test stress tests the localtime() call to find out whether it is threadsafe. See *perlthrtut* for more information.

86.15.12 Failure of Thread (5.005-style) tests

Note that support for 5.005-style threading is deprecated, experimental and practically unsupported. In 5.10, it is expected to be removed. You should migrate your code to ithreads.

The following tests are known to fail due to fundamental problems in the 5.005 threading implementation. These are not new failures–Perl 5.005_0x has the same bugs, but didn't have these tests.

```
../ext/B/t/xref.t                  255 65280    14   12  85.71%  3-14
../ext/List/Util/t/first.t         255 65280     7    4  57.14%  2 5-7
../lib/English.t                     2   512    54    2   3.70%  2-3
../lib/FileCache.t                               5    1  20.00%  5
../lib/Filter/Simple/t/data.t                    6    3  50.00%  1-3
../lib/Filter/Simple/t/filter_only.              9    3  33.33%  1-2 5
../lib/Math/BigInt/t/bare_mbf.t               1627    4   0.25%  8 11 1626-1627
../lib/Math/BigInt/t/bigfltpm.t               1629    4   0.25%  10 13 1628-
                                                                 1629
../lib/Math/BigInt/t/sub_mbf.t                1633    4   0.24%  8 11 1632-1633
../lib/Math/BigInt/t/with_sub.t               1628    4   0.25%  9 12 1627-1628
../lib/Tie/File/t/31_autodefer.t   255 65280    65   32  49.23%  34-65
../lib/autouse.t                                10    1  10.00%  4
op/flip.t                                       15    1   6.67%  15
```

These failures are unlikely to get fixed as 5.005-style threads are considered fundamentally broken. (Basically what happens is that competing threads can corrupt shared global state, one good example being regular expression engine's state.)

86.15.13 Timing problems

The following tests may fail intermittently because of timing problems, for example if the system is heavily loaded.

```
    t/op/alarm.t
    ext/Time/HiRes/HiRes.t
    lib/Benchmark.t
    lib/Memoize/t/expmod_t.t
    lib/Memoize/t/speed.t
```

In case of failure please try running them manually, for example

```
    ./perl -Ilib ext/Time/HiRes/HiRes.t
```

86.15.14 Tied/Magical Array/Hash Elements Do Not Autovivify

For normal arrays `$foo = \$bar[1]` will assign `undef` to `$bar[1]` (assuming that it didn't exist before), but for tied/magical arrays and hashes such autovivification does not happen because there is currently no way to catch the reference creation. The same problem affects slicing over non-existent indices/keys of a tied/magical array/hash.

86.15.15 Unicode in package/class and subroutine names does not work

One can have Unicode in identifier names, but not in package/class or subroutine names. While some limited functionality towards this does exist as of Perl 5.8.0, that is more accidental than designed; use of Unicode for the said purposes is unsupported.

One reason of this unfinishedness is its (currently) inherent unportability: since both package names and subroutine names may need to be mapped to file and directory names, the Unicode capability of the filesystem becomes important– and there unfortunately aren't portable answers.

86.16 Platform Specific Problems

86.16.1 AIX

- If using the AIX native make command, instead of just "make" issue "make all". In some setups the former has been known to spuriously also try to run "make install". Alternatively, you may want to use GNU make.
- In AIX 4.2, Perl extensions that use C++ functions that use statics may have problems in that the statics are not getting initialized. In newer AIX releases, this has been solved by linking Perl with the libC_r library, but unfortunately in AIX 4.2 the said library has an obscure bug where the various functions related to time (such as time() and gettimeofday()) return broken values, and therefore in AIX 4.2 Perl is not linked against libC_r.
- vac 5.0.0.0 May Produce Buggy Code For Perl

  The AIX C compiler vac version 5.0.0.0 may produce buggy code, resulting in a few random tests failing when run as part of "make test", but when the failing tests are run by hand, they succeed. We suggest upgrading to at least vac version 5.0.1.0, that has been known to compile Perl correctly. "lslpp -L|grep vac.C" will tell you the vac version. See README.aix.
- If building threaded Perl, you may get compilation warning from pp_sys.c:

  ```
  "pp_sys.c", line 4651.39: 1506-280 (W) Function argument assignment between types "unsigned char
  ```

  This is harmless; it is caused by the getnetbyaddr() and getnetbyaddr_r() having slightly different types for their first argument.

86.16.2 Alpha systems with old gccs fail several tests

If you see op/pack, op/pat, op/regexp, or ext/Storable tests failing in a Linux/alpha or *BSD/Alpha, it's probably time to upgrade your gcc. gccs prior to 2.95.3 are definitely not good enough, and gcc 3.1 may be even better. (RedHat Linux/alpha with gcc 3.1 reported no problems, as did Linux 2.4.18 with gcc 2.95.4.) (In Tru64, it is preferable to use the bundled C compiler.)

86.16.3 AmigaOS

Perl 5.8.0 doesn't build in AmigaOS. It broke at some point during the ithreads work and we could not find Amiga experts to unbreak the problems. Perl 5.6.1 still works for AmigaOS (as does the the 5.7.2 development release).

86.16.4 BeOS

The following tests fail on 5.8.0 Perl in BeOS Personal 5.03:

```
t/op/lfs............................FAILED at test 17
t/op/magic..........................FAILED at test 24
ext/Fcntl/t/syslfs..................FAILED at test 17
ext/File/Glob/t/basic...............FAILED at test 3
ext/POSIX/t/sigaction...............FAILED at test 13
ext/POSIX/t/waitpid.................FAILED at test 1
```

See *perlbeos* (README.beos) for more details.

86.16.5 Cygwin "unable to remap"

For example when building the Tk extension for Cygwin, you may get an error message saying "unable to remap". This is known problem with Cygwin, and a workaround is detailed in here:
http://sources.redhat.com/ml/cygwin/2001-12/msg00894.html

86.16.6 Cygwin ndbm tests fail on FAT

One can build but not install (or test the build of) the NDBM_File on FAT filesystems. Installation (or build) on NTFS works fine. If one attempts the test on a FAT install (or build) the following failures are expected:

```
../ext/NDBM_File/ndbm.t       13  3328    71   59  83.10%  1-2 4 16-71
../ext/ODBM_File/odbm.t      255 65280    ??   ??       %  ??
../lib/AnyDBM_File.t           2   512    12    2  16.67%  1 4
../lib/Memoize/t/errors.t      0   139    11    5  45.45%  7-11
../lib/Memoize/t/tie_ndbm.t   13  3328     4    4 100.00%  1-4
run/fresh_perl.t                          97    1   1.03%  91
```

NDBM_File fails and ODBM_File just coredumps.

If you intend to run only on FAT (or if using AnyDBM_File on FAT), run Configure with the -Ui_ndbm and -Ui_dbm options to prevent NDBM_File and ODBM_File being built.

86.16.7 DJGPP Failures

```
t/op/stat...........................FAILED at test 29
lib/File/Find/t/find................FAILED at test 1
lib/File/Find/t/taint...............FAILED at test 1
lib/h2xs............................FAILED at test 15
lib/Pod/t/eol.......................FAILED at test 1
lib/Test/Harness/t/strap-analyze....FAILED at test 8
lib/Test/Harness/t/test-harness.....FAILED at test 23
lib/Test/Simple/t/exit..............FAILED at test 1
```

The above failures are known as of 5.8.0 with native builds with long filenames, but there are a few more if running under dosemu because of limitations (and maybe bugs) of dosemu:

```
t/comp/cpp..........................FAILED at test 3
t/op/inccode........................(crash)
```

and a few lib/ExtUtils tests, and several hundred Encode/t/Aliases.t failures that work fine with long filenames. So you really might prefer native builds and long filenames.

86.16.8 FreeBSD built with ithreads coredumps reading large directories

This is a known bug in FreeBSD 4.5's readdir_r(), it has been fixed in FreeBSD 4.6 (see *perlfreebsd* (README.freebsd)).

86.16.9 FreeBSD Failing locale Test 117 For ISO 8859-15 Locales

The ISO 8859-15 locales may fail the locale test 117 in FreeBSD. This is caused by the characters \xFF (y with diaeresis) and \xBE (Y with diaeresis) not behaving correctly when being matched case-insensitively. Apparently this problem has been fixed in the latest FreeBSD releases. (http://www.freebsd.org/cgi/query-pr.cgi?pr=34308)

86.16.10 IRIX fails ext/List/Util/t/shuffle.t or Digest::MD5

IRIX with MIPSpro 7.3.1.2m or 7.3.1.3m compiler may fail the List::Util test ext/List/Util/t/shuffle.t by dumping core. This seems to be a compiler error since if compiled with gcc no core dump ensues, and no failures have been seen on the said test on any other platform.

Similarly, building the Digest::MD5 extension has been known to fail with "*** Termination code 139 (bu21)".

The cure is to drop optimization level (Configure -Doptimize=-O2).

86.16.11 HP-UX lib/posix Subtest 9 Fails When LP64-Configured

If perl is configured with -Duse64bitall, the successful result of the subtest 10 of lib/posix may arrive before the successful result of the subtest 9, which confuses the test harness so much that it thinks the subtest 9 failed.

86.16.12 Linux with glibc 2.2.5 fails t/op/int subtest #6 with -Duse64bitint

This is a known bug in the glibc 2.2.5 with long long integers. (http://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=65612)

86.16.13 Linux With Sfio Fails op/misc Test 48

No known fix.

86.16.14 Mac OS X

Please remember to set your environment variable LC_ALL to "C" (setenv LC_ALL C) before running "make test" to avoid a lot of warnings about the broken locales of Mac OS X.

The following tests are known to fail in Mac OS X 10.1.5 because of buggy (old) implementations of Berkeley DB included in Mac OS X:

```
Failed Test                 Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------
../ext/DB_File/t/db-btree.t    0    11    ??   ??       %  ??
../ext/DB_File/t/db-recno.t              149    3   2.01%  61 63 65
```

If you are building on a UFS partition, you will also probably see t/op/stat.t subtest #9 fail. This is caused by Darwin's UFS not supporting inode change time.

Also the ext/POSIX/t/posix.t subtest #10 fails but it is skipped for now because the failure is Apple's fault, not Perl's (blocked signals are lost).

If you Configure with ithreads, ext/threads/t/libc.t will fail. Again, this is not Perl's fault– the libc of Mac OS X is not threadsafe (in this particular test, the localtime() call is found to be threadunsafe.)

86.16.15 Mac OS X dyld undefined symbols

If after installing Perl 5.8.0 you are getting warnings about missing symbols, for example

```
    dyld: perl Undefined symbols
    _perl_sv_2pv
    _perl_get_sv
```

you probably have an old pre-Perl-5.8.0 installation (or parts of one) in /Library/Perl (the undefined symbols used to exist in pre-5.8.0 Perls). It seems that for some reason "make install" doesn't always completely overwrite the files in /Library/Perl. You can move the old Perl shared library out of the way like this:

```
    cd /Library/Perl/darwin/CORE
    mv libperl.dylib libperlold.dylib
```

and then reissue "make install". Note that the above of course is extremely disruptive for anything using the /usr/local/bin/perl. If that doesn't help, you may have to try removing all the .bundle files from beneath /Library/Perl, and again "make install"-ing.

86.16.16 OS/2 Test Failures

The following tests are known to fail on OS/2 (for clarity only the failures are shown, not the full error messages):

```
../lib/ExtUtils/t/Mkbootstrap.t    1   256    18    1   5.56%  8
../lib/ExtUtils/t/Packlist.t       1   256    34    1   2.94%  17
../lib/ExtUtils/t/basic.t          1   256    17    1   5.88%  14
lib/os2_process.t                  2   512   227    2   0.88%  174 209
lib/os2_process_kid.t                        227    2   0.88%  174 209
lib/rx_cmprt.t                   255 65280    18    3  16.67%  16-18
```

86.16.17 op/sprintf tests 91, 129, and 130

The op/sprintf tests 91, 129, and 130 are known to fail on some platforms. Examples include any platform using sfio, and Compaq/Tandem's NonStop-UX.

Test 91 is known to fail on QNX6 (nto), because `sprintf '%e',0` incorrectly produces `0.000000e+0` instead of `0.000000e+00`.

For tests 129 and 130, the failing platforms do not comply with the ANSI C Standard: lines 19ff on page 134 of ANSI X3.159 1989, to be exact. (They produce something other than "1" and "-1" when formatting 0.6 and -0.6 using the printf format "%.0f"; most often, they produce "0" and "-0".)

86.16.18 SCO

The socketpair tests are known to be unhappy in SCO 3.2v5.0.4:

```
ext/Socket/socketpair.t...............FAILED tests 15-45
```

86.16.19 Solaris 2.5

In case you are still using Solaris 2.5 (aka SunOS 5.5), you may experience failures (the test core dumping) in lib/locale.t. The suggested cure is to upgrade your Solaris.

86.16.20 Solaris x86 Fails Tests With -Duse64bitint

The following tests are known to fail in Solaris x86 with Perl configured to use 64 bit integers:

```
ext/Data/Dumper/t/dumper............FAILED at test 268
ext/Devel/Peek/Peek.................FAILED at test 7
```

86.16.21 SUPER-UX (NEC SX)

The following tests are known to fail on SUPER-UX:

```
op/64bitint..........................FAILED tests 29-30, 32-33, 35-36
op/arith.............................FAILED tests 128-130
op/pack..............................FAILED tests 25-5625
op/pow...............................
op/taint.............................# msgsnd failed
../ext/IO/lib/IO/t/io_poll...........FAILED tests 3-4
../ext/IPC/SysV/ipcsysv..............FAILED tests 2, 5-6
../ext/IPC/SysV/t/msg................FAILED tests 2, 4-6
../ext/Socket/socketpair.............FAILED tests 12
../lib/IPC/SysV......................FAILED tests 2, 5-6
../lib/warnings......................FAILED tests 115-116, 118-119
```

The op/pack failure ("Cannot compress negative numbers at op/pack.t line 126") is serious but as of yet unsolved. It points at some problems with the signedness handling of the C compiler, as do the 64bitint, arith, and pow failures. Most of the rest point at problems with SysV IPC.

86.16.22 Term::ReadKey not working on Win32

Use Term::ReadKey 2.20 or later.

86.16.23 UNICOS/mk

- During Configure, the test

  ```
  Guessing which symbols your C compiler and preprocessor define...
  ```

  will probably fail with error messages like

  ```
  CC-20 cc: ERROR File = try.c, Line = 3
    The identifier "bad" is undefined.

    bad switch yylook 79bad switch yylook 79bad switch yylook 79bad switch yylook 79#ifdef A29K
    ^

  CC-65 cc: ERROR File = try.c, Line = 3
    A semicolon is expected at this point.
  ```

  This is caused by a bug in the awk utility of UNICOS/mk. You can ignore the error, but it does cause a slight problem: you cannot fully benefit from the h2ph utility (see *h2ph*) that can be used to convert C headers to Perl libraries, mainly used to be able to access from Perl the constants defined using C preprocessor, cpp. Because of the above error, parts of the converted headers will be invisible. Luckily, these days the need for h2ph is rare.

- If building Perl with interpreter threads (ithreads), the getgrent(), getgrnam(), and getgrgid() functions cannot return the list of the group members due to a bug in the multithreaded support of UNICOS/mk. What this means is that in list context the functions will return only three values, not four.

86.16.24 UTS

There are a few known test failures, see *perluts* (README.uts).

86.16.25 VOS (Stratus)

When Perl is built using the native build process on VOS Release 14.5.0 and GNU C++/GNU Tools 2.0.1, all attempted tests either pass or result in TODO (ignored) failures.

86.16.26 VMS

There should be no reported test failures with a default configuration, though there are a number of tests marked TODO that point to areas needing further debugging and/or porting work.

86.16.27 Win32

In multi-CPU boxes, there are some problems with the I/O buffering: some output may appear twice.

86.16.28 XML::Parser not working

Use XML::Parser 2.31 or later.

86.16.29 z/OS (OS/390)

z/OS has rather many test failures but the situation is actually much better than it was in 5.6.0; it's just that so many new modules and tests have been added.

```
Failed Test                   Stat Wstat Total Fail  Failed  List of Failed
---------------------------------------------------------------------------
../ext/Data/Dumper/t/dumper.t              357    8   2.24%  311 314 325 327
                                                             331 333 337 339
../ext/IO/lib/IO/t/io_unix.t                 5    4  80.00%  2-5
../ext/Storable/t/downgrade.t   12  3072   169   12   7.10%  14-15 46-47 78-79
                                                             110-111 150 161
../lib/ExtUtils/t/Constant.t   121 30976    48   48 100.00%  1-48
../lib/ExtUtils/t/Embed.t                    9    9 100.00%  1-9
op/pat.t                                   922    7   0.76%  665 776 785 832-
                                                             834 845
op/sprintf.t                               224    3   1.34%  98 100 136
op/tr.t                                     97    5   5.15%  63 71-74
uni/fold.t                                 780    6   0.77%  61 169 196 661
                                                             710-711
```

The failures in dumper.t and downgrade.t are problems in the tests, those in io_unix and sprintf are problems in the USS (UDP sockets and printf formats). The pat, tr, and fold failures are genuine Perl problems caused by EBCDIC (and in the pat and fold cases, combining that with Unicode). The Constant and Embed are probably problems in the tests (since they test Perl's ability to build extensions, and that seems to be working reasonably well.)

86.16.30 Unicode Support on EBCDIC Still Spotty

Though mostly working, Unicode support still has problem spots on EBCDIC platforms. One such known spot are the `\p{}` and `\P{}` regular expression constructs for code points less than 256: the `pP` are testing for Unicode code points, not knowing about EBCDIC.

86.16.31 Seen In Perl 5.7 But Gone Now

`Time::Piece` (previously known as `Time::Object`) was removed because it was felt that it didn't have enough value in it to be a core module. It is still a useful module, though, and is available from the CPAN.

Perl 5.8 unfortunately does not build anymore on AmigaOS; this broke accidentally at some point. Since there are not that many Amiga developers available, we could not get this fixed and tested in time for 5.8.0. Perl 5.6.1 still works for AmigaOS (as does the the 5.7.2 development release).

The `PerlIO::Scalar` and `PerlIO::Via` (capitalised) were renamed as `PerlIO::scalar` and `PerlIO::via` (all lowercase) just before 5.8.0. The main rationale was to have all core PerlIO layers to have all lowercase names. The "plugins" are named as usual, for example `PerlIO::via::QuotedPrint`.

The `threads::shared::queue` and `threads::shared::semaphore` were renamed as `Thread::Queue` and `Thread::Semaphore` just before 5.8.0. The main rationale was to have thread modules to obey normal naming, `Thread::` (the `threads` and `threads::shared` themselves are more pragma-like, they affect compile-time, so they stay lowercase).

86.17 Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup and the perl bug database at http://bugs.perl.org/ . There may also be information at http://www.perl.com/ , the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to perlbug@perl.org to be analysed by the Perl porting team.

86.18 SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

86.19 HISTORY

Written by Jarkko Hietaniemi <*jhi@iki.fi*>.

Chapter 87

perl573delta

What's new for perl v5.7.3

87.1 DESCRIPTION

This document describes differences between the 5.7.2 release and the 5.7.3 release.

(To view the differences between the 5.6.0 release and the 5.7.0 release, see *perl570delta*. To view the differences between the 5.7.0 release and the 5.7.1 release, see *perl571delta*. To view the differences between the 5.7.1 release and the 5.7.2 release, see *perl572delta*.)

87.2 Changes

This is just a selected list of some of the more notable changes. The numbers refer to the Perl repository change numbers; see *Changes58* (or *Changes* in Perl 5.8.1). In addition to these changes, lots of work took place in integrating threads, PerlIO, and Unicode; general code cleanup; and last but not least porting to non-UNIX lands such as Win32, VMS, Cygwin, DJGPP, VOS, MacOS Classic, and EBCDIC.

1. add LC_MESSAGES to POSIX :locale_h export tag
2. add DEL to [:cntrl:]
3. make h2ph understand constants like 1234L and 5678LL
4. Win32: fix bugs in handling of the virtualized environment
5. fix a bug in the security taint checking of open()
6. make perl fork() safe even on platforms that don't have pthread_atfork()
7. make switching optimization and debugging levels during Perl builds easier via the OPTIMIZE environment variable
8. make split()'s unused captures to be undef, not ''
9. Search::Dict: allow transforming lines before comparing
10. allow installing extra modules or bundles when building Perl
11. add -Wall in cflags when compiling with gcc to weed out dubious C practices
12. pluggable optimizer
13. WinCE: integrate the port

14. Win32: 4-arg select was broken
15. introduce the perlivp utility for verifying the Perl installation (IVP = Installation Verification Procedure)
16. rename lib/unicode to lib/unicore to avoid case-insensitivity problems with lib/Unicode
17. remove Time::Piece
18. document that use utf8 is not the right way most of the time
19. allow builing perl with -DUSE_UTF8_SCRIPTS which makes UTF-8 the default script encoding (not the default since that would break all scripts having legacy eight-bit data in them)
20. division preserving 64-bit integers
21. document the coderef-in-@INC feature
22. modulo (%) preserving 64-bit integers
23. update to Unicode 3.1.1
24. add the \[$@%&*] prototype support
25. oct() and hex() in glorious 64 bit
26. Class::Struct: allow recursive classes
27. fix unpack U to be the reverse of pack U
28. VMS: waitpid enhancements
29. unpack("Z*Z*", pack("Z*Z*", ..)) was broken
30. Devel::Peek: display UTF-8 SVs also also as \x{...}
31. Data::Dumper: option to sort hashes
32. add perlpodspec
33. threadsafe DynaLoader, re, Opcode, File::Glob, and B
34. support BeOS better
35. read-only hashes (user-level interface is Hash::Util)
36. add Devel::PPPort
37. add the sort pragma
38. VMS: fix perl -P
39. add perlpacktut
40. SUPER-UX: add hints file
41. Win32: non-blocking waitpid(-1,WNOHANG)
42. introduce the -t option for gentler taint checking
43. add the if pragma
44. implement IV/UV/NV/long double un/packing with j/J/F/D
45. document the new taint behaviour of exec LIST and system LIST

87.3 Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup and the perl bug database at http://bugs.perl.org. There may also be information at http://www.perl.com/, the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to perlbug@perl.org to be analysed by the Perl porting team.

87.4 SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

87.5 HISTORY

Written by Jarkko Hietaniemi <*jhi@iki.fi*>, with many contributions from The Perl Porters and Perl Users submitting feedback and patches.

Send omissions or corrections to <*perlbug@perl.org*>.

Chapter 88

perl572delta

What's new for perl v5.7.2

88.1 DESCRIPTION

This document describes differences between the 5.7.1 release and the 5.7.2 release.

(To view the differences between the 5.6.0 release and the 5.7.0 release, see *perl570delta*. To view the differences between the 5.7.0 release and the 5.7.1 release, see *perl571delta*.)

88.2 Security Vulnerability Closed

(This change was already made in 5.7.0 but bears repeating here.)

A security vulnerability affecting all Perl versions prior to 5.6.1 was found in August 2000. The vulnerability does not affect default installations and as far as is known affects only the Linux platform.

You should upgrade your Perl to 5.6.1 as soon as possible. Patches for earlier releases exist but using the patches require full recompilation from the source code anyway, so 5.6.1 is your best choice.

See http://www.cpan.org/src/5.0/sperl-2000-08-05/sperl-2000-08-05.txt for more information.

88.3 Incompatible Changes

88.3.1 64-bit platforms and malloc

If your pointers are 64 bits wide, the Perl malloc is no more being used because it simply does not work with 8-byte pointers. Also, usually the system malloc on such platforms are much better optimized for such large memory models than the Perl malloc.

88.3.2 AIX Dynaloading

The AIX dynaloading now uses in AIX releases 4.3 and newer the native dlopen interface of AIX instead of the old emulated interface. This change will probably break backward compatibility with compiled modules. The change was made to make Perl more compliant with other applications like modperl which are using the AIX native interface.

88.3.3 Socket Extension Dynamic in VMS

The Socket extension is now dynamically loaded instead of being statically built in. This may or may not be a problem with ancient TCP/IP stacks of VMS: we do not know since we weren't able to test Perl in such configurations.

88.3.4 Different Definition of the Unicode Character Classes \p{In...}

As suggested by the Unicode consortium, the Unicode character classes now prefer *scripts* as opposed to *blocks* (as defined by Unicode); in Perl, when the `\p{In....}` and the `\p{In....}` regular expression constructs are used. This has changed the definition of some of those character classes.

The difference between scripts and blocks is that scripts are the glyphs used by a language or a group of languages, while the blocks are more artificial groupings of 256 characters based on the Unicode numbering.

In general this change results in more inclusive Unicode character classes, but changes to the other direction also do take place: for example while the script `Latin` includes all the Latin characters and their various diacritic-adorned versions, it does not include the various punctuation or digits (since they are not solely `Latin`).

Changes in the character class semantics may have happened if a script and a block happen to have the same name, for example `Hebrew`. In such cases the script wins and `\p{InHebrew}` now means the script definition of Hebrew. The block definition in still available, though, by appending `Block` to the name: `\p{InHebrewBlock}` means what `\p{InHebrew}` meant in perl 5.6.0. For the full list of affected character classes, see Blocks in *perlunicode*.

88.3.5 Deprecations

The current user-visible implementation of pseudo-hashes (the weird use of the first array element) is deprecated starting from Perl 5.8.0 and will be removed in Perl 5.10.0, and the feature will be implemented differently. Not only is the current interface rather ugly, but the current implementation slows down normal array and hash use quite noticeably. The `fields` pragma interface will remain available.

The syntaxes `@a->[...]` and `@h->{...}` have now been deprecated.

The suidperl is also considered to be too much a risk to continue maintaining and the suidperl code is likely to be removed in a future release.

The `package;` syntax (`package` without an argument has been deprecated. Its semantics were never that clear and its implementation even less so. If you have used that feature to disallow all but fully qualified variables, `use strict;` instead.

The chdir(undef) and chdir(”) behaviors to match chdir() has been deprecated. In future versions, chdir(undef) and chdir(”) will simply fail.

88.4 Core Enhancements

In general a lot of fixing has happened in the area of Perl's understanding of numbers, both integer and floating point. Since in many systems the standard number parsing functions like `strtoul()` and `atof()` seem to have bugs, Perl tries to work around their deficiencies. This results hopefully in more accurate numbers.

- The rules for allowing underscores (underbars) in numeric constants have been relaxed and simplified: now you can have an underscore **between digits**.
- GMAGIC (right-hand side magic) could in many cases such as string concatenation be invoked too many times.
- Lexicals I: lexicals outside an eval "" weren't resolved correctly inside a subroutine definition inside the eval "" if they were not already referenced in the top level of the eval""ed code.
- Lexicals II: lexicals leaked at file scope into subroutines that were declared before the lexicals.
- Lvalue subroutines can now return `undef` in list context.
- The `op_clear` and `op_null` are now exported.
- A new special regular expression variable has been introduced: `$^N`, which contains the most-recently closed group (submatch).
- *utime* now supports `utime undef, undef, @files` to change the file timestamps to the current time.

- The Perl parser has been stress tested using both random input and Markov chain input.
- `eval "v200"` now works.
- VMS now works under PerlIO.
- END blocks are now run even if you exit/die in a BEGIN block. The execution of END blocks is now controlled by PL_exit_flags & PERL_EXIT_DESTRUCT_END. This enables the new behaviour for perl embedders. This will default in 5.10. See *perlembed*.

88.5 Modules and Pragmata

88.5.1 New Modules and Distributions

- *Attribute::Handlers* - Simpler definition of attribute handlers
- *ExtUtils::Constant* - generate XS code to import C header constants
- *I18N::Langinfo* - query locale information
- *I18N::LangTags* - functions for dealing with RFC3066-style language tags
- *libnet* - a collection of perl5 modules related to network programming

  Perl installation leaves libnet unconfigured, use *libnetcfg* to configure.
- *List::Util* - selection of general-utility list subroutines
- *Locale::Maketext* - framework for localization
- *Memoize* - Make your functions faster by trading space for time
- *NEXT* - pseudo-class for method redispatch
- *Scalar::Util* - selection of general-utility scalar subroutines
- *Test::More* - yet another framework for writing test scripts
- *Test::Simple* - Basic utilities for writing tests
- *Time::HiRes* - high resolution ualarm, usleep, and gettimeofday
- *Time::Piece* - Object Oriented time objects

  (Previously known as *Time::Object*.)
- *Time::Seconds* - a simple API to convert seconds to other date values
- *UnicodeCD* - Unicode Character Database

88.5.2 Updated And Improved Modules and Pragmata

- *B::Deparse* module has been significantly enhanced. It now can deparse almost all of the standard test suite (so that the tests still succeed). There is a make target "test.deparse" for trying this out.
- *Class::Struct* now assigns the array/hash element if the accessor is called with an array/hash element as the **sole** argument.
- *Cwd* extension is now (even) faster.
- DB_File extension has been updated to version 1.77.
- *Fcntl*, *Socket*, and *Sys::Syslog* have been rewritten to use the new-style constant dispatch section (see *ExtUtils::Constant*).

- *File::Find* is now (again) reentrant. It also has been made more portable.
- *File::Glob* now supports `GLOB_LIMIT` constant to limit the size of the returned list of filenames.
- *IO::Socket::INET* now supports `LocalPort` of zero (usually meaning that the operating system will make one up.)
- The *vars* pragma now supports declaring fully qualified variables. (Something that `our()` does not and will not support.)

88.6 Utility Changes

- The *emacs/e2ctags.pl* is now much faster.
- *h2ph* now supports C trigraphs.
- *h2xs* uses the new *ExtUtils::Constant* module which will affect newly created extensions that define constants. Since the new code is more correct (if you have two constants where the first one is a prefix of the second one, the first constant **never** gets defined), less lossy (it uses integers for integer constant, as opposed to the old code that used floating point numbers even for integer constants), and slightly faster, you might want to consider regenerating your extension code (the new scheme makes regenerating easy). *h2xs* now also supports C trigraphs.
- *libnetcfg* has been added to configure the libnet.
- The *Pod::Html* (and thusly *pod2html*) now allows specifying a cache directory.

88.7 New Documentation

- *Locale::Maketext::TPJ13* is an article about software localization, originally published in The Perl Journal #13, republished here with kind permission.
- More README.$PLATFORM files have been converted into pod, which also means that they also be installed as perl$PLATFORM documentation files. The new files are *perlapollo*, *perlbeos*, *perldgux*, *perlhurd*, *perlmint*, *perlnetware*, *perlplan9*, *perlqnx*, and *perltru64*.
- The *Todo* and *Todo-5.6* files have been merged into *perltodo*.
- Use of the *gprof* tool to profile Perl has been documented in *perlhack*. There is a make target "perl.gprof" for generating a gprofiled Perl executable.

88.8 Installation and Configuration Improvements

88.8.1 New Or Improved Platforms

- AIX should now work better with gcc, threads, and 64-bitness. Also the long doubles support in AIX should be better now. See *perlaix*.
- AtheOS (http://www.atheos.cx/) is a new platform.
- DG/UX platform now supports the 5.005-style threads. See *perldgux*.
- DYNIX/ptx platform (a.k.a. dynixptx) is supported at or near osvers 4.5.2.
- Several Mac OS (Classic) portability patches have been applied. We hope to get a fully working port by 5.8.0. (The remaining problems relate to the changed IO model of Perl.) See *perlmacos*.
- Mac OS X (or Darwin) should now be able to build Perl even on HFS+ filesystems. (The case-insensitivity confused the Perl build process.)
- NetWare from Novell is now supported. See *perlnetware*.
- The Amdahl UTS UNIX mainframe platform is now supported.

88.8.2 Generic Improvements

- In AFS installations one can configure the root of the AFS to be somewhere else than the default */afs* by using the Configure parameter `-Dafsroot=/some/where/else`.
- The version of Berkeley DB used when the Perl (and, presumably, the DB_File extension) was built is now available as `@Config{qw(db_version_major db_version_minor db_version_patch)}` from Perl and as `DB_VERSION_MAJOR_CFG DB_VERSION_MINOR_CFG DB_VERSION_PATCH_CFG` from C.
- The Thread extension is now not built at all under ithreads (`Configure -Duseithreads`) because it wouldn't work anyway (the Thread extension requires being Configured with `-Duse5005threads`).
- The `B::Deparse` compiler backend has been so significantly improved that almost the whole Perl test suite passes after being deparsed. A make target has been added to help in further testing: `make test.deparse`.

88.9 Selected Bug Fixes

- The autouse pragma didn't work for Multi::Part::Function::Names.
- The behaviour of non-decimal but numeric string constants such as "0x23" was platform-dependent: in some platforms that was seen as 35, in some as 0, in some as a floating point number (don't ask). This was caused by Perl using the operating system libraries in a situation where the result of the string to number conversion is undefined: now Perl consistently handles such strings as zero in numeric contexts.
- *dprofpp* -R didn't work.
- PERL5OPT with embedded spaces didn't work.
- *Sys::Syslog* ignored the `LOG_AUTH` constant.

88.9.1 Platform Specific Changes and Fixes

- Some versions of glibc have a broken modfl(). This affects builds with `-Duselongdouble`. This version of Perl detects this brokenness and has a workaround for it. The glibc release 2.2.2 is known to have fixed the modfl() bug.

88.10 New or Changed Diagnostics

- In the regular expression diagnostics the `<< HERE` marker introduced in 5.7.0 has been changed to be `<- HERE` since too many people found the << to be too similar to here-document starters.
- If you try to pack in *perlfunc* a number less than 0 or larger than 255 using the `"C"` format you will get an optional warning. Similarly for the `"c"` format and a number less than -128 or more than 127.
- Certain regex modifiers such as `(?o)` make sense only if applied to the entire regex. You will an optional warning if you try to do otherwise.
- Using arrays or hashes as references (e.g. `%foo->{bar}` has been deprecated for a while. Now you will get an optional warning.

88.11 Source Code Enhancements

88.11.1 MAGIC constants

The MAGIC constants (e.g. `'P'`) have been macrofied (e.g. `PERL_MAGIC_TIED`) for better source code readability and maintainability.

88.11.2 Better commented code

perly.c, *sv.c*, and *sv.h* have now been extensively commented.

88.11.3 Regex pre-/post-compilation items matched up

The regex compiler now maintains a structure that identifies nodes in the compiled bytecode with the corresponding syntactic features of the original regex expression. The information is attached to the new `offsets` member of the `struct regexp`. See *perldebguts* for more complete information.

88.11.4 gcc -Wall

The C code has been made much more `gcc -Wall` clean. Some warning messages still remain, though, so if you are compiling with gcc you will see some warnings about dubious practices. The warnings are being worked on.

88.12 New Tests

Several new tests have been added, especially for the *lib* subsection.

The tests are now reported in a different order than in earlier Perls. (This happens because the test scripts from under t/lib have been moved to be closer to the library/extension they are testing.)

88.13 Known Problems

Note that unlike other sections in this document (which describe changes since 5.7.0) this section is cumulative containing known problems for all the 5.7 releases.

88.13.1 AIX

- In AIX 4.2 Perl extensions that use C++ functions that use statics may have problems in that the statics are not getting initialized. In newer AIX releases this has been solved by linking Perl with the libC_r library, but unfortunately in AIX 4.2 the said library has an obscure bug where the various functions related to time (such as time() and gettimeofday()) return broken values, and therefore in AIX 4.2 Perl is not linked against the libC_r.
- vac 5.0.0.0 May Produce Buggy Code For Perl

  The AIX C compiler vac version 5.0.0.0 may produce buggy code, resulting in few random tests failing, but when the failing tests are run by hand, they succeed. We suggest upgrading to at least vac version 5.0.1.0, that has been known to compile Perl correctly. "lslpp -L|grep vac.C" will tell you the vac version.

88.13.2 Amiga Perl Invoking Mystery

One cannot call Perl using the `volume:` syntax, that is, `perl -v` works, but for example `bin:perl -v` doesn't. The exact reason is known but the current suspect is the *ixemul* library.

88.13.3 lib/ftmp-security tests warn 'system possibly insecure'

Don't panic. Read INSTALL 'make test' section instead.

88.13.4 Cygwin intermittent failures of lib/Memoize/t/expire_file 11 and 12

The subtests 11 and 12 sometimes fail and sometimes work.

88.13.5 HP-UX lib/io_multihomed Fails When LP64-Configured

The lib/io_multihomed test may hang in HP-UX if Perl has been configured to be 64-bit. Because other 64-bit platforms do not hang in this test, HP-UX is suspect. All other tests pass in 64-bit HP-UX. The test attempts to create and connect to "multihomed" sockets (sockets which have multiple IP addresses).

88.13.6 HP-UX lib/posix Subtest 9 Fails When LP64-Configured

If perl is configured with -Duse64bitall, the successful result of the subtest 10 of lib/posix may arrive before the successful result of the subtest 9, which confuses the test harness so much that it thinks the subtest 9 failed.

88.13.7 Linux With Sfio Fails op/misc Test 48

No known fix.

88.13.8 OS/390

OS/390 has rather many test failures but the situation is actually better than it was in 5.6.0, it's just that so many new modules and tests have been added.

```
Failed Test                     Stat Wstat Total Fail  Failed  List of Failed
---------------------------------------------------------------------------
../ext/B/Deparse.t                            14    1   7.14%  14
../ext/B/Showlex.t                             1    1 100.00%  1
../ext/Encode/Encode/Tcl.t                   610   13   2.13%  592 594 596 598
                                                               600 602 604-610
../ext/IO/lib/IO/t/io_unix.t     113 28928     5    3  60.00%  3-5
../ext/POSIX/POSIX.t                          29    1   3.45%  14
../ext/Storable/t/lock.t         255 65280     5    3  60.00%  3-5
../lib/locale.t                  129 33024   117   19  16.24%  99-117
../lib/warnings.t                            434    1   0.23%  75
../lib/ExtUtils.t                             27    1   3.70%  25
../lib/Math/BigInt/t/bigintpm.t             1190    1   0.08%  1145
../lib/Unicode/UCD.t                          81   48  59.26%  1-16 49-64 66-81
../lib/User/pwent.t                            9    1  11.11%  4
op/pat.t                                     660    6   0.91%  242-243 424-425
                                                               626-627
op/split.t                         0     9    ??   ??       %  ??
op/taint.t                                   174    3   1.72%  156 162 168
op/tr.t                                       70    3   4.29%  50 58-59
Failed 16/422 test scripts, 96.21% okay. 105/23251 subtests failed, 99.55% okay.
```

88.13.9 op/sprintf tests 129 and 130

The op/sprintf tests 129 and 130 are known to fail on some platforms. Examples include any platform using sfio, and Compaq/Tandem's NonStop-UX. The failing platforms do not comply with the ANSI C Standard, line 19ff on page 134 of ANSI X3.159 1989 to be exact. (They produce something other than "1" and "-1" when formatting 0.6 and -0.6 using the printf format "%.0f", most often they produce "0" and "-0".)

88.13.10 Failure of Thread tests

Note that support for 5.005-style threading remains experimental.

The following tests are known to fail due to fundamental problems in the 5.005 threading implementation. These are not new failures–Perl 5.005_0x has the same bugs, but didn't have these tests.

```
lib/autouse.t              4
t/lib/thr5005.t            19-20
```

88.13.11 UNICOS

- ext/POSIX/sigaction subtests 6 and 13 may fail.
- lib/ExtUtils may spuriously claim that subtest 28 failed, which is interesting since the test only has 27 tests.
- Numerous numerical test failures

```
    op/numconvert                209,210,217,218
    op/override                  7
    ext/Time/HiRes/HiRes         9
    lib/Math/BigInt/t/bigintpm   1145
    lib/Math/Trig                25
```

  These tests fail because of yet unresolved floating point inaccuracies.

88.13.12 UTS

There are a few known test failures, see *perluts*.

88.13.13 VMS

Rather many tests are failing in VMS but that actually more tests succeed in VMS than they used to, it's just that there are many, many more tests than there used to be.

Here are the known failures from some compiler/platform combinations.

DEC C V5.3-006 on OpenVMS VAX V6.2

```
  [-.ext.list.util.t]tainted..............FAILED on test 3
  [-.ext.posix]sigaction..................FAILED on test 7
  [-.ext.time.hires]hires.................FAILED on test 14
  [-.lib.file.find]taint..................FAILED on test 17
  [-.lib.math.bigint.t]bigintpm...........FAILED on test 1183
  [-.lib.test.simple.t]exit...............FAILED on test 1
  [.lib]vmsish............................FAILED on test 13
  [.op]sprintf............................FAILED on test 12
  Failed 8/399 tests, 91.23% okay.
```

DEC C V6.0-001 on OpenVMS Alpha V7.2-1 and Compaq C V6.2-008 on OpenVMS Alpha V7.1

```
  [-.ext.list.util.t]tainted..............FAILED on test 3
  [-.lib.file.find]taint..................FAILED on test 17
  [-.lib.test.simple.t]exit...............FAILED on test 1
  [.lib]vmsish............................FAILED on test 13
  Failed 4/399 tests, 92.48% okay.
```

Compaq C V6.4-005 on OpenVMS Alpha 7.2.1

```
  [-.ext.b]showlex........................FAILED on test 1
  [-.ext.list.util.t]tainted..............FAILED on test 3
  [-.lib.file.find]taint..................FAILED on test 17
  [-.lib.test.simple.t]exit...............FAILED on test 1
  [.lib]vmsish............................FAILED on test 13
  [.op]misc...............................FAILED on test 49
  Failed 6/401 tests, 92.77% okay.
```

88.13.14 Win32

In multi-CPU boxes there are some problems with the I/O buffering: some output may appear twice.

88.13.15 Localising a Tied Variable Leaks Memory

```
use Tie::Hash;
tie my %tie_hash => 'Tie::StdHash';

...

local($tie_hash{Foo}) = 1; # leaks
```

Code like the above is known to leak memory every time the local() is executed.

88.13.16 Self-tying of Arrays and Hashes Is Forbidden

Self-tying of arrays and hashes is broken in rather deep and hard-to-fix ways. As a stop-gap measure to avoid people from getting frustrated at the mysterious results (core dumps, most often) it is for now forbidden (you will get a fatal error even from an attempt).

88.13.17 Variable Attributes are not Currently Usable for Tieing

This limitation will hopefully be fixed in future. (Subroutine attributes work fine for tieing, see *Attribute::Handlers*).

88.13.18 Building Extensions Can Fail Because Of Largefiles

Some extensions like mod_perl are known to have issues with 'largefiles', a change brought by Perl 5.6.0 in which file offsets default to 64 bits wide, where supported. Modules may fail to compile at all or compile and work incorrectly. Currently there is no good solution for the problem, but Configure now provides appropriate non-largefile ccflags, ldflags, libswanted, and libs in the %Config hash (e.g., $Config{ccflags_nolargefiles}) so the extensions that are having problems can try configuring themselves without the largefileness. This is admittedly not a clean solution, and the solution may not even work at all. One potential failure is whether one can (or, if one can, whether it's a good idea) link together at all binaries with different ideas about file offsets, all this is platform-dependent.

88.13.19 The Compiler Suite Is Still Experimental

The compiler suite is slowly getting better but is nowhere near working order yet.

88.13.20 The Long Double Support is Still Experimental

The ability to configure Perl's numbers to use "long doubles", floating point numbers of hopefully better accuracy, is still experimental. The implementations of long doubles are not yet widespread and the existing implementations are not quite mature or standardised, therefore trying to support them is a rare and moving target. The gain of more precision may also be offset by slowdown in computations (more bits to move around, and the operations are more likely to be executed by less optimised libraries).

88.14 Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup and the perl bug database at http://bugs.perl.org/ There may also be information at http://www.perl.com/perl/ , the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to perlbug@perl.org to be analysed by the Perl porting team.

88.15 SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

88.16 HISTORY

Written by Jarkko Hietaniemi <*jhi@iki.fi*>, with many contributions from The Perl Porters and Perl Users submitting feedback and patches.

Send omissions or corrections to <*perlbug@perl.org*>.

Chapter 89

perl571delta

What's new for perl v5.7.1

89.1 DESCRIPTION

This document describes differences between the 5.7.0 release and the 5.7.1 release.

(To view the differences between the 5.6.0 release and the 5.7.0 release, see *perl570delta*.)

89.2 Security Vulnerability Closed

(This change was already made in 5.7.0 but bears repeating here.)

A potential security vulnerability in the optional suidperl component of Perl was identified in August 2000. suidperl is neither built nor installed by default. As of April 2001 the only known vulnerable platform is Linux, most likely all Linux distributions. CERT and various vendors and distributors have been alerted about the vulnerability. See http://www.cpan.org/src/5.0/sperl-2000-08-05/sperl-2000-08-05.txt for more information.

The problem was caused by Perl trying to report a suspected security exploit attempt using an external program, /bin/mail. On Linux platforms the /bin/mail program had an undocumented feature which when combined with suidperl gave access to a root shell, resulting in a serious compromise instead of reporting the exploit attempt. If you don't have /bin/mail, or if you have 'safe setuid scripts', or if suidperl is not installed, you are safe.

The exploit attempt reporting feature has been completely removed from all the Perl 5.7 releases (and will be gone also from the maintenance release 5.6.1), so that particular vulnerability isn't there anymore. However, further security vulnerabilities are, unfortunately, always possible. The suidperl code is being reviewed and if deemed too risky to continue to be supported, it may be completely removed from future releases. In any case, suidperl should only be used by security experts who know exactly what they are doing and why they are using suidperl instead of some other solution such as sudo (see http://www.courtesan.com/sudo/).

89.3 Incompatible Changes

- Although "you shouldn't do that", it was possible to write code that depends on Perl's hashed key order (Data::Dumper does this). The new algorithm "One-at-a-Time" produces a different hashed key order. More details are in §89.6.
- The list of filenames from glob() (or <...>) is now by default sorted alphabetically to be csh-compliant. (bsd_glob() does still sort platform natively, ASCII or EBCDIC, unless GLOB_ALPHASORT is specified.)

89.4 Core Enhancements

89.4.1 AUTOLOAD Is Now Lvaluable

AUTOLOAD is now lvaluable, meaning that you can add the :lvalue attribute to AUTOLOAD subroutines and you can assign to the AUTOLOAD return value.

89.4.2 PerlIO is Now The Default

- IO is now by default done via PerlIO rather than system's "stdio". PerlIO allows "layers" to be "pushed" onto a file handle to alter the handle's behaviour. Layers can be specified at open time via 3-arg form of open:

  ```
  open($fh,'>:crlf :utf8', $path) || ...
  ```

  or on already opened handles via extended `binmode`:

  ```
  binmode($fh,':encoding(iso-8859-7)');
  ```

  The built-in layers are: unix (low level read/write), stdio (as in previous Perls), perlio (re-implementation of stdio buffering in a portable manner), crlf (does CRLF <=> "\n" translation as on Win32, but available on any platform). A mmap layer may be available if platform supports it (mostly UNIXes).

  Layers to be applied by default may be specified via the 'open' pragma.

  See §89.9 for the effects of PerlIO on your architecture name.

- File handles can be marked as accepting Perl's internal encoding of Unicode (UTF-8 or UTF-EBCDIC depending on platform) by a pseudo layer ":utf8" :

  ```
  open($fh,">:utf8","Uni.txt");
  ```

  Note for EBCDIC users: the pseudo layer ":utf8" is erroneously named for you since it's not UTF-8 what you will be getting but instead UTF-EBCDIC. See *perlunicode*, *utf8*, and http://www.unicode.org/unicode/reports/tr16/ for more information. In future releases this naming may change.

- File handles can translate character encodings from/to Perl's internal Unicode form on read/write via the ":encoding()" layer.

- File handles can be opened to "in memory" files held in Perl scalars via:

  ```
  open($fh,'>', \$variable) || ...
  ```

- Anonymous temporary files are available without need to 'use FileHandle' or other module via

  ```
  open($fh,"+>", undef) || ...
  ```

  That is a literal undef, not an undefined value.

- The list form of `open` is now implemented for pipes (at least on UNIX):

  ```
  open($fh,"-|", 'cat', '/etc/motd')
  ```

  creates a pipe, and runs the equivalent of exec('cat', '/etc/motd') in the child process.

- The following builtin functions are now overridable: chop(), chomp(), each(), keys(), pop(), push(), shift(), splice(), unshift().

- Formats now support zero-padded decimal fields.

- Perl now tries internally to use integer values in numeric conversions and basic arithmetics (+ - * /) if the arguments are integers, and tries also to keep the results stored internally as integers. This change leads into often slightly faster and always less lossy arithmetics. (Previously Perl always preferred floating point numbers in its math.)
- The printf() and sprintf() now support parameter reordering using the `%\d+\$` and `*\d+\$` syntaxes. For example

  ```
  print "%2\$s %1\$s\n", "foo", "bar";
  ```

  will print "bar foo\n"; This feature helps in writing internationalised software.
- Unicode in general should be now much more usable. Unicode can be used in hash keys, Unicode in regular expressions should work now, Unicode in tr/// should work now (though tr/// seems to be a particularly tricky to get right, so you have been warned)
- The Unicode Character Database coming with Perl has been upgraded to Unicode 3.1. For more information, see http://www.unicode.org/ , and http://www.unicode.org/unicode/reports/tr27/

  For developers interested in enhancing Perl's Unicode capabilities: almost all the UCD files are included with the Perl distribution in the lib/unicode subdirectory. The most notable omission, for space considerations, is the Unihan database.
- The Unicode character classes `\p{Blank}` and `\p{SpacePerl}` have been added. "Blank" is like C isblank(), that is, it contains only "horizontal whitespace" (the space character is, the newline isn't), and the "SpacePerl" is the Unicode equivalent of `\s` (`\p{Space}` isn't, since that includes the vertical tabulator character, whereas `\s` doesn't.)

89.4.3 Signals Are Now Safe

Perl used to be fragile in that signals arriving at inopportune moments could corrupt Perl's internal state.

89.5 Modules and Pragmata

89.5.1 New Modules

- B::Concise, by Stephen McCamant, is a new compiler backend for walking the Perl syntax tree, printing concise info about ops. The output is highly customisable.

  See *B::Concise* for more information.
- Class::ISA, by Sean Burke, for reporting the search path for a class's ISA tree, has been added.

  See *Class::ISA* for more information.
- Cwd has now a split personality: if possible, an extension is used, (this will hopefully be both faster and more secure and robust) but if not possible, the familiar Perl library implementation is used.
- Digest, a frontend module for calculating digests (checksums), from Gisle Aas, has been added.

  See *Digest* for more information.
- Digest::MD5 for calculating MD5 digests (checksums), by Gisle Aas, has been added.

  ```
  use Digest::MD5 'md5_hex';

  $digest = md5_hex("Thirsty Camel");

  print $digest, "\n"; # 01d19d9d2045e005c3f1b80e8b164de1
  ```

  NOTE: the MD5 backward compatibility module is deliberately not included since its use is discouraged.

  See *Digest::MD5* for more information.

- Encode, by Nick Ing-Simmons, provides a mechanism to translate between different character encodings. Support for Unicode, ISO-8859-*, ASCII, CP*, KOI8-R, and three variants of EBCDIC are compiled in to the module. Several other encodings (like Japanese, Chinese, and MacIntosh encodings) are included and will be loaded at runtime.

  Any encoding supported by Encode module is also available to the ":encoding()" layer if PerlIO is used.

  See *Encode* for more information.

- Filter::Simple is an easy-to-use frontend to Filter::Util::Call, from Damian Conway.

```
    # in MyFilter.pm:

    package MyFilter;

    use Filter::Simple sub {
        while (my ($from, $to) = splice @_, 0, 2) {
                s/$from/$to/g;
        }
    };

    1;

    # in user's code:

    use MyFilter qr/red/ => 'green';

    print "red\n";   # this code is filtered, will print "green\n"
    print "bored\n"; # this code is filtered, will print "bogreen\n"

    no MyFilter;

    print "red\n";   # this code is not filtered, will print "red\n"
```

  See *Filter::Simple* for more information.

- Filter::Util::Call, by Paul Marquess, provides you with the framework to write *Source Filters* in Perl. For most uses the frontend Filter::Simple is to be preferred. See *Filter::Util::Call* for more information.

- Locale::Constants, Locale::Country, Locale::Currency, and Locale::Language, from Neil Bowers, have been added. They provide the codes for various locale standards, such as "fr" for France, "usd" for US Dollar, and "jp" for Japanese.

```
    use Locale::Country;

    $country = code2country('jp');               # $country gets 'Japan'
    $code    = country2code('Norway');           # $code gets 'no'
```

  See *Locale::Constants*, *Locale::Country*, *Locale::Currency*, and *Locale::Language* for more information.

- MIME::Base64, by Gisle Aas, allows you to encode data in base64.

```
    use MIME::Base64;

    $encoded = encode_base64('Aladdin:open sesame');
    $decoded = decode_base64($encoded);

    print $encoded, "\n"; # "QWxhZGRpbjpvcGVuIHNlc2FtZQ=="
```

See *MIME::Base64* for more information.

- MIME::QuotedPrint, by Gisle Aas, allows you to encode data in quoted-printable encoding.

  ```
  use MIME::QuotedPrint;

  $encoded = encode_qp("Smiley in Unicode: \x{263a}");
  $decoded = decode_qp($encoded);

  print $encoded, "\n"; # "Smiley in Unicode: =263A"
  ```

  MIME::QuotedPrint has been enhanced to provide the basic methods necessary to use it with PerlIO::Via as in :

  ```
  use MIME::QuotedPrint;
  open($fh,">Via(MIME::QuotedPrint)",$path)
  ```

  See *MIME::QuotedPrint* for more information.

- PerlIO::Scalar, by Nick Ing-Simmons, provides the implementation of IO to "in memory" Perl scalars as discussed above. It also serves as an example of a loadable layer. Other future possibilities include PerlIO::Array and PerlIO::Code. See *PerlIO::Scalar* for more information.

- PerlIO::Via, by Nick Ing-Simmons, acts as a PerlIO layer and wraps PerlIO layer functionality provided by a class (typically implemented in perl code).

  ```
  use MIME::QuotedPrint;
  open($fh,">Via(MIME::QuotedPrint)",$path)
  ```

  This will automatically convert everything output to `$fh` to Quoted-Printable. See *PerlIO::Via* for more information.

- Pod::Text::Overstrike, by Joe Smith, has been added. It converts POD data to formatted overstrike text. See *Pod::Text::Overstrike* for more information.

- Switch from Damian Conway has been added. Just by saying

  ```
  use Switch;
  ```

  you have `switch` and `case` available in Perl.

  ```
  use Switch;

  switch ($val) {

              case 1          { print "number 1" }
              case "a"        { print "string a" }
              case [1..10,42] { print "number in list" }
              case (@array)   { print "number in list" }
              case /\w+/      { print "pattern" }
              case qr/\w+/    { print "pattern" }
              case (%hash)    { print "entry in hash" }
              case (\%hash)   { print "entry in hash" }
              case (\&sub)    { print "arg to subroutine" }
              else            { print "previous case not true" }
  }
  ```

  See *Switch* for more information.

- Text::Balanced from Damian Conway has been added, for extracting delimited text sequences from strings.

  ```
  use Text::Balanced 'extract_delimited';

  ($a, $b) = extract_delimited("'never say never', he never said", "'", '');
  ```

  $a will be "'never say never'", $b will be ', he never said'.

  In addition to extract_delimited() there are also extract_bracketed(), extract_quotelike(), extract_codeblock(), extract_variable(), extract_tagged(), extract_multiple(), gen_delimited_pat(), and gen_extract_tagged(). With these you can implement rather advanced parsing algorithms. See *Text::Balanced* for more information.

- Tie::RefHash::Nestable, by Edward Avis, allows storing hash references (unlike the standard Tie::RefHash) The module is contained within Tie::RefHash.

- XS::Typemap, by Tim Jenness, is a test extension that exercises XS typemaps. Nothing gets installed but for extension writers the code is worth studying.

89.5.2 Updated And Improved Modules and Pragmata

- B::Deparse should be now more robust. It still far from providing a full round trip for any random piece of Perl code, though, and is under active development: expect more robustness in 5.7.2.

- Class::Struct can now define the classes in compile time.

- Math::BigFloat has undergone much fixing, and in addition the fmod() function now supports modulus operations.

  (The fixed Math::BigFloat module is also available in CPAN for those who can't upgrade their Perl: http://www.cpan.org/authors/id/J/JP/JPEACOCK/)

- Devel::Peek now has an interface for the Perl memory statistics (this works only if you are using perl's malloc, and if you have compiled with debugging).

- IO::Socket has now atmark() method, which returns true if the socket is positioned at the out-of-band mark. The method is also exportable as a sockatmark() function.

- IO::Socket::INET has support for ReusePort option (if your platform supports it). The Reuse option now has an alias, ReuseAddr. For clarity you may want to prefer ReuseAddr.

- Net::Ping has been enhanced. There is now "external" protocol which uses Net::Ping::External module which runs external ping(1) and parses the output. An alpha version of Net::Ping::External is available in CPAN and in 5.7.2 the Net::Ping::External may be integrated to Perl.

- The `open` pragma allows layers other than ":raw" and ":crlf" when using PerlIO.

- POSIX::sigaction() is now much more flexible and robust. You can now install coderef handlers, 'DEFAULT', and 'IGNORE' handlers, installing new handlers was not atomic.

- The Test module has been significantly enhanced. Its use is greatly recommended for module writers.

- The utf8:: name space (as in the pragma) provides various Perl-callable functions to provide low level access to Perl's internal Unicode representation. At the moment only length() has been implemented.

The following modules have been upgraded from the versions at CPAN: CPAN, CGI, DB_File, File::Temp, Getopt::Long, Pod::Man, Pod::Text, Storable, Text-Tabs+Wrap.

89.6 Performance Enhancements

- Hashes now use Bob Jenkins "One-at-a-Time" hashing key algorithm (http://burtleburtle.net/bob/hash/doobs.html). This algorithm is reasonably fast while producing a much better spread of values than the old hashing algorithm (originally by Chris Torek, later tweaked by Ilya Zakharevich). Hash values output from the algorithm on a hash of all 3-char printable ASCII keys comes much closer to passing the DIEHARD random number generation tests. According to perlbench, this change has not affected the overall speed of Perl.
- unshift() should now be noticeably faster.

89.7 Utility Changes

- h2xs now produces template README.
- s2p has been completely rewritten in Perl. (It is in fact a full implementation of sed in Perl.)
- xsubpp now supports OUT keyword.

89.8 New Documentation

89.8.1 perlclib

Internal replacements for standard C library functions. (Interesting only for extension writers and Perl core hackers.)

89.8.2 perliol

Internals of PerlIO with layers.

89.8.3 README.aix

Documentation on compiling Perl on AIX has been added. AIX has several different C compilers and getting the right patch level is essential. On install README.aix will be installed as *perlaix*.

89.8.4 README.bs2000

Documentation on compiling Perl on the POSIX-BC platform (an EBCDIC mainframe environment) has been added.

This was formerly known as README.posix-bc but the name was considered to be too confusing (it has nothing to do with the POSIX module or the POSIX standard). On install README.bs2000 will be installed as *perlbs2000*.

89.8.5 README.macos

In perl 5.7.1 (and in the 5.6.1) the MacPerl sources have been synchronised with the standard Perl sources. To compile MacPerl some additional steps are required, and this file documents those steps. On install README.macos will be installed as *perlmacos*.

89.8.6 README.mpeix

The README.mpeix has been podified, which means that this information about compiling and using Perl on the MPE/iX miniframe platform will be installed as *perlmpeix*.

89.8.7 README.solaris

README.solaris has been created and Solaris wisdom from elsewhere in the Perl documentation has been collected there. On install README.solaris will be installed as *perlsolaris*.

89.8.8 README.vos

The README.vos has been podified, which means that this information about compiling and using Perl on the Stratus VOS miniframe platform will be installed as *perlvos*.

89.8.9 Porting/repository.pod

Documentation on how to use the Perl source repository has been added.

89.9 Installation and Configuration Improvements

- Because PerlIO is now the default on most platforms, "-perlio" doesn't get appended to the $Config{archname} (also known as $ˆO) anymore. Instead, if you explicitly choose not to use perlio (Configure command line option -Uuseperlio), you will get "-stdio" appended.
- Another change related to the architecture name is that "-64all" (-Duse64bitall, or "maximally 64-bit") is appended only if your pointers are 64 bits wide. (To be exact, the use64bitall is ignored.)
- APPLLIB_EXP, a less-know configuration-time definition, has been documented. It can be used to prepend site-specific directories to Perl's default search path (@INC), see INSTALL for information.
- Building Berkeley DB3 for compatibility modes for DB, NDBM, and ODBM has been documented in INSTALL.
- If you are on IRIX or Tru64 platforms, new profiling/debugging options have been added, see *perlhack* for more information about pixie and Third Degree.

89.9.1 New Or Improved Platforms

For the list of platforms known to support Perl, see Supported Platforms in *perlport*.

- AIX dynamic loading should be now better supported.
- After a long pause, AmigaOS has been verified to be happy with Perl.
- EBCDIC platforms (z/OS, also known as OS/390, POSIX-BC, and VM/ESA) have been regained. Many test suite tests still fail and the co-existence of Unicode and EBCDIC isn't quite settled, but the situation is much better than with Perl 5.6. See *perlos390*, *perlbs2000* (for POSIX-BC), and *perlvmesa* for more information.
- Building perl with -Duseithreads or -Duse5005threads now works under HP-UX 10.20 (previously it only worked under 10.30 or later). You will need a thread library package installed. See README.hpux.
- Mac OS Classic (MacPerl has of course been available since perl 5.004 but now the source code bases of standard Perl and MacPerl have been synchronised)
- NCR MP-RAS is now supported.
- NonStop-UX is now supported.
- Amdahl UTS is now supported.
- z/OS (formerly known as OS/390, formerly known as MVS OE) has now support for dynamic loading. This is not selected by default, however, you must specify -Dusedl in the arguments of Configure.

89.9.2 Generic Improvements

- Configure no longer includes the DBM libraries (dbm, gdbm, db, ndbm) when building the Perl binary. The only exception to this is SunOS 4.x, which needs them.
- Some new Configure symbols, useful for extension writers:

  d_cmsghdr
  For struct cmsghdr.

  d_fcntl_can_lock
  Whether fcntl() can be used for file locking.

  d_fsync

  d_getitimer

  d_getpagsz
  For getpagesize(), though you should prefer POSIX::sysconf(_SC_PAGE_SIZE))

  d_msghdr_s
  For struct msghdr.

  need_va_copy
  Whether one needs to use Perl_va_copy() to copy varargs.

  d_readv

  d_recvmsg

  d_sendmsg

  sig_size
  The number of elements in an array needed to hold all the available signals.

  d_sockatmark

  d_strtoq

  d_u32align
  Whether one needs to access character data aligned by U32 sized pointers.

  d_ualarm

  d_usleep

- Removed Configure symbols: the PDP-11 memory model settings: huge, large, medium, models.
- SOCKS support is now much more robust.
- If your file system supports symbolic links you can build Perl outside of the source directory by

```
mkdir perl/build/directory
cd perl/build/directory
sh /path/to/perl/source/Configure -Dmksymlinks ...
```

  This will create in perl/build/directory a tree of symbolic links pointing to files in /path/to/perl/source. The original files are left unaffected. After Configure has finished you can just say

```
make all test
```

  and Perl will be built and tested, all in perl/build/directory.

89.10 Selected Bug Fixes

Numerous memory leaks and uninitialized memory accesses have been hunted down. Most importantly anonymous subs used to leak quite a bit.

- chop(@list) in list context returned the characters chopped in reverse order. This has been reversed to be in the right order.
- The order of DESTROYs has been made more predictable.
- mkdir() now ignores trailing slashes in the directory name, as mandated by POSIX.
- Attributes (like :shared) didn't work with our().
- The PERL5OPT environment variable (for passing command line arguments to Perl) didn't work for more than a single group of options.
- The tainting behaviour of sprintf() has been rationalized. It does not taint the result of floating point formats anymore, making the behaviour consistent with that of string interpolation.
- All but the first argument of the IO syswrite() method are now optional.
- Tie::ARRAY SPLICE method was broken.
- vec() now tries to work with characters <= 255 when possible, but it leaves higher character values in place. In that case, if vec() was used to modify the string, it is no longer considered to be utf8-encoded.

89.10.1 Platform Specific Changes and Fixes

- Linux previously had problems related to sockaddrlen when using accept(), revcfrom() (in Perl: recv()), getpeername(), and getsockname().
- Previously DYNIX/ptx had problems in its Configure probe for non-blocking I/O.
- Windows
  - Borland C++ v5.5 is now a supported compiler that can build Perl. However, the generated binaries continue to be incompatible with those generated by the other supported compilers (GCC and Visual C++).
  - Win32::GetCwd() correctly returns C:\ instead of C: when at the drive root. Other bugs in chdir() and Cwd::cwd() have also been fixed.
  - Duping socket handles with open(F, ">&MYSOCK") now works under Windows 9x.
  - HTML files will be installed in c:\perl\html instead of c:\perl\lib\pod\html
  - The makefiles now provide a single switch to bulk-enable all the features enabled in ActiveState ActivePerl (a popular binary distribution).

89.11 New or Changed Diagnostics

Two new debugging options have been added: if you have compiled your Perl with debugging, you can use the -DT and -DR options to trace tokenising and to add reference counts to displaying variables, respectively.

- If an attempt to use a (non-blessed) reference as an array index is made, a warning is given.
- `push @a;` and `unshift @a;` (with no values to push or unshift) now give a warning. This may be a problem for generated and evaled code.

89.12 Changed Internals

- Some new APIs: ptr_table_clear(), ptr_table_free(), sv_setref_uv(). For the full list of the available APIs see *perlapi*.
- dTHR and djSP have been obsoleted; the former removed (because it's a no-op) and the latter replaced with dSP.
- Perl now uses system malloc instead of Perl malloc on all 64-bit platforms, and even in some not-always-64-bit platforms like AIX, IRIX, and Solaris. This change breaks backward compatibility but Perl's malloc has problems with large address spaces and also the speed of vendors' malloc is generally better in large address space machines (Perl's malloc is mostly tuned for space).

89.13 New Tests

Many new tests have been added. The most notable is probably the lib/1_compile: it is very notable because running it takes quite a long time – it test compiles all the Perl modules in the distribution. Please be patient.

89.14 Known Problems

Note that unlike other sections in this document (which describe changes since 5.7.0) this section is cumulative containing known problems for all the 5.7 releases.

89.14.1 AIX vac 5.0.0.0 May Produce Buggy Code For Perl

The AIX C compiler vac version 5.0.0.0 may produce buggy code, resulting in few random tests failing, but when the failing tests are run by hand, they succeed. We suggest upgrading to at least vac version 5.0.1.0, that has been known to compile Perl correctly. "lslpp -L|grep vac.C" will tell you the vac version.

89.14.2 lib/ftmp-security tests warn 'system possibly insecure'

Don't panic. Read INSTALL 'make test' section instead.

89.14.3 lib/io_multihomed Fails In LP64-Configured HP-UX

The lib/io_multihomed test may hang in HP-UX if Perl has been configured to be 64-bit. Because other 64-bit platforms do not hang in this test, HP-UX is suspect. All other tests pass in 64-bit HP-UX. The test attempts to create and connect to "multihomed" sockets (sockets which have multiple IP addresses).

89.14.4 Test lib/posix Subtest 9 Fails In LP64-Configured HP-UX

If perl is configured with -Duse64bitall, the successful result of the subtest 10 of lib/posix may arrive before the successful result of the subtest 9, which confuses the test harness so much that it thinks the subtest 9 failed.

89.14.5 lib/b test 19

The test fails on various platforms (PA64 and IA64 are known), but the exact cause is still being investigated.

89.14.6 Linux With Sfio Fails op/misc Test 48

No known fix.

89.14.7 sigaction test 13 in VMS

The test is known to fail; whether it's because of VMS of because of faulty test is not known.

89.14.8 sprintf tests 129 and 130

The op/sprintf tests 129 and 130 are known to fail on some platforms. Examples include any platform using sfio, and Compaq/Tandem's NonStop-UX. The failing platforms do not comply with the ANSI C Standard, line 19ff on page 134 of ANSI X3.159 1989 to be exact. (They produce something else than "1" and "-1" when formatting 0.6 and -0.6 using the printf format "%.0f", most often they produce "0" and "-0".)

89.14.9 Failure of Thread tests

The subtests 19 and 20 of lib/thr5005.t test are known to fail due to fundamental problems in the 5.005 threading implementation. These are not new failures–Perl 5.005_0x has the same bugs, but didn't have these tests. (Note that support for 5.005-style threading remains experimental.)

89.14.10 Localising a Tied Variable Leaks Memory

```
    use Tie::Hash;
    tie my %tie_hash => 'Tie::StdHash';

    ...

    local($tie_hash{Foo}) = 1; # leaks
```

Code like the above is known to leak memory every time the local() is executed.

89.14.11 Self-tying of Arrays and Hashes Is Forbidden

Self-tying of arrays and hashes is broken in rather deep and hard-to-fix ways. As a stop-gap measure to avoid people from getting frustrated at the mysterious results (core dumps, most often) it is for now forbidden (you will get a fatal error even from an attempt).

89.14.12 Building Extensions Can Fail Because Of Largefiles

Some extensions like mod_perl are known to have issues with 'largefiles', a change brought by Perl 5.6.0 in which file offsets default to 64 bits wide, where supported. Modules may fail to compile at all or compile and work incorrectly. Currently there is no good solution for the problem, but Configure now provides appropriate non-largefile ccflags, ldflags, libswanted, and libs in the %Config hash (e.g., $Config{ccflags_nolargefiles}) so the extensions that are having problems can try configuring themselves without the largefileness. This is admittedly not a clean solution, and the solution may not even work at all. One potential failure is whether one can (or, if one can, whether it's a good idea) link together at all binaries with different ideas about file offsets, all this is platform-dependent.

89.14.13 The Compiler Suite Is Still Experimental

The compiler suite is slowly getting better but is nowhere near working order yet.

89.15 Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup and the perl bug database at http://bugs.perl.org/ There may also be information at http://www.perl.com/perl/ , the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to perlbug@perl.org to be analysed by the Perl porting team.

89.16 SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

89.17 HISTORY

Written by Jarkko Hietaniemi <*jhi@iki.fi*>, with many contributions from The Perl Porters and Perl Users submitting feedback and patches.

Send omissions or corrections to <*perlbug@perl.org*>.

Chapter 90

perl570delta

What's new for perl v5.7.0

90.1 DESCRIPTION

This document describes differences between the 5.6.0 release and the 5.7.0 release.

90.2 Security Vulnerability Closed

A potential security vulnerability in the optional suidperl component of Perl has been identified. suidperl is neither built nor installed by default. As of September the 2nd, 2000, the only known vulnerable platform is Linux, most likely all Linux distributions. CERT and various vendors have been alerted about the vulnerability.

The problem was caused by Perl trying to report a suspected security exploit attempt using an external program, /bin/mail. On Linux platforms the /bin/mail program had an undocumented feature which when combined with suidperl gave access to a root shell, resulting in a serious compromise instead of reporting the exploit attempt. If you don't have /bin/mail, or if you have 'safe setuid scripts', or if suidperl is not installed, you are safe.

The exploit attempt reporting feature has been completely removed from the Perl 5.7.0 release, so that particular vulnerability isn't there anymore. However, further security vulnerabilities are, unfortunately, always possible. The suidperl code is being reviewed and if deemed too risky to continue to be supported, it may be completely removed from future releases. In any case, suidperl should only be used by security experts who know exactly what they are doing and why they are using suidperl instead of some other solution such as sudo (see http://www.courtesan.com/sudo/).

90.3 Incompatible Changes

- Arrays now always interpolate into double-quoted strings: constructs like "foo@bar" now always assume `@bar` is an array, whether or not the compiler has seen use of `@bar`.
- The semantics of bless(REF, REF) were unclear and until someone proves it to make some sense, it is forbidden.
- A reference to a reference now stringify as "REF(0x81485ec)" instead of "SCALAR(0x81485ec)" in order to be more consistent with the return value of ref().
- The very dusty examples in the eg/ directory have been removed. Suggestions for new shiny examples welcome but the main issue is that the examples need to be documented, tested and (most importantly) maintained.
- The obsolete chat2 library that should never have been allowed to escape the laboratory has been decommissioned.
- The unimplemented POSIX regex features [[.cc.]] and [[=c=]] are still recognised but now cause fatal errors. The previous behaviour of ignoring them by default and warning if requested was unacceptable since it, in a way, falsely promised that the features could be used.

- The (bogus) escape sequences \8 and \9 now give an optional warning ("Unrecognized escape passed through"). There is no need to \-escape any `\w` character.
- lstat(FILEHANDLE) now gives a warning because the operation makes no sense. In future releases this may become a fatal error.
- The long deprecated uppercase aliases for the string comparison operators (EQ, NE, LT, LE, GE, GT) have now been removed.
- The regular expression captured submatches ($1, $2, ...) are now more consistently unset if the match fails, instead of leaving false data lying around in them.
- The tr///C and tr///U features have been removed and will not return; the interface was a mistake. Sorry about that. For similar functionality, see pack('U0', ...) and pack('C0', ...).

90.4 Core Enhancements

- `perl -d:Module=arg,arg,arg` now works (previously one couldn't pass in multiple arguments.)
- my __PACKAGE__ $obj now works.
- `no Module;` now works even if there is no "sub unimport" in the Module.
- The numerical comparison operators return `undef` if either operand is a NaN. Previously the behaviour was unspecified.
- `pack('U0a*', ...)` can now be used to force a string to UTF-8.
- prototype(\&) is now available.
- There is now an UNTIE method.

90.5 Modules and Pragmata

90.5.1 New Modules

- File::Temp allows one to create temporary files and directories in an easy, portable, and secure way.
- Storable gives persistence to Perl data structures by allowing the storage and retrieval of Perl data to and from files in a fast and compact binary format.

90.5.2 Updated And Improved Modules and Pragmata

- The following independently supported modules have been updated to newer versions from CPAN: CGI, CPAN, DB_File, File::Spec, Getopt::Long, the podlators bundle, Pod::LaTeX, Pod::Parser, Term::ANSIColor, Test.
- Bug fixes and minor enhancements have been applied to B::Deparse, Data::Dumper, IO::Poll, IO::Socket::INET, Math::BigFloat, Math::Complex, Math::Trig, Net::protoent, the re pragma, SelfLoader, Sys::SysLog, Test::Harness, Text::Wrap, UNIVERSAL, and the warnings pragma.
- The attributes::reftype() now works on tied arguments.
- AutoLoader can now be disabled with `no AutoLoader;`,
- The English module can now be used without the infamous performance hit by saying

```
        use English '-no_performance_hit';
```

(Assuming, of course, that one doesn't need the troublesome variables `` $` ``, `$&`, or `$'`.) Also, introduced `@LAST_MATCH_START` and `@LAST_MATCH_END` English aliases for `@-` and `@+`.

- File::Find now has pre- and post-processing callbacks. It also correctly changes directories when chasing symbolic links. Callbacks (naughtily) exiting with "next;" instead of "return;" now work.
- File::Glob::glob() renamed to File::Glob::bsd_glob() to avoid prototype mismatch with CORE::glob().
- IPC::Open3 now allows the use of numeric file descriptors.
- use lib now works identically to @INC. Removing directories with 'no lib' now works.
- `%INC` now localised in a Safe compartment so that use/require work.
- The Shell module now has an OO interface.

90.6 Utility Changes

- The Emacs perl mode (emacs/cperl-mode.el) has been updated to version 4.31.
- Perlbug is now much more robust. It also sends the bug report to perl.org, not perl.com.
- The perlcc utility has been rewritten and its user interface (that is, command line) is much more like that of the UNIX C compiler, cc.
- The xsubpp utility for extension writers now understands POD documentation embedded in the *.xs files.

90.7 New Documentation

- perl56delta details the changes between the 5.005 release and the 5.6.0 release.
- perldebtut is a Perl debugging tutorial.
- perlebcdic contains considerations for running Perl on EBCDIC platforms. Note that unfortunately EBCDIC platforms that used to supported back in Perl 5.005 are still unsupported by Perl 5.7.0; the plan, however, is to bring them back to the fold.
- perlnewmod tells about writing and submitting a new module.
- perlposix-bc explains using Perl on the POSIX-BC platform (an EBCDIC mainframe platform).
- perlretut is a regular expression tutorial.
- perlrequick is a regular expressions quick-start guide. Yes, much quicker than perlretut.
- perlutil explains the command line utilities packaged with the Perl distribution.

90.8 Performance Enhancements

- map() that changes the size of the list should now work faster.
- sort() has been changed to use mergesort internally as opposed to the earlier quicksort. For very small lists this may result in slightly slower sorting times, but in general the speedup should be at least 20%. Additional bonuses are that the worst case behaviour of sort() is now better (in computer science terms it now runs in time O(N log N), as opposed to quicksort's Theta(N**2) worst-case run time behaviour), and that sort() is now stable (meaning that elements with identical keys will stay ordered as they were before the sort).

90.9 Installation and Configuration Improvements

90.9.1 Generic Improvements

- INSTALL now explains how you can configure Perl to use 64-bit integers even on non-64-bit platforms.
- Policy.sh policy change: if you are reusing a Policy.sh file (see INSTALL) and you use Configure -Dprefix=/foo/bar and in the old Policy $prefix eq $siteprefix and $prefix eq $vendorprefix, all of them will now be changed to the new prefix, /foo/bar. (Previously only $prefix changed.) If you do not like this new behaviour, specify prefix, siteprefix, and vendorprefix explicitly.
- A new optional location for Perl libraries, otherlibdirs, is available. It can be used for example for vendor add-ons without disturbing Perl's own library directories.
- In many platforms the vendor-supplied 'cc' is too stripped-down to build Perl (basically, 'cc' doesn't do ANSI C). If this seems to be the case and 'cc' does not seem to be the GNU C compiler 'gcc', an automatic attempt is made to find and use 'gcc' instead.
- gcc needs to closely track the operating system release to avoid build problems. If Configure finds that gcc was built for a different operating system release than is running, it now gives a clearly visible warning that there may be trouble ahead.
- If binary compatibility with the 5.005 release is not wanted, Configure no longer suggests including the 5.005 modules in @INC.
- Configure `-S` can now run non-interactively.
- configure.gnu now works with options with whitespace in them.
- installperl now outputs everything to STDERR.
- $Config{byteorder} is now computed dynamically (this is more robust with "fat binaries" where an executable image contains binaries for more than one binary platform.)

90.10 Selected Bug Fixes

- Several debugger fixes: exit code now reflects the script exit code, condition `"0"` now treated correctly, the `d` command now checks line number, the `$.` no longer gets corrupted, all debugger output now goes correctly to the socket if RemotePort is set.
- `*foo{FORMAT}` now works.
- Lexical warnings now propagating correctly between scopes.
- Line renumbering with eval and `#line` now works.
- Fixed numerous memory leaks, especially in eval "".
- Modulus of unsigned numbers now works (4063328477 % 65535 used to return 27406, instead of 27047).
- Some "not a number" warnings introduced in 5.6.0 eliminated to be more compatible with 5.005. Infinity is now recognised as a number.
- our() variables will not cause "will not stay shared" warnings.
- pack "Z" now correctly terminates the string with "\0".
- Fix password routines which in some shadow password platforms (e.g. HP-UX) caused getpwent() to return every other entry.
- printf() no longer resets the numeric locale to "C".

- `q(a\\b)` now parses correctly as `'a\\b'`.
- Printing quads (64-bit integers) with printf/sprintf now works without the q L ll prefixes (assuming you are on a quad-capable platform).
- Regular expressions on references and overloaded scalars now work.
- scalar() now forces scalar context even when used in void context.
- sort() arguments are now compiled in the right wantarray context (they were accidentally using the context of the sort() itself).
- Changed the POSIX character class `[[:space:]]` to include the (very rare) vertical tab character. Added a new POSIX-ish character class `[[:blank:]]` which stands for horizontal whitespace (currently, the space and the tab).
- $AUTOLOAD, sort(), lock(), and spawning subprocesses in multiple threads simultaneously are now thread-safe.
- Allow read-only string on left hand side of non-modifying tr///.
- Several Unicode fixes (but still not perfect).
  - BOMs (byte order marks) in the beginning of Perl files (scripts, modules) should now be transparently skipped. UTF-16 (UCS-2) encoded Perl files should now be read correctly.
  - The character tables have been updated to Unicode 3.0.1.
  - chr() for values greater than 127 now create utf8 when under use utf8.
  - Comparing with utf8 data does not magically upgrade non-utf8 data into utf8.
  - `IsAlnum`, `IsAlpha`, and `IsWord` now match titlecase.
  - Concatenation with the `.` operator or via variable interpolation, `eq`, `substr`, `reverse`, `quotemeta`, the `x` operator, substitution with `s///`, single-quoted UTF-8, should now work–in theory.
  - The `tr///` operator now works *slightly* better but is still rather broken. Note that the `tr///CU` functionality has been removed (but see pack('U0', ...)).
  - vec() now refuses to deal with characters >255.
  - Zero entries were missing from the Unicode classes like `IsDigit`.
- UNIVERSAL::isa no longer caches methods incorrectly. (This broke the Tk extension with 5.6.0.)

90.10.1 Platform Specific Changes and Fixes

- BSDI 4.*

  Perl now works on post-4.0 BSD/OSes.
- All BSDs

  Setting `$0` now works (as much as possible; see perlvar for details).
- Cygwin

  Numerous updates; currently synchronised with Cygwin 1.1.4.
- EPOC

  EPOC update after Perl 5.6.0. See README.epoc.
- FreeBSD 3.*

  Perl now works on post-3.0 FreeBSDs.
- HP-UX

  README.hpux updated; `Configure -Duse64bitall` now almost works.

- IRIX

  Numerous compilation flag and hint enhancements; accidental mixing of 32-bit and 64-bit libraries (a doomed attempt) made much harder.

- Linux

  Long doubles should now work (see INSTALL).

- Mac OS Classic

  Compilation of the standard Perl distribution in Mac OS Classic should now work if you have the Metrowerks development environment and the missing Mac-specific toolkit bits. Contact the macperl mailing list for details.

- MPE/iX

  MPE/iX update after Perl 5.6.0. See README.mpeix.

- NetBSD/sparc

  Perl now works on NetBSD/sparc.

- OS/2

  Now works with usethreads (see INSTALL).

- Solaris

  64-bitness using the Sun Workshop compiler now works.

- Tru64 (aka Digital UNIX, aka DEC OSF/1)

  The operating system version letter now recorded in $Config{osvers}. Allow compiling with gcc (previously explicitly forbidden). Compiling with gcc still not recommended because buggy code results, even with gcc 2.95.2.

- Unicos

  Fixed various alignment problems that lead into core dumps either during build or later; no longer dies on math errors at runtime; now using full quad integers (64 bits), previously was using only 46 bit integers for speed.

- VMS

  chdir() now works better despite a CRT bug; now works with MULTIPLICITY (see INSTALL); now works with Perl's malloc.

- Windows

  – accept() no longer leaks memory.
  – Better chdir() return value for a non-existent directory.
  – New %ENV entries now propagate to subprocesses.
  – $ENV{LIB} now used to search for libs under Visual C.
  – A failed (pseudo)fork now returns undef and sets errno to EAGAIN.
  – Allow REG_EXPAND_SZ keys in the registry.
  – Can now send() from all threads, not just the first one.
  – Fake signal handling reenabled, bugs and all.
  – Less stack reserved per thread so that more threads can run concurrently. (Still 16M per thread.)
  – `File::Spec->tmpdir()` now prefers C:/temp over /tmp (works better when perl is running as service).
  – Better UNC path handling under ithreads.
  – wait() and waitpid() now work much better.
  – winsock handle leak fixed.

90.11 New or Changed Diagnostics

All regular expression compilation error messages are now hopefully easier to understand both because the error message now comes before the failed regex and because the point of failure is now clearly marked.

The various "opened only for", "on closed", "never opened" warnings drop the `main::` prefix for filehandles in the `main` package, for example `STDIN` instead of <main::STDIN>.

The "Unrecognized escape" warning has been extended to include `\8`, `\9`, and `\_`. There is no need to escape any of the `\w` characters.

90.12 Changed Internals

- perlapi.pod (a companion to perlguts) now attempts to document the internal API.
- You can now build a really minimal perl called microperl. Building microperl does not require even running Configure; `make -f Makefile.micro` should be enough. Beware: microperl makes many assumptions, some of which may be too bold; the resulting executable may crash or otherwise misbehave in wondrous ways. For careful hackers only.
- Added rsignal(), whichsig(), do_join() to the publicised API.
- Made possible to propagate customised exceptions via croak()ing.
- Added is_utf8_char(), is_utf8_string(), bytes_to_utf8(), and utf8_to_bytes().
- Now xsubs can have attributes just like subs.

90.13 Known Problems

90.13.1 Unicode Support Still Far From Perfect

We're working on it. Stay tuned.

90.13.2 EBCDIC Still A Lost Platform

The plan is to bring them back.

90.13.3 Building Extensions Can Fail Because Of Largefiles

Certain extensions like mod_perl and BSD::Resource are known to have issues with 'largefiles', a change brought by Perl 5.6.0 in which file offsets default to 64 bits wide, where supported. Modules may fail to compile at all or compile and work incorrectly. Currently there is no good solution for the problem, but Configure now provides appropriate non-largefile ccflags, ldflags, libswanted, and libs in the %Config hash (e.g., $Config{ccflags_nolargefiles}) so the extensions that are having problems can try configuring themselves without the largefileness. This is admittedly not a clean solution, and the solution may not even work at all. One potential failure is whether one can (or, if one can, whether it's a good idea) link together at all binaries with different ideas about file offsets, all this is platform-dependent.

90.13.4 ftmp-security tests warn 'system possibly insecure'

Don't panic. Read INSTALL 'make test' section instead.

90.13.5 Test lib/posix Subtest 9 Fails In LP64-Configured HP-UX

If perl is configured with -Duse64bitall, the successful result of the subtest 10 of lib/posix may arrive before the successful result of the subtest 9, which confuses the test harness so much that it thinks the subtest 9 failed.

90.13.6 Long Doubles Still Don't Work In Solaris

The experimental long double support is still very much so in Solaris. (Other platforms like Linux and Tru64 are beginning to solidify in this area.)

90.13.7 Linux With Sfio Fails op/misc Test 48

No known fix.

90.13.8 Storable tests fail in some platforms

If any Storable tests fail the use of Storable is not advisable.

- Many Storable tests fail on AIX configured with 64 bit integers.

  So far unidentified problems break Storable in AIX if Perl is configured to use 64 bit integers. AIX in 32-bit mode works and other 64-bit platforms work with Storable.
- DOS DJGPP may hang when testing Storable.
- st-06compat fails in UNICOS and UNICOS/mk.

  This means that you cannot read old (pre-Storable-0.7) Storable images made in other platforms.
- st-store.t and st-retrieve may fail with Compaq C 6.2 on OpenVMS Alpha 7.2.

90.13.9 Threads Are Still Experimental

Multithreading is still an experimental feature. Some platforms emit the following message for lib/thr5005

```
    #
    # This is a KNOWN FAILURE, and one of the reasons why threading
    # is still an experimental feature.  It is here to stop people
    # from deploying threads in production. ;-)
    #
```

and another known thread-related warning is

```
   pragma/overload......Unbalanced saves: 3 more saves than restores
   panic: magic_mutexfree during global destruction.
   ok
   lib/selfloader.......Unbalanced saves: 3 more saves than restores
   panic: magic_mutexfree during global destruction.
   ok
   lib/st-dclone........Unbalanced saves: 3 more saves than restores
   panic: magic_mutexfree during global destruction.
   ok
```

90.13.10 The Compiler Suite Is Still Experimental

The compiler suite is slowly getting better but is nowhere near working order yet. The backend part that has seen perhaps the most progress is the bytecode compiler.

90.14 Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup and the perl bug database at http://bugs.perl.org/ There may also be information at http://www.perl.com/perl/ , the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to perlbug@perl.org to be analysed by the Perl porting team.

90.15 SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

90.16 HISTORY

Written by Jarkko Hietaniemi <*jhi@iki.fi*>, with many contributions from The Perl Porters and Perl Users submitting feedback and patches.

Send omissions or corrections to <*perlbug@perl.org*>.

Chapter 91

perl561delta

What's new for perl v5.6.x

91.1 DESCRIPTION

This document describes differences between the 5.005 release and the 5.6.1 release.

91.2 Summary of changes between 5.6.0 and 5.6.1

This section contains a summary of the changes between the 5.6.0 release and the 5.6.1 release. More details about the changes mentioned here may be found in the *Changes* files that accompany the Perl source distribution. See *perlhack* for pointers to online resources where you can inspect the individual patches described by these changes.

91.2.1 Security Issues

suidperl will not run /bin/mail anymore, because some platforms have a /bin/mail that is vulnerable to buffer overflow attacks.

Note that suidperl is neither built nor installed by default in any recent version of perl. Use of suidperl is highly discouraged. If you think you need it, try alternatives such as sudo first. See http://www.courtesan.com/sudo/ .

91.2.2 Core bug fixes

This is not an exhaustive list. It is intended to cover only the significant user-visible changes.

`UNIVERSAL::isa()`

A bug in the caching mechanism used by `UNIVERSAL::isa()` that affected base.pm has been fixed. The bug has existed since the 5.005 releases, but wasn't tickled by base.pm in those releases.

Memory leaks

Various cases of memory leaks and attempts to access uninitialized memory have been cured. See §91.14 below for further issues.

Numeric conversions

Numeric conversions did not recognize changes in the string value properly in certain circumstances.

In other situations, large unsigned numbers (those above 2**31) could sometimes lose their unsignedness, causing bogus results in arithmetic operations.

Integer modulus on large unsigned integers sometimes returned incorrect values.

Perl 5.6.0 generated "not a number" warnings on certain conversions where previous versions didn't.

These problems have all been rectified.

Infinity is now recognized as a number.

qw(a\\b)

In Perl 5.6.0, qw(a\\b) produced a string with two backslashes instead of one, in a departure from the behavior in previous versions. The older behavior has been reinstated.

caller()

caller() could cause core dumps in certain situations. Carp was sometimes affected by this problem.

Bugs in regular expressions

Pattern matches on overloaded values are now handled correctly.

Perl 5.6.0 parsed m/\x{ab}/ incorrectly, leading to spurious warnings. This has been corrected.

The RE engine found in Perl 5.6.0 accidentally pessimised certain kinds of simple pattern matches. These are now handled better.

Regular expression debug output (whether through `use re 'debug'` or via `-Dr`) now looks better.

Multi-line matches like `"a\nxb\n" =~ /(?!\A)x/m` were flawed. The bug has been fixed.

Use of $& could trigger a core dump under some situations. This is now avoided.

Match variables $1 et al., weren't being unset when a pattern match was backtracking, and the anomaly showed up inside `/...(?{ ...  })...` etc. These variables are now tracked correctly.

pos() did not return the correct value within s///ge in earlier versions. This is now handled correctly.

"slurp" mode

readline() on files opened in "slurp" mode could return an extra "" at the end in certain situations. This has been corrected.

Autovivification of symbolic references to special variables

Autovivification of symbolic references of special variables described in *perlvar* (as in `${$num}`) was accidentally disabled. This works again now.

Lexical warnings

Lexical warnings now propagate correctly into `eval "..."`.

`use warnings qw(FATAL all)` did not work as intended. This has been corrected.

Lexical warnings could leak into other scopes in some situations. This is now fixed.

warnings::enabled() now reports the state of $^W correctly if the caller isn't using lexical warnings.

Spurious warnings and errors

Perl 5.6.0 could emit spurious warnings about redefinition of dl_error() when statically building extensions into perl. This has been corrected.

"our" variables could result in bogus "Variable will not stay shared" warnings. This is now fixed.

"our" variables of the same name declared in two sibling blocks resulted in bogus warnings about "redeclaration" of the variables. The problem has been corrected.

glob()

Compatibility of the builtin glob() with old csh-based glob has been improved with the addition of GLOB_ALPHASORT option. See `File::Glob`.

File::Glob::glob() has been renamed to File::Glob::bsd_glob() because the name clashes with the builtin glob(). The older name is still available for compatibility, but is deprecated.

Spurious syntax errors generated in certain situations, when glob() caused File::Glob to be loaded for the first time, have been fixed.

Tainting

Some cases of inconsistent taint propagation (such as within hash values) have been fixed.

The tainting behavior of sprintf() has been rationalized. It does not taint the result of floating point formats anymore, making the behavior consistent with that of string interpolation.

sort()

Arguments to sort() weren't being provided the right wantarray() context. The comparison block is now run in scalar context, and the arguments to be sorted are always provided list context.

sort() is also fully reentrant, in the sense that the sort function can itself call sort(). This did not work reliably in previous releases.

#line directives

#line directives now work correctly when they appear at the very beginning of `eval "..."`.

Subroutine prototypes

The (\&) prototype now works properly.

map()

map() could get pathologically slow when the result list it generates is larger than the source list. The performance has been improved for common scenarios.

Debugger

Debugger exit code now reflects the script exit code.

Condition `"0"` in breakpoints is now treated correctly.

The `d` command now checks the line number.

`$.` is no longer corrupted by the debugger.

All debugger output now correctly goes to the socket if RemotePort is set.

PERL5OPT

PERL5OPT can be set to more than one switch group. Previously, it used to be limited to one group of options only.

chop()

chop(@list) in list context returned the characters chopped in reverse order. This has been reversed to be in the right order.

Unicode support

Unicode support has seen a large number of incremental improvements, but continues to be highly experimental. It is not expected to be fully supported in the 5.6.x maintenance releases.

substr(), join(), repeat(), reverse(), quotemeta() and string concatenation were all handling Unicode strings incorrectly in Perl 5.6.0. This has been corrected.

Support for `tr///CU` and `tr///UC` etc., have been removed since we realized the interface is broken. For similar functionality, see `pack` in *perlfunc*.

The Unicode Character Database has been updated to version 3.0.1 with additions made available to the public as of August 30, 2000.

The Unicode character classes \p{Blank} and \p{SpacePerl} have been added. "Blank" is like C isblank(), that is, it contains only "horizontal whitespace" (the space character is, the newline isn't), and the "SpacePerl" is the Unicode equivalent of `\s` (\p{Space} isn't, since that includes the vertical tabulator character, whereas `\s` doesn't.)

If you are experimenting with Unicode support in perl, the development versions of Perl may have more to offer. In particular, I/O layers are now available in the development track, but not in the maintenance track, primarily to do backward compatibility issues. Unicode support is also evolving rapidly on a daily basis in the development track–the maintenance track only reflects the most conservative of these changes.

64-bit support

Support for 64-bit platforms has been improved, but continues to be experimental. The level of support varies greatly among platforms.

Compiler

The B Compiler and its various backends have had many incremental improvements, but they continue to remain highly experimental. Use in production environments is discouraged.

The perlcc tool has been rewritten so that the user interface is much more like that of a C compiler.

The perlbc tools has been removed. Use `perlcc -B` instead.

Lvalue subroutines

There have been various bugfixes to support lvalue subroutines better. However, the feature still remains experimental.

IO::Socket

IO::Socket::INET failed to open the specified port if the service name was not known. It now correctly uses the supplied port number as is.

File::Find

File::Find now chdir()s correctly when chasing symbolic links.

xsubpp

xsubpp now tolerates embedded POD sections.

`no Module;`

`no Module;` does not produce an error even if Module does not have an unimport() method. This parallels the behavior of `use` vis-a-vis `import`.

Tests

A large number of tests have been added.

91.2.3 Core features

untie() will now call an UNTIE() hook if it exists. See *perltie* for details.

The `-DT` command line switch outputs copious tokenizing information. See *perlrun*.

Arrays are now always interpolated in double-quotish strings. Previously, `"foo@bar.com"` used to be a fatal error at compile time, if an array `@bar` was not used or declared. This transitional behavior was intended to help migrate perl4 code, and is deemed to be no longer useful. See §91.3.55.

keys(), each(), pop(), push(), shift(), splice() and unshift() can all be overridden now.

`my __PACKAGE__ $obj` now does the expected thing.

91.2.4 Configuration issues

On some systems (IRIX and Solaris among them) the system malloc is demonstrably better. While the defaults haven't been changed in order to retain binary compatibility with earlier releases, you may be better off building perl with `Configure -Uusemymalloc ...` as discussed in the *INSTALL* file.

`Configure` has been enhanced in various ways:

- Minimizes use of temporary files.
- By default, does not link perl with libraries not used by it, such as the various dbm libraries. SunOS 4.x hints preserve behavior on that platform.
- Support for pdp11-style memory models has been removed due to obsolescence.

- Building outside the source tree is supported on systems that have symbolic links. This is done by running

  ```
  sh /path/to/source/Configure -Dmksymlinks ...
  make all test install
  ```

  in a directory other than the perl source directory. See *INSTALL*.
- `Configure -S` can be run non-interactively.

91.2.5 Documentation

README.aix, README.solaris and README.macos have been added. README.posix-bc has been renamed to README.bs2000. These are installed as *perlaix*, *perlsolaris*, *perlmacos*, and *perlbs2000* respectively.

The following pod documents are brand new:

```
perlclib    Internal replacements for standard C library functions
perldebtut  Perl debugging tutorial
perlebcdic  Considerations for running Perl on EBCDIC platforms
perlnewmod  Perl modules: preparing a new module for distribution
perlrequick Perl regular expressions quick start
perlretut   Perl regular expressions tutorial
perlutil    utilities packaged with the Perl distribution
```

The *INSTALL* file has been expanded to cover various issues, such as 64-bit support.

A longer list of contributors has been added to the source distribution. See the file `AUTHORS`.

Numerous other changes have been made to the included documentation and FAQs.

91.2.6 Bundled modules

The following modules have been added.

B::Concise

Walks Perl syntax tree, printing concise info about ops. See *B::Concise*.

File::Temp

Returns name and handle of a temporary file safely. See *File::Temp*.

Pod::LaTeX

Converts Pod data to formatted LaTeX. See *Pod::LaTeX*.

Pod::Text::Overstrike

Converts POD data to formatted overstrike text. See *Pod::Text::Overstrike*.

The following modules have been upgraded.

CGI

CGI v2.752 is now included.

CPAN

CPAN v1.59_54 is now included.

Class::Struct

Various bugfixes have been added.

DB_File

DB_File v1.75 supports newer Berkeley DB versions, among other improvements.

Devel::Peek

Devel::Peek has been enhanced to support dumping of memory statistics, when perl is built with the included malloc().

File::Find

File::Find now supports pre and post-processing of the files in order to sort() them, etc.

Getopt::Long

Getopt::Long v2.25 is included.

IO::Poll

Various bug fixes have been included.

IPC::Open3

IPC::Open3 allows use of numeric file descriptors.

Math::BigFloat

The fmod() function supports modulus operations. Various bug fixes have also been included.

Math::Complex

Math::Complex handles inf, NaN etc., better.

Net::Ping

ping() could fail on odd number of data bytes, and when the echo service isn't running. This has been corrected.

Opcode

A memory leak has been fixed.

Pod::Parser

Version 1.13 of the Pod::Parser suite is included.

Pod::Text

Pod::Text and related modules have been upgraded to the versions in podlators suite v2.08.

SDBM_File

On dosish platforms, some keys went missing because of lack of support for files with "holes". A workaround for the problem has been added.

Sys::Syslog

Various bug fixes have been included.

Tie::RefHash

Now supports Tie::RefHash::Nestable to automagically tie hashref values.

Tie::SubstrHash

Various bug fixes have been included.

91.2.7 Platform-specific improvements

The following new ports are now available.

NCR MP-RAS

NonStop-UX

Perl now builds under Amdahl UTS.

Perl has also been verified to build under Amiga OS.

Support for EPOC has been much improved. See README.epoc.

Building perl with -Duseithreads or -Duse5005threads now works under HP-UX 10.20 (previously it only worked under 10.30 or later). You will need a thread library package installed. See README.hpux.

Long doubles should now work under Linux.

Mac OS Classic is now supported in the mainstream source package. See README.macos.

Support for MPE/iX has been updated. See README.mpeix.

Support for OS/2 has been improved. See `os2/Changes` and README.os2.

Dynamic loading on z/OS (formerly OS/390) has been improved. See README.os390.

Support for VMS has seen many incremental improvements, including better support for operators like backticks and system(), and better %ENV handling. See `README.vms` and *perlvms*.

Support for Stratus VOS has been improved. See `vos/Changes` and README.vos.

Support for Windows has been improved.

- fork() emulation has been improved in various ways, but still continues to be experimental. See *perlfork* for known bugs and caveats.
- %SIG has been enabled under USE_ITHREADS, but its use is completely unsupported under all configurations.
- Borland C++ v5.5 is now a supported compiler that can build Perl. However, the generated binaries continue to be incompatible with those generated by the other supported compilers (GCC and Visual C++).
- Non-blocking waits for child processes (or pseudo-processes) are supported via `waitpid($pid, &POSIX::WNOHANG)`.
- A memory leak in accept() has been fixed.
- wait(), waitpid() and backticks now return the correct exit status under Windows 9x.
- Trailing new %ENV entries weren't propagated to child processes. This is now fixed.
- Current directory entries in %ENV are now correctly propagated to child processes.
- Duping socket handles with open(F, ">&MYSOCK") now works under Windows 9x.
- The makefiles now provide a single switch to bulk-enable all the features enabled in ActiveState ActivePerl (a popular binary distribution).
- Win32::GetCwd() correctly returns C:\ instead of C: when at the drive root. Other bugs in chdir() and Cwd::cwd() have also been fixed.
- fork() correctly returns undef and sets EAGAIN when it runs out of pseudo-process handles.
- ExtUtils::MakeMaker now uses $ENV{LIB} to search for libraries.
- UNC path handling is better when perl is built to support fork().
- A handle leak in socket handling has been fixed.
- send() works from within a pseudo-process.

Unless specifically qualified otherwise, the remainder of this document covers changes between the 5.005 and 5.6.0 releases.

91.3 Core Enhancements

91.3.1 Interpreter cloning, threads, and concurrency

Perl 5.6.0 introduces the beginnings of support for running multiple interpreters concurrently in different threads. In conjunction with the perl_clone() API call, which can be used to selectively duplicate the state of any given interpreter, it is possible to compile a piece of code once in an interpreter, clone that interpreter one or more times, and run all the resulting interpreters in distinct threads.

On the Windows platform, this feature is used to emulate fork() at the interpreter level. See *perlfork* for details about that.

This feature is still in evolution. It is eventually meant to be used to selectively clone a subroutine and data reachable from that subroutine in a separate interpreter and run the cloned subroutine in a separate thread. Since there is no shared data between the interpreters, little or no locking will be needed (unless parts of the symbol table are explicitly shared). This is obviously intended to be an easy-to-use replacement for the existing threads support.

Support for cloning interpreters and interpreter concurrency can be enabled using the -Dusethreads Configure option (see win32/Makefile for how to enable it on Windows.) The resulting perl executable will be functionally identical to one that was built with -Dmultiplicity, but the perl_clone() API call will only be available in the former.

-Dusethreads enables the cpp macro USE_ITHREADS by default, which in turn enables Perl source code changes that provide a clear separation between the op tree and the data it operates with. The former is immutable, and can therefore be shared between an interpreter and all of its clones, while the latter is considered local to each interpreter, and is therefore copied for each clone.

Note that building Perl with the -Dusemultiplicity Configure option is adequate if you wish to run multiple **independent** interpreters concurrently in different threads. -Dusethreads only provides the additional functionality of the perl_clone() API call and other support for running **cloned** interpreters concurrently.

```
    NOTE: This is an experimental feature.  Implementation details are
    subject to change.
```

91.3.2 Lexically scoped warning categories

You can now control the granularity of warnings emitted by perl at a finer level using the `use warnings` pragma. *warnings* and *perllexwarn* have copious documentation on this feature.

91.3.3 Unicode and UTF-8 support

Perl now uses UTF-8 as its internal representation for character strings. The `utf8` and `bytes` pragmas are used to control this support in the current lexical scope. See *perlunicode*, *utf8* and *bytes* for more information.

This feature is expected to evolve quickly to support some form of I/O disciplines that can be used to specify the kind of input and output data (bytes or characters). Until that happens, additional modules from CPAN will be needed to complete the toolkit for dealing with Unicode.

```
    NOTE: This should be considered an experimental feature.  Implementation
    details are subject to change.
```

91.3.4 Support for interpolating named characters

The new `\N` escape interpolates named characters within strings. For example, `"Hi!  \N{WHITE SMILING FACE}"` evaluates to a string with a Unicode smiley face at the end.

91.3.5 "our" declarations

An "our" declaration introduces a value that can be best understood as a lexically scoped symbolic alias to a global variable in the package that was current where the variable was declared. This is mostly useful as an alternative to the `vars` pragma, but also provides the opportunity to introduce typing and other attributes for such variables. See our in *perlfunc*.

91.3.6 Support for strings represented as a vector of ordinals

Literals of the form `v1.2.3.4` are now parsed as a string composed of characters with the specified ordinals. This is an alternative, more readable way to construct (possibly Unicode) strings instead of interpolating characters, as in `"\x{1}\x{2}\x{3}\x{4}"`. The leading `v` may be omitted if there are more than two ordinals, so `1.2.3` is parsed the same as `v1.2.3`.

Strings written in this form are also useful to represent version "numbers". It is easy to compare such version "numbers" (which are really just plain strings) using any of the usual string comparison operators `eq`, `ne`, `lt`, `gt`, etc., or perform bitwise string operations on them using |, &, etc.

In conjunction with the new `$^V` magic variable (which contains the perl version as a string), such literals can be used as a readable way to check if you're running a particular version of Perl:

```
# this will parse in older versions of Perl also
if ($^V and $^V gt v5.6.0) {
    # new features supported
}
```

`require` and `use` also have some special magic to support such literals. They will be interpreted as a version rather than as a module name:

```
require v5.6.0;             # croak if $^V lt v5.6.0
use v5.6.0;                 # same, but croaks at compile-time
```

Alternatively, the `v` may be omitted if there is more than one dot:

```
require 5.6.0;
use 5.6.0;
```

Also, `sprintf` and `printf` support the Perl-specific format flag `%v` to print ordinals of characters in arbitrary strings:

```
printf "v%vd", $^V;         # prints current version, such as "v5.5.650"
printf "%*vX", ":", $addr;  # formats IPv6 address
printf "%*vb", " ", $bits;  # displays bitstring
```

See Scalar value constructors in *perldata* for additional information.

91.3.7 Improved Perl version numbering system

Beginning with Perl version 5.6.0, the version number convention has been changed to a "dotted integer" scheme that is more commonly found in open source projects.

Maintenance versions of v5.6.0 will be released as v5.6.1, v5.6.2 etc. The next development series following v5.6.0 will be numbered v5.7.x, beginning with v5.7.0, and the next major production release following v5.6.0 will be v5.8.0.

The English module now sets $PERL_VERSION to $^V (a string value) rather than `$]` (a numeric value). (This is a potential incompatibility. Send us a report via perlbug if you are affected by this.)

The v1.2.3 syntax is also now legal in Perl. See Support for strings represented as a vector of ordinals for more on that.

To cope with the new versioning system's use of at least three significant digits for each version component, the method used for incrementing the subversion number has also changed slightly. We assume that versions older than v5.6.0 have been incrementing the subversion component in multiples of 10. Versions after v5.6.0 will increment them by 1. Thus, using the new notation, 5.005_03 is the "same" as v5.5.30, and the first maintenance version following v5.6.0 will be v5.6.1 (which should be read as being equivalent to a floating point value of 5.006_001 in the older format, stored in `$]`).

91.3.8 New syntax for declaring subroutine attributes

Formerly, if you wanted to mark a subroutine as being a method call or as requiring an automatic lock() when it is entered, you had to declare that with a `use attrs` pragma in the body of the subroutine. That can now be accomplished with declaration syntax, like this:

```
    sub mymethod : locked method ;
    ...
    sub mymethod : locked method {
        ...
    }

    sub othermethod :locked :method ;
    ...
    sub othermethod :locked :method {
        ...
    }
```

(Note how only the first `:` is mandatory, and whitespace surrounding the `:` is optional.)

AutoSplit.pm and *SelfLoader.pm* have been updated to keep the attributes with the stubs they provide. See *attributes*.

91.3.9 File and directory handles can be autovivified

Similar to how constructs such as `$x->[0]` autovivify a reference, handle constructors (open(), opendir(), pipe(), socketpair(), sysopen(), socket(), and accept()) now autovivify a file or directory handle 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
    }
```

91.3.10 open() with more than two arguments

If open() is passed three arguments instead of two, the second argument is used as the mode and the third argument is taken to be the file name. This is primarily useful for protecting against unintended magic behavior of the traditional two-argument form. See open in *perlfunc*.

91.3.11 64-bit support

Any platform that has 64-bit integers either

```
        (1) natively as longs or ints
        (2) via special compiler flags
        (3) using long long or int64_t
```

is able to use "quads" (64-bit integers) as follows:

- constants (decimal, hexadecimal, octal, binary) in the code
- arguments to oct() and hex()
- arguments to print(), printf() and sprintf() (flag prefixes ll, L, q)
- printed as such
- pack() and unpack() "q" and "Q" formats
- in basic arithmetics: + - * / % (NOTE: operating close to the limits of the integer values may produce surprising results)
- in bit arithmetics: & | ^ ~ << >> (NOTE: these used to be forced to be 32 bits wide but now operate on the full native width.)
- vec()

Note that unless you have the case (a) you will have to configure and compile Perl using the -Duse64bitint Configure flag.

```
NOTE: The Configure flags -Duselonglong and -Duse64bits have been
deprecated.  Use -Duse64bitint instead.
```

There are actually two modes of 64-bitness: the first one is achieved using Configure -Duse64bitint and the second one using Configure -Duse64bitall. The difference is that the first one is minimal and the second one maximal. The first works in more places than the second.

The `use64bitint` does only as much as is required to get 64-bit integers into Perl (this may mean, for example, using "long longs") while your memory may still be limited to 2 gigabytes (because your pointers could still be 32-bit). Note that the name `64bitint` does not imply that your C compiler will be using 64-bit `int`s (it might, but it doesn't have to): the `use64bitint` means that you will be able to have 64 bits wide scalar values.

The `use64bitall` goes all the way by attempting to switch also integers (if it can), longs (and pointers) to being 64-bit. This may create an even more binary incompatible Perl than -Duse64bitint: the resulting executable may not run at all in a 32-bit box, or you may have to reboot/reconfigure/rebuild your operating system to be 64-bit aware.

Natively 64-bit systems like Alpha and Cray need neither -Duse64bitint nor -Duse64bitall.

Last but not least: note that due to Perl's habit of always using floating point numbers, the quads are still not true integers. When quads overflow their limits (0...18_446_744_073_709_551_615 unsigned, -9_223_372_036_854_775_808...9_223_372_036_854_775_807 signed), they are silently promoted to floating point numbers, after which they will start losing precision (in their lower digits).

```
NOTE: 64-bit support is still experimental on most platforms.
Existing support only covers the LP64 data model.  In particular, the
LLP64 data model is not yet supported.  64-bit libraries and system
APIs on many platforms have not stabilized--your mileage may vary.
```

91.3.12 Large file support

If you have filesystems that support "large files" (files larger than 2 gigabytes), you may now also be able to create and access them from Perl.

```
NOTE: The default action is to enable large file support, if
available on the platform.
```

If the large file support is on, and you have a Fcntl constant O_LARGEFILE, the O_LARGEFILE is automatically added to the flags of sysopen().

Beware that unless your filesystem also supports "sparse files" seeking to umpteen petabytes may be inadvisable.

Note that in addition to requiring a proper file system to do large files you may also need to adjust your per-process (or your per-system, or per-process-group, or per-user-group) maximum filesize limits before running Perl scripts that try to handle large files, especially if you intend to write such files.

Finally, in addition to your process/process group maximum filesize limits, you may have quota limits on your filesystems that stop you (your user id or your user group id) from using large files.

Adjusting your process/user/group/file system/operating system limits is outside the scope of Perl core language. For process limits, you may try increasing the limits using your shell's limits/limit/ulimit command before running Perl. The BSD::Resource extension (not included with the standard Perl distribution) may also be of use, it offers the getrlimit/setrlimit interface that can be used to adjust process resource usage limits, including the maximum filesize limit.

91.3.13 Long doubles

In some systems you may be able to use long doubles to enhance the range and precision of your double precision floating point numbers (that is, Perl's numbers). Use Configure -Duselongdouble to enable this support (if it is available).

91.3.14 "more bits"

You can "Configure -Dusemorebits" to turn on both the 64-bit support and the long double support.

91.3.15 Enhanced support for sort() subroutines

Perl subroutines with a prototype of `($$)`, and XSUBs in general, can now be used as sort subroutines. In either case, the two elements to be compared are passed as normal parameters in @_. See sort in *perlfunc*.

For unprototyped sort subroutines, the historical behavior of passing the elements to be compared as the global variables $a and $b remains unchanged.

91.3.16 `sort $coderef @foo` allowed

sort() did not accept a subroutine reference as the comparison function in earlier versions. This is now permitted.

91.3.17 File globbing implemented internally

Perl now uses the File::Glob implementation of the glob() operator automatically. This avoids using an external csh process and the problems associated with it.

```
    NOTE: This is currently an experimental feature.  Interfaces and
    implementation are subject to change.
```

91.3.18 Support for CHECK blocks

In addition to `BEGIN`, `INIT`, `END`, `DESTROY` and `AUTOLOAD`, subroutines named `CHECK` are now special. These are queued up during compilation and behave similar to END blocks, except they are called at the end of compilation rather than at the end of execution. They cannot be called directly.

91.3.19 POSIX character class syntax [: :] supported

For example to match alphabetic characters use /[[:alpha:]]/. See *perlre* for details.

91.3.20 Better pseudo-random number generator

In 5.005_0x and earlier, perl's rand() function used the C library rand(3) function. As of 5.005_52, Configure tests for drand48(), random(), and rand() (in that order) and picks the first one it finds.

These changes should result in better random numbers from rand().

91.3.21 Improved `qw//` operator

The `qw//` operator is now evaluated at compile time into a true list instead of being replaced with a run time call to `split()`. This removes the confusing misbehaviour of `qw//` in scalar context, which had inherited that behaviour from split().

Thus:

```
    $foo = ($bar) = qw(a b c); print "$foo|$bar\n";
```

now correctly prints "3|a", instead of "2|a".

91.3.22 Better worst-case behavior of hashes

Small changes in the hashing algorithm have been implemented in order to improve the distribution of lower order bits in the hashed value. This is expected to yield better performance on keys that are repeated sequences.

91.3.23 pack() format 'Z' supported

The new format type 'Z' is useful for packing and unpacking null-terminated strings. See pack in *perlfunc*.

91.3.24 pack() format modifier '!' supported

The new format type modifier '!' is useful for packing and unpacking native shorts, ints, and longs. See pack in *perlfunc*.

91.3.25 pack() and unpack() support counted strings

The template character '/' can be used to specify a counted string type to be packed or unpacked. See pack in *perlfunc*.

91.3.26 Comments in pack() templates

The '#' character in a template introduces a comment up to end of the line. This facilitates documentation of pack() templates.

91.3.27 Weak references

In previous versions of Perl, you couldn't cache objects so as to allow them to be deleted if the last reference from outside the cache is deleted. The reference in the cache would hold a reference count on the object and the objects would never be destroyed.

Another familiar problem is with circular references. When an object references itself, its reference count would never go down to zero, and it would not get destroyed until the program is about to exit.

Weak references solve this by allowing you to "weaken" any reference, that is, make it not count towards the reference count. When the last non-weak reference to an object is deleted, the object is destroyed and all the weak references to the object are automatically undef-ed.

To use this feature, you need the Devel::WeakRef package from CPAN, which contains additional documentation.

```
    NOTE: This is an experimental feature.  Details are subject to change.
```

91.3.28 Binary numbers supported

Binary numbers are now supported as literals, in s?printf formats, and `oct()`:

```
    $answer = 0b101010;
    printf "The answer is: %b\n", oct("0b101010");
```

91.3.29 Lvalue subroutines

Subroutines can now return modifiable lvalues. See Lvalue subroutines in *perlsub*.

```
    NOTE: This is an experimental feature.  Details are subject to change.
```

91.3.30 Some arrows may be omitted in calls through references

Perl now allows the arrow to be omitted in many constructs involving subroutine calls through references. For example, `$foo[10]->('foo')` may now be written `$foo[10]('foo')`. This is rather similar to how the arrow may be omitted from `$foo[10]->{'foo'}`. Note however, that the arrow is still required for `foo(10)->('bar')`.

91.3.31 Boolean assignment operators are legal lvalues

Constructs such as `($a ||= 2) += 1` are now allowed.

91.3.32 exists() is supported on subroutine names

The exists() builtin now works on subroutine names. A subroutine is considered to exist if it has been declared (even if implicitly). See exists in *perlfunc* for examples.

91.3.33 exists() and delete() are supported on array elements

The exists() and delete() builtins now work on simple arrays as well. The behavior is similar to that on hash elements.

exists() can be used to check whether an array element has been initialized. This avoids autovivifying array elements that don't exist. If the array is tied, the EXISTS() method in the corresponding tied package will be invoked.

delete() may be used to remove an element from the array and return it. The array element at that position returns to its uninitialized state, so that testing for the same element with exists() will return false. If the element happens to be the one at the end, the size of the array also shrinks up to the highest element that tests true for exists(), or 0 if none such is found. If the array is tied, the DELETE() method in the corresponding tied package will be invoked.

See exists in *perlfunc* and delete in *perlfunc* for examples.

91.3.34 Pseudo-hashes work better

Dereferencing some types of reference values in a pseudo-hash, such as `$ph->{foo}[1]`, was accidentally disallowed. This has been corrected.

When applied to a pseudo-hash element, exists() now reports whether the specified value exists, not merely if the key is valid.

delete() now works on pseudo-hashes. When given a pseudo-hash element or slice it deletes the values corresponding to the keys (but not the keys themselves). See Pseudo-hashes: Using an array as a hash in *perlref*.

Pseudo-hash slices with constant keys are now optimized to array lookups at compile-time.

List assignments to pseudo-hash slices are now supported.

The `fields` pragma now provides ways to create pseudo-hashes, via fields::new() and fields::phash(). See *fields*.

```
    NOTE: The pseudo-hash data type continues to be experimental.
    Limiting oneself to the interface elements provided by the
    fields pragma will provide protection from any future changes.
```

91.3.35 Automatic flushing of output buffers

fork(), exec(), system(), qx//, and pipe open()s now flush buffers of all files opened for output when the operation was attempted. This mostly eliminates confusing buffering mishaps suffered by users unaware of how Perl internally handles I/O.

This is not supported on some platforms like Solaris where a suitably correct implementation of fflush(NULL) isn't available.

91.3.36 Better diagnostics on meaningless filehandle operations

Constructs such as `open(<FH>)` and `close(<FH>)` are compile time errors. Attempting to read from filehandles that were opened only for writing will now produce warnings (just as writing to read-only filehandles does).

91.3.37 Where possible, buffered data discarded from duped input filehandle

`open(NEW, "<&OLD")` now attempts to discard any data that was previously read and buffered in `OLD` before duping the handle. On platforms where doing this is allowed, the next read operation on `NEW` will return the same data as the corresponding operation on `OLD`. Formerly, it would have returned the data from the start of the following disk block instead.

91.3.38 eof() has the same old magic as <>

`eof()` would return true if no attempt to read from <> had yet been made. `eof()` has been changed to have a little magic of its own, it now opens the <> files.

91.3.39 binmode() can be used to set :crlf and :raw modes

binmode() now accepts a second argument that specifies a discipline for the handle in question. The two pseudo-disciplines ":raw" and ":crlf" are currently supported on DOS-derivative platforms. See binmode in *perlfunc* and *open*.

91.3.40 `-T` filetest recognizes UTF-8 encoded files as "text"

The algorithm used for the `-T` filetest has been enhanced to correctly identify UTF-8 content as "text".

91.3.41 system(), backticks and pipe open now reflect exec() failure

On Unix and similar platforms, system(), qx() and open(FOO, "cmd |") etc., are implemented via fork() and exec(). When the underlying exec() fails, earlier versions did not report the error properly, since the exec() happened to be in a different process.

The child process now communicates with the parent about the error in launching the external command, which allows these constructs to return with their usual error value and set $!.

91.3.42 Improved diagnostics

Line numbers are no longer suppressed (under most likely circumstances) during the global destruction phase.

Diagnostics emitted from code running in threads other than the main thread are now accompanied by the thread ID.

Embedded null characters in diagnostics now actually show up. They used to truncate the message in prior versions.

$foo::a and $foo::b are now exempt from "possible typo" warnings only if sort() is encountered in package `foo`.

Unrecognized alphabetic escapes encountered when parsing quote constructs now generate a warning, since they may take on new semantics in later versions of Perl.

Many diagnostics now report the internal operation in which the warning was provoked, like so:

```
    Use of uninitialized value in concatenation (.) at (eval 1) line 1.
    Use of uninitialized value in print at (eval 1) line 1.
```

Diagnostics that occur within eval may also report the file and line number where the eval is located, in addition to the eval sequence number and the line number within the evaluated text itself. For example:

```
    Not enough arguments for scalar at (eval 4)[newlib/perl5db.pl:1411] line 2, at EOF
```

91.3.43 Diagnostics follow STDERR

Diagnostic output now goes to whichever file the `STDERR` handle is pointing at, instead of always going to the underlying C runtime library's `stderr`.

91.3.44 More consistent close-on-exec behavior

On systems that support a close-on-exec flag on filehandles, the flag is now set for any handles created by pipe(), socketpair(), socket(), and accept(), if that is warranted by the value of $^F that may be in effect. Earlier versions neglected to set the flag for handles created with these operators. See pipe in *perlfunc*, socketpair in *perlfunc*, socket in *perlfunc*, accept in *perlfunc*, and $^F in *perlvar*.

91.3.45 syswrite() ease-of-use

The length argument of `syswrite()` has become optional.

91.3.46 Better syntax checks on parenthesized unary operators

Expressions such as:

```
    print defined(&foo,&bar,&baz);
    print uc("foo","bar","baz");
    undef($foo,&bar);
```

used to be accidentally allowed in earlier versions, and produced unpredictable behaviour. Some produced ancillary warnings when used in this way; others silently did the wrong thing.

The parenthesized forms of most unary operators that expect a single argument now ensure that they are not called with more than one argument, making the cases shown above syntax errors. The usual behaviour of:

```
    print defined &foo, &bar, &baz;
    print uc "foo", "bar", "baz";
    undef $foo, &bar;
```

remains unchanged. See *perlop*.

91.3.47 Bit operators support full native integer width

The bit operators (& | ^ ~ << >>) now operate on the full native integral width (the exact size of which is available in $Config{ivsize}). For example, if your platform is either natively 64-bit or if Perl has been configured to use 64-bit integers, these operations apply to 8 bytes (as opposed to 4 bytes on 32-bit platforms). For portability, be sure to mask off the excess bits in the result of unary ~, e.g., `~$x & 0xffffffff`.

91.3.48 Improved security features

More potentially unsafe operations taint their results for improved security.

The `passwd` and `shell` fields returned by the getpwent(), getpwnam(), and getpwuid() are now tainted, because the user can affect their own encrypted password and login shell.

The variable modified by shmread(), and messages returned by msgrcv() (and its object-oriented interface IPC::SysV::Msg::rcv) are also tainted, because other untrusted processes can modify messages and shared memory segments for their own nefarious purposes.

91.3.49 More functional bareword prototype (*)

Bareword prototypes have been rationalized to enable them to be used to override builtins that accept barewords and interpret them in a special way, such as `require` or `do`.

Arguments prototyped as * will now be visible within the subroutine as either a simple scalar or as a reference to a typeglob. See Prototypes in *perlsub*.

91.3.50 `require` and do may be overridden

`require` and `do 'file'` operations may be overridden locally by importing subroutines of the same name into the current package (or globally by importing them into the CORE::GLOBAL:: namespace). Overriding `require` will also affect `use`, provided the override is visible at compile-time. See Overriding Built-in Functions in *perlsub*.

91.3.51 $ ˆX variables may now have names longer than one character

Formerly, $ˆX was synonymous with ${"\cX"}, but $ˆXY was a syntax error. Now variable names that begin with a control character may be arbitrarily long. However, for compatibility reasons, these variables *must* be written with explicit braces, as `${ˆXY}` for example. `${ˆXYZ}` is synonymous with ${"\cXYZ"}. Variable names with more than one control character, such as `${ˆXYˆZ}`, are illegal.

The old syntax has not changed. As before, 'ˆX' may be either a literal control-X character or the two-character sequence 'caret' plus 'X'. When braces are omitted, the variable name stops after the control character. Thus `"$ˆXYZ"` continues to be synonymous with `$ˆX . "YZ"` as before.

As before, lexical variables may not have names beginning with control characters. As before, variables whose names begin with a control character are always forced to be in package 'main'. All such variables are reserved for future extensions, except those that begin with `ˆ_`, which may be used by user programs and are guaranteed not to acquire special meaning in any future version of Perl.

91.3.52 New variable $ ˆC reflects `-c` switch

`$ˆC` has a boolean value that reflects whether perl is being run in compile-only mode (i.e. via the `-c` switch). Since BEGIN blocks are executed under such conditions, this variable enables perl code to determine whether actions that make sense only during normal running are warranted. See *perlvar*.

91.3.53 New variable $ ˆV contains Perl version as a string

`$ˆV` contains the Perl version number as a string composed of characters whose ordinals match the version numbers, i.e. v5.6.0. This may be used in string comparisons.

See `Support for strings represented as a vector of ordinals` for an example.

91.3.54 Optional Y2K warnings

If Perl is built with the cpp macro `PERL_Y2KWARN` defined, it emits optional warnings when concatenating the number 19 with another number.

This behavior must be specifically enabled when running Configure. See *INSTALL* and *README.Y2K*.

91.3.55 Arrays now always interpolate into double-quoted strings

In double-quoted strings, arrays now interpolate, no matter what. The behavior in earlier versions of perl 5 was that arrays would interpolate into strings if the array had been mentioned before the string was compiled, and otherwise Perl would raise a fatal compile-time error. In versions 5.000 through 5.003, the error was

```
        Literal @example now requires backslash
```

In versions 5.004_01 through 5.6.0, the error was

```
        In string, @example now must be written as \@example
```

The idea here was to get people into the habit of writing `"fred\@example.com"` when they wanted a literal `@` sign, just as they have always written `"Give me back my \$5"` when they wanted a literal `$` sign.

Starting with 5.6.1, when Perl now sees an `@` sign in a double-quoted string, it *always* attempts to interpolate an array, regardless of whether or not the array has been used or declared already. The fatal error has been downgraded to an optional warning:

```
        Possible unintended interpolation of @example in string
```

This warns you that `"fred@example.com"` is going to turn into `fred.com` if you don't backslash the `@`. See http://www.plover.com/˜mjd/perl/at-error.html for more details about the history here.

91.4 Modules and Pragmata

91.4.1 Modules

attributes

While used internally by Perl as a pragma, this module also provides a way to fetch subroutine and variable attributes. See *attributes*.

B

The Perl Compiler suite has been extensively reworked for this release. More of the standard Perl testsuite passes when run under the Compiler, but there is still a significant way to go to achieve production quality compiled executables.

```
    NOTE: The Compiler suite remains highly experimental.  The
    generated code may not be correct, even when it manages to execute
    without errors.
```

Benchmark

Overall, Benchmark results exhibit lower average error and better timing accuracy.

You can now run tests for *n* seconds instead of guessing the right number of tests to run: e.g., timethese(-5, ...) will run each code for at least 5 CPU seconds. Zero as the "number of repetitions" means "for at least 3 CPU seconds". The output format has also changed. For example:

```
   use Benchmark;$x=3;timethese(-5,{a=>sub{$x*$x},b=>sub{$x**2}})
```

will now output something like this:

```
   Benchmark: running a, b, each for at least 5 CPU seconds...
            a:  5 wallclock secs ( 5.77 usr +  0.00 sys =  5.77 CPU) @ 200551.91/s (n=1156516)
            b:  4 wallclock secs ( 5.00 usr +  0.02 sys =  5.02 CPU) @ 159605.18/s (n=800686)
```

New features: "each for at least N CPU seconds...", "wallclock secs", and the "@ operations/CPU second (n=operations)".

timethese() now returns a reference to a hash of Benchmark objects containing the test results, keyed on the names of the tests.

timethis() now returns the iterations field in the Benchmark result object instead of 0.

timethese(), timethis(), and the new cmpthese() (see below) can also take a format specifier of 'none' to suppress output.

A new function countit() is just like timeit() except that it takes a TIME instead of a COUNT.

A new function cmpthese() prints a chart comparing the results of each test returned from a timethese() call. For each possible pair of tests, the percentage speed difference (iters/sec or seconds/iter) is shown.

For other details, see *Benchmark*.

ByteLoader

The ByteLoader is a dedicated extension to generate and run Perl bytecode. See *ByteLoader*.

constant

References can now be used.

The new version also allows a leading underscore in constant names, but disallows a double leading underscore (as in "__LINE__"). Some other names are disallowed or warned against, including BEGIN, END, etc. Some names which were forced into main:: used to fail silently in some cases; now they're fatal (outside of main::) and an optional warning (inside of main::). The ability to detect whether a constant had been set with a given name has been added.

See *constant*.

charnames

This pragma implements the `\N` string escape. See *charnames*.

Data::Dumper

A `Maxdepth` setting can be specified to avoid venturing too deeply into deep data structures. See *Data::Dumper*.

The XSUB implementation of Dump() is now automatically called if the `Useqq` setting is not in use.

Dumping `qr//` objects works correctly.

DB

`DB` is an experimental module that exposes a clean abstraction to Perl's debugging API.

DB_File

DB_File can now be built with Berkeley DB versions 1, 2 or 3. See `ext/DB_File/Changes`.

Devel::DProf

Devel::DProf, a Perl source code profiler has been added. See *Devel::DProf* and *dprofpp*.

Devel::Peek

The Devel::Peek module provides access to the internal representation of Perl variables and data. It is a data debugging tool for the XS programmer.

Dumpvalue

The Dumpvalue module provides screen dumps of Perl data.

DynaLoader

DynaLoader now supports a dl_unload_file() function on platforms that support unloading shared objects using dlclose().

Perl can also optionally arrange to unload all extension shared objects loaded by Perl. To enable this, build Perl with the Configure option `-Accflags=-DDL_UNLOAD_ALL_AT_EXIT`. (This maybe useful if you are using Apache with mod_perl.)

English

$PERL_VERSION now stands for `$^V` (a string value) rather than for `$]` (a numeric value).

Env

Env now supports accessing environment variables like PATH as array variables.

Fcntl

More Fcntl constants added: F_SETLK64, F_SETLKW64, O_LARGEFILE for large file (more than 4GB) access (NOTE: the O_LARGEFILE is automatically added to sysopen() flags if large file support has been configured, as is the default), Free/Net/OpenBSD locking behaviour flags F_FLOCK, F_POSIX, Linux F_SHLCK, and O_ACCMODE: the combined mask of O_RDONLY, O_WRONLY, and O_RDWR. The seek()/sysseek() constants SEEK_SET, SEEK_CUR, and SEEK_END are available via the `:seek` tag. The chmod()/stat() S_IF* constants and S_IS* functions are available via the `:mode` tag.

File::Compare

A compare_text() function has been added, which allows custom comparison functions. See *File::Compare*.

File::Find

File::Find now works correctly when the wanted() function is either autoloaded or is a symbolic reference.

A bug that caused File::Find to lose track of the working directory when pruning top-level directories has been fixed.

File::Find now also supports several other options to control its behavior. It can follow symbolic links if the `follow` option is specified. Enabling the `no_chdir` option will make File::Find skip changing the current directory when walking directories. The `untaint` flag can be useful when running with taint checks enabled.

See *File::Find*.

File::Glob

This extension implements BSD-style file globbing. By default, it will also be used for the internal implementation of the glob() operator. See *File::Glob*.

File::Spec

New methods have been added to the File::Spec module: devnull() returns the name of the null device (/dev/null on Unix) and tmpdir() the name of the temp directory (normally /tmp on Unix). There are now also methods to convert between absolute and relative filenames: abs2rel() and rel2abs(). For compatibility with operating systems that specify volume names in file paths, the splitpath(), splitdir(), and catdir() methods have been added.

File::Spec::Functions

The new File::Spec::Functions modules provides a function interface to the File::Spec module. Allows shorthand

```
    $fullname = catfile($dir1, $dir2, $file);
```

instead of

```
    $fullname = File::Spec->catfile($dir1, $dir2, $file);
```

Getopt::Long

Getopt::Long licensing has changed to allow the Perl Artistic License as well as the GPL. It used to be GPL only, which got in the way of non-GPL applications that wanted to use Getopt::Long.

Getopt::Long encourages the use of Pod::Usage to produce help messages. For example:

```
    use Getopt::Long;
    use Pod::Usage;
    my $man = 0;
    my $help = 0;
    GetOptions('help|?' => \$help, man => \$man) or pod2usage(2);
    pod2usage(1) if $help;
    pod2usage(-exitstatus => 0, -verbose => 2) if $man;
```

```
__END__

=head1 NAME

sample - Using Getopt::Long and Pod::Usage

=head1 SYNOPSIS

sample [options] [file ...]

 Options:
   -help            brief help message
   -man             full documentation

=head1 OPTIONS

=over 8

=item B<-help>

Print a brief help message and exits.

=item B<-man>

Prints the manual page and exits.

=back

=head1 DESCRIPTION

B<This program> will read the given input file(s) and do something
useful with the contents thereof.

=cut
```

See *Pod::Usage* for details.

A bug that prevented the non-option call-back <> from being specified as the first argument has been fixed.

To specify the characters < and > as option starters, use ><. Note, however, that changing option starters is strongly deprecated.

IO

write() and syswrite() will now accept a single-argument form of the call, for consistency with Perl's syswrite().

You can now create a TCP-based IO::Socket::INET without forcing a connect attempt. This allows you to configure its options (like making it non-blocking) and then call connect() manually.

A bug that prevented the IO::Socket::protocol() accessor from ever returning the correct value has been corrected.

IO::Socket::connect now uses non-blocking IO instead of alarm() to do connect timeouts.

IO::Socket::accept now uses select() instead of alarm() for doing timeouts.

IO::Socket::INET->new now sets $! correctly on failure. $@ is still set for backwards compatibility.

JPL

Java Perl Lingo is now distributed with Perl. See jpl/README for more information.

lib

`use lib` now weeds out any trailing duplicate entries. `no lib` removes all named entries.

Math::BigInt

The bitwise operations <<, >>, &, |, and ~ are now supported on bigints.

Math::Complex

The accessor methods Re, Im, arg, abs, rho, and theta can now also act as mutators (accessor $z->Re(), mutator $z->Re(3)).

The class method `display_format` and the corresponding object method `display_format`, in addition to accepting just one argument, now can also accept a parameter hash. Recognized keys of a parameter hash are `"style"`, which corresponds to the old one parameter case, and two new parameters: `"format"`, which is a printf()-style format string (defaults usually to `"%.15g"`, you can revert to the default by setting the format string to `undef`) used for both parts of a complex number, and `"polar_pretty_print"` (defaults to true), which controls whether an attempt is made to try to recognize small multiples and rationals of pi (2pi, pi/2) at the argument (angle) of a polar complex number.

The potentially disruptive change is that in list context both methods now *return the parameter hash*, instead of only the value of the `"style"` parameter.

Math::Trig

A little bit of radial trigonometry (cylindrical and spherical), radial coordinate conversions, and the great circle distance were added.

Pod::Parser, Pod::InputObjects

Pod::Parser is a base class for parsing and selecting sections of pod documentation from an input stream. This module takes care of identifying pod paragraphs and commands in the input and hands off the parsed paragraphs and commands to user-defined methods which are free to interpret or translate them as they see fit.

Pod::InputObjects defines some input objects needed by Pod::Parser, and for advanced users of Pod::Parser that need more about a command besides its name and text.

As of release 5.6.0 of Perl, Pod::Parser is now the officially sanctioned "base parser code" recommended for use by all pod2xxx translators. Pod::Text (pod2text) and Pod::Man (pod2man) have already been converted to use Pod::Parser and efforts to convert Pod::HTML (pod2html) are already underway. For any questions or comments about pod parsing and translating issues and utilities, please use the pod-people@perl.org mailing list.

For further information, please see *Pod::Parser* and *Pod::InputObjects*.

Pod::Checker, podchecker

This utility checks pod files for correct syntax, according to *perlpod*. Obvious errors are flagged as such, while warnings are printed for mistakes that can be handled gracefully. The checklist is not complete yet. See *Pod::Checker*.

Pod::ParseUtils, Pod::Find

These modules provide a set of gizmos that are useful mainly for pod translators. Pod::Find traverses directory structures and returns found pod files, along with their canonical names (like `File::Spec::Unix`). Pod::ParseUtils contains **Pod::List** (useful for storing pod list information), **Pod::Hyperlink** (for parsing the contents of `L<>` sequences) and **Pod::Cache** (for caching information about pod files, e.g., link nodes).

Pod::Select, podselect

Pod::Select is a subclass of Pod::Parser which provides a function named "podselect()" to filter out user-specified sections of raw pod documentation from an input stream. podselect is a script that provides access to Pod::Select from other scripts to be used as a filter. See *Pod::Select*.

Pod::Usage, pod2usage

Pod::Usage provides the function "pod2usage()" to print usage messages for a Perl script based on its embedded pod documentation. The pod2usage() function is generally useful to all script authors since it lets them write and maintain a single source (the pods) for documentation, thus removing the need to create and maintain redundant usage message text consisting of information already in the pods.

There is also a pod2usage script which can be used from other kinds of scripts to print usage messages from pods (even for non-Perl scripts with pods embedded in comments).

For details and examples, please see *Pod::Usage*.

Pod::Text and Pod::Man

Pod::Text has been rewritten to use Pod::Parser. While pod2text() is still available for backwards compatibility, the module now has a new preferred interface. See *Pod::Text* for the details. The new Pod::Text module is easily subclassed for tweaks to the output, and two such subclasses (Pod::Text::Termcap for man-page-style bold and underlining using termcap information, and Pod::Text::Color for markup with ANSI color sequences) are now standard.

pod2man has been turned into a module, Pod::Man, which also uses Pod::Parser. In the process, several outstanding bugs related to quotes in section headers, quoting of code escapes, and nested lists have been fixed. pod2man is now a wrapper script around this module.

SDBM_File

An EXISTS method has been added to this module (and sdbm_exists() has been added to the underlying sdbm library), so one can now call exists on an SDBM_File tied hash and get the correct result, rather than a runtime error.

A bug that may have caused data loss when more than one disk block happens to be read from the database in a single FETCH() has been fixed.

Sys::Syslog

Sys::Syslog now uses XSUBs to access facilities from syslog.h so it no longer requires syslog.ph to exist.

Sys::Hostname

Sys::Hostname now uses XSUBs to call the C library's gethostname() or uname() if they exist.

Term::ANSIColor

Term::ANSIColor is a very simple module to provide easy and readable access to the ANSI color and highlighting escape sequences, supported by most ANSI terminal emulators. It is now included standard.

Time::Local

The timelocal() and timegm() functions used to silently return bogus results when the date fell outside the machine's integer range. They now consistently croak() if the date falls in an unsupported range.

Win32

The error return value in list context has been changed for all functions that return a list of values. Previously these functions returned a list with a single element `undef` if an error occurred. Now these functions return the empty list in these situations. This applies to the following functions:

```
    Win32::FsType
    Win32::GetOSVersion
```

The remaining functions are unchanged and continue to return `undef` on error even in list context.

The Win32::SetLastError(ERROR) function has been added as a complement to the Win32::GetLastError() function.

The new Win32::GetFullPathName(FILENAME) returns the full absolute pathname for FILENAME in scalar context. In list context it returns a two-element list containing the fully qualified directory name and the filename. See *Win32*.

XSLoader

The XSLoader extension is a simpler alternative to DynaLoader. See *XSLoader*.

DBM Filters

A new feature called "DBM Filters" has been added to all the DBM modules–DB_File, GDBM_File, NDBM_File, ODBM_File, and SDBM_File. DBM Filters add four new methods to each DBM module:

```
filter_store_key
filter_store_value
filter_fetch_key
filter_fetch_value
```

These can be used to filter key-value pairs before the pairs are written to the database or just after they are read from the database. See *perldbmfilter* for further information.

91.4.2 Pragmata

`use attrs` is now obsolete, and is only provided for backward-compatibility. It's been replaced by the `sub : attributes` syntax. See Subroutine Attributes in *perlsub* and *attributes*.

Lexical warnings pragma, `use warnings;`, to control optional warnings. See *perllexwarn*.

`use filetest` to control the behaviour of filetests (`-r` `-w` ...). Currently only one subpragma implemented, "use filetest 'access';", that uses access(2) or equivalent to check permissions instead of using stat(2) as usual. This matters in filesystems where there are ACLs (access control lists): the stat(2) might lie, but access(2) knows better.

The `open` pragma can be used to specify default disciplines for handle constructors (e.g. open()) and for qx//. The two pseudo-disciplines `:raw` and `:crlf` are currently supported on DOS-derivative platforms (i.e. where binmode is not a no-op). See also §91.3.39.

91.5 Utility Changes

91.5.1 dprofpp

`dprofpp` is used to display profile data generated using `Devel::DProf`. See *dprofpp*.

91.5.2 find2perl

The `find2perl` utility now uses the enhanced features of the File::Find module. The -depth and -follow options are supported. Pod documentation is also included in the script.

91.5.3 h2xs

The `h2xs` tool can now work in conjunction with `C::Scan` (available from CPAN) to automatically parse real-life header files. The `-M`, `-a`, `-k`, and `-o` options are new.

91.5.4 perlcc

`perlcc` now supports the C and Bytecode backends. By default, it generates output from the simple C backend rather than the optimized C backend.

Support for non-Unix platforms has been improved.

91.5.5 perldoc

`perldoc` has been reworked to avoid possible security holes. It will not by default let itself be run as the superuser, but you may still use the **-U** switch to try to make it drop privileges first.

91.5.6 The Perl Debugger

Many bug fixes and enhancements were added to *perl5db.pl*, the Perl debugger. The help documentation was rearranged. New commands include `< ?`, `> ?`, and `{ ?` to list out current actions, `man` *`docpage`* to run your doc viewer on some perl docset, and support for quoted options. The help information was rearranged, and should be viewable once again if you're using **less** as your pager. A serious security hole was plugged–you should immediately remove all older versions of the Perl debugger as installed in previous releases, all the way back to perl3, from your system to avoid being bitten by this.

91.6 Improved Documentation

Many of the platform-specific README files are now part of the perl installation. See *perl* for the complete list.

perlapi.pod

The official list of public Perl API functions.

perlboot.pod

A tutorial for beginners on object-oriented Perl.

perlcompile.pod

An introduction to using the Perl Compiler suite.

perldbmfilter.pod

A howto document on using the DBM filter facility.

perldebug.pod

All material unrelated to running the Perl debugger, plus all low-level guts-like details that risked crushing the casual user of the debugger, have been relocated from the old manpage to the next entry below.

perldebguts.pod

This new manpage contains excessively low-level material not related to the Perl debugger, but slightly related to debugging Perl itself. It also contains some arcane internal details of how the debugging process works that may only be of interest to developers of Perl debuggers.

perlfork.pod

Notes on the fork() emulation currently available for the Windows platform.

perlfilter.pod

An introduction to writing Perl source filters.

perlhack.pod

Some guidelines for hacking the Perl source code.

perlintern.pod

A list of internal functions in the Perl source code. (List is currently empty.)

perllexwarn.pod

Introduction and reference information about lexically scoped warning categories.

perlnumber.pod

Detailed information about numbers as they are represented in Perl.

perlopentut.pod

A tutorial on using open() effectively.

perlreftut.pod

A tutorial that introduces the essentials of references.

perltootc.pod

A tutorial on managing class data for object modules.

perltodo.pod

Discussion of the most often wanted features that may someday be supported in Perl.

perlunicode.pod

An introduction to Unicode support features in Perl.

91.7 Performance enhancements

91.7.1 Simple sort() using { $ a <=> $ b } and the like are optimized

Many common sort() operations using a simple inlined block are now optimized for faster performance.

91.7.2 Optimized assignments to lexical variables

Certain operations in the RHS of assignment statements have been optimized to directly set the lexical variable on the LHS, eliminating redundant copying overheads.

91.7.3 Faster subroutine calls

Minor changes in how subroutine calls are handled internally provide marginal improvements in performance.

91.7.4 delete(), each(), values() and hash iteration are faster

The hash values returned by delete(), each(), values() and hashes in a list context are the actual values in the hash, instead of copies. This results in significantly better performance, because it eliminates needless copying in most situations.

91.8 Installation and Configuration Improvements

91.8.1 -Dusethreads means something different

The -Dusethreads flag now enables the experimental interpreter-based thread support by default. To get the flavor of experimental threads that was in 5.005 instead, you need to run Configure with "-Dusethreads -Duse5005threads".

As of v5.6.0, interpreter-threads support is still lacking a way to create new threads from Perl (i.e., `use Thread;` will not work with interpreter threads). `use Thread;` continues to be available when you specify the -Duse5005threads option to Configure, bugs and all.

```
    NOTE: Support for threads continues to be an experimental feature.
    Interfaces and implementation are subject to sudden and drastic changes.
```

91.8.2 New Configure flags

The following new flags may be enabled on the Configure command line by running Configure with `-Dflag`.

```
usemultiplicity
usethreads useithreads      (new interpreter threads: no Perl API yet)
usethreads use5005threads   (threads as they were in 5.005)

use64bitint                 (equal to now deprecated 'use64bits')
use64bitall

uselongdouble
usemorebits
uselargefiles
usesocks                    (only SOCKS v5 supported)
```

91.8.3 Threadedness and 64-bitness now more daring

The Configure options enabling the use of threads and the use of 64-bitness are now more daring in the sense that they no more have an explicit list of operating systems of known threads/64-bit capabilities. In other words: if your operating system has the necessary APIs and datatypes, you should be able just to go ahead and use them, for threads by Configure -Dusethreads, and for 64 bits either explicitly by Configure -Duse64bitint or implicitly if your system has 64-bit wide datatypes. See also §91.3.11.

91.8.4 Long Doubles

Some platforms have "long doubles", floating point numbers of even larger range than ordinary "doubles". To enable using long doubles for Perl's scalars, use -Duselongdouble.

91.8.5 -Dusemorebits

You can enable both -Duse64bitint and -Duselongdouble with -Dusemorebits. See also §91.3.11.

91.8.6 -Duselargefiles

Some platforms support system APIs that are capable of handling large files (typically, files larger than two gigabytes). Perl will try to use these APIs if you ask for -Duselargefiles.

See §91.3.12 for more information.

91.8.7 installusrbinperl

You can use "Configure -Uinstallusrbinperl" which causes installperl to skip installing perl also as /usr/bin/perl. This is useful if you prefer not to modify /usr/bin for some reason or another but harmful because many scripts assume to find Perl in /usr/bin/perl.

91.8.8 SOCKS support

You can use "Configure -Dusesocks" which causes Perl to probe for the SOCKS proxy protocol library (v5, not v4). For more information on SOCKS, see:

```
http://www.socks.nec.com/
```

91.8.9 `-A` flag

You can "post-edit" the Configure variables using the Configure `-A` switch. The editing happens immediately after the platform specific hints files have been processed but before the actual configuration process starts. Run `Configure -h` to find out the full `-A` syntax.

91.8.10 Enhanced Installation Directories

The installation structure has been enriched to improve the support for maintaining multiple versions of perl, to provide locations for vendor-supplied modules, scripts, and manpages, and to ease maintenance of locally-added modules, scripts, and manpages. See the section on Installation Directories in the INSTALL file for complete details. For most users building and installing from source, the defaults should be fine.

If you previously used `Configure -Dsitelib` or `-Dsitearch` to set special values for library directories, you might wish to consider using the new `-Dsiteprefix` setting instead. Also, if you wish to re-use a config.sh file from an earlier version of perl, you should be sure to check that Configure makes sensible choices for the new directories. See INSTALL for complete details.

91.8.11 gcc automatically tried if 'cc' does not seem to be working

In many platforms the vendor-supplied 'cc' is too stripped-down to build Perl (basically, the 'cc' doesn't do ANSI C). If this seems to be the case and the 'cc' does not seem to be the GNU C compiler 'gcc', an automatic attempt is made to find and use 'gcc' instead.

91.9 Platform specific changes

91.9.1 Supported platforms

- The Mach CThreads (NEXTSTEP, OPENSTEP) are now supported by the Thread extension.
- GNU/Hurd is now supported.
- Rhapsody/Darwin is now supported.
- EPOC is now supported (on Psion 5).
- The cygwin port (formerly cygwin32) has been greatly improved.

91.9.2 DOS

- Perl now works with djgpp 2.02 (and 2.03 alpha).
- Environment variable names are not converted to uppercase any more.
- Incorrect exit codes from backticks have been fixed.
- This port continues to use its own builtin globbing (not File::Glob).

91.9.3 OS390 (OpenEdition MVS)

Support for this EBCDIC platform has not been renewed in this release. There are difficulties in reconciling Perl's standardization on UTF-8 as its internal representation for characters with the EBCDIC character set, because the two are incompatible.

It is unclear whether future versions will renew support for this platform, but the possibility exists.

91.9.4 VMS

Numerous revisions and extensions to configuration, build, testing, and installation process to accommodate core changes and VMS-specific options.

Expand %ENV-handling code to allow runtime mapping to logical names, CLI symbols, and CRTL environ array.

Extension of subprocess invocation code to accept filespecs as command "verbs".

Add to Perl command line processing the ability to use default file types and to recognize Unix-style `2>&1`.

Expansion of File::Spec::VMS routines, and integration into ExtUtils::MM_VMS.

Extension of ExtUtils::MM_VMS to handle complex extensions more flexibly.

Barewords at start of Unix-syntax paths may be treated as text rather than only as logical names.

Optional secure translation of several logical names used internally by Perl.

Miscellaneous bugfixing and porting of new core code to VMS.

Thanks are gladly extended to the many people who have contributed VMS patches, testing, and ideas.

91.9.5 Win32

Perl can now emulate fork() internally, using multiple interpreters running in different concurrent threads. This support must be enabled at build time. See *perlfork* for detailed information.

When given a pathname that consists only of a drivename, such as `A:`, opendir() and stat() now use the current working directory for the drive rather than the drive root.

The builtin XSUB functions in the Win32:: namespace are documented. See *Win32*.

$^X now contains the full path name of the running executable.

A Win32::GetLongPathName() function is provided to complement Win32::GetFullPathName() and Win32::GetShortPathName(). See *Win32*.

POSIX::uname() is supported.

system(1,...) now returns true process IDs rather than process handles. kill() accepts any real process id, rather than strictly return values from system(1,...).

For better compatibility with Unix, `kill(0, $pid)` can now be used to test whether a process exists.

The `Shell` module is supported.

Better support for building Perl under command.com in Windows 95 has been added.

Scripts are read in binary mode by default to allow ByteLoader (and the filter mechanism in general) to work properly. For compatibility, the DATA filehandle will be set to text mode if a carriage return is detected at the end of the line containing the __END__ or __DATA__ token; if not, the DATA filehandle will be left open in binary mode. Earlier versions always opened the DATA filehandle in text mode.

The glob() operator is implemented via the `File::Glob` extension, which supports glob syntax of the C shell. This increases the flexibility of the glob() operator, but there may be compatibility issues for programs that relied on the older globbing syntax. If you want to preserve compatibility with the older syntax, you might want to run perl with `-MFile::DosGlob`. For details and compatibility information, see *File::Glob*.

91.10 Significant bug fixes

91.10.1 <HANDLE> on empty files

With `$/` set to `undef`, "slurping" an empty file returns a string of zero length (instead of `undef`, as it used to) the first time the HANDLE is read after `$/` is set to `undef`. Further reads yield `undef`.

This means that the following will append "foo" to an empty file (it used to do nothing):

```
perl -0777 -pi -e 's/^/foo/' empty_file
```

The behaviour of:

```
perl -pi -e 's/^/foo/' empty_file
```

is unchanged (it continues to leave the file empty).

91.10.2 `eval '...'` improvements

Line numbers (as reflected by caller() and most diagnostics) within `eval '...'` were often incorrect where here documents were involved. This has been corrected.

Lexical lookups for variables appearing in `eval '...'` within functions that were themselves called within an `eval '...'` were searching the wrong place for lexicals. The lexical search now correctly ends at the subroutine's block boundary.

The use of `return` within `eval {...}` caused $@ not to be reset correctly when no exception occurred within the eval. This has been fixed.

Parsing of here documents used to be flawed when they appeared as the replacement expression in `eval 's/.../.../e'`. This has been fixed.

91.10.3 All compilation errors are true errors

Some "errors" encountered at compile time were by necessity generated as warnings followed by eventual termination of the program. This enabled more such errors to be reported in a single run, rather than causing a hard stop at the first error that was encountered.

The mechanism for reporting such errors has been reimplemented to queue compile-time errors and report them at the end of the compilation as true errors rather than as warnings. This fixes cases where error messages leaked through in the form of warnings when code was compiled at run time using `eval STRING`, and also allows such errors to be reliably trapped using `eval "..."`.

91.10.4 Implicitly closed filehandles are safer

Sometimes implicitly closed filehandles (as when they are localized, and Perl automatically closes them on exiting the scope) could inadvertently set $? or $!. This has been corrected.

91.10.5 Behavior of list slices is more consistent

When taking a slice of a literal list (as opposed to a slice of an array or hash), Perl used to return an empty list if the result happened to be composed of all undef values.

The new behavior is to produce an empty list if (and only if) the original list was empty. Consider the following example:

```
    @a = (1,undef,undef,2)[2,1,2];
```

The old behavior would have resulted in @a having no elements. The new behavior ensures it has three undefined elements.

Note in particular that the behavior of slices of the following cases remains unchanged:

```
    @a = ()[1,2];
    @a = (getpwent)[7,0];
    @a = (anything_returning_empty_list())[2,1,2];
    @a = @b[2,1,2];
    @a = @c{'a','b','c'};
```

See *perldata*.

91.10.6 `(\$)` prototype and `$foo{a}`

A scalar reference prototype now correctly allows a hash or array element in that slot.

91.10.7 `goto &sub` and AUTOLOAD

The `goto &sub` construct works correctly when `&sub` happens to be autoloaded.

91.10.8 `-bareword` allowed under `use integer`

The autoquoting of barewords preceded by `-` did not work in prior versions when the `integer` pragma was enabled. This has been fixed.

91.10.9 Failures in DESTROY()

When code in a destructor threw an exception, it went unnoticed in earlier versions of Perl, unless someone happened to be looking in $@ just after the point the destructor happened to run. Such failures are now visible as warnings when warnings are enabled.

91.10.10 Locale bugs fixed

printf() and sprintf() previously reset the numeric locale back to the default "C" locale. This has been fixed.

Numbers formatted according to the local numeric locale (such as using a decimal comma instead of a decimal dot) caused "isn't numeric" warnings, even while the operations accessing those numbers produced correct results. These warnings have been discontinued.

91.10.11 Memory leaks

The `eval 'return sub {...}'` construct could sometimes leak memory. This has been fixed.

Operations that aren't filehandle constructors used to leak memory when used on invalid filehandles. This has been fixed.

Constructs that modified `@_` could fail to deallocate values in `@_` and thus leak memory. This has been corrected.

91.10.12 Spurious subroutine stubs after failed subroutine calls

Perl could sometimes create empty subroutine stubs when a subroutine was not found in the package. Such cases stopped later method lookups from progressing into base packages. This has been corrected.

91.10.13 Taint failures under `-U`

When running in unsafe mode, taint violations could sometimes cause silent failures. This has been fixed.

91.10.14 END blocks and the `-c` switch

Prior versions used to run BEGIN **and** END blocks when Perl was run in compile-only mode. Since this is typically not the expected behavior, END blocks are not executed anymore when the `-c` switch is used, or if compilation fails.

See §91.3.18 for how to run things when the compile phase ends.

91.10.15 Potential to leak DATA filehandles

Using the `__DATA__` token creates an implicit filehandle to the file that contains the token. It is the program's responsibility to close it when it is done reading from it.

This caveat is now better explained in the documentation. See *perldata*.

91.11 New or Changed Diagnostics

"%s" variable %s masks earlier declaration in same %s

(W misc) A "my" or "our" 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 referents to it are destroyed.

"my sub" not yet implemented

(F) Lexically scoped subroutines are not yet implemented. Don't try that yet.

"our" variable %s redeclared

(W misc) You seem to have already declared the same global once before in the current lexical scope.

'!' allowed only after types %s

(F) The '!' is allowed in pack() and unpack() only after certain types. See pack in *perlfunc*.

/ cannot take a count

(F) You had an unpack template indicating a counted-length string, but you have also specified an explicit size for the string. See pack in *perlfunc*.

/ must be followed by a, A or Z

(F) You had an unpack template indicating a counted-length string, which must be followed by one of the letters a, A or Z to indicate what sort of string is to be unpacked. See pack in *perlfunc*.

/ must be followed by a*, A* or Z*

(F) You had a pack template indicating a counted-length string, Currently the only things that can have their length counted are a*, A* or Z*. See pack in *perlfunc*.

/ must follow a numeric type

(F) You had an unpack template that contained a '#', but this did not follow some numeric unpack specification. See pack in *perlfunc*.

/%s/: Unrecognized escape \\%c passed through

(W regexp) You used a backslash-character combination which is not recognized by Perl. This combination appears in an interpolated variable or a `'`-delimited regular expression. The character was understood literally.

/%s/: Unrecognized escape \\%c in character class passed through

(W regexp) You used a backslash-character combination which is not recognized by Perl inside character classes. The character was understood literally.

/%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.

%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 *perlsub*.

%s argument is not a HASH or ARRAY element

(F) The argument to exists() must be a hash or array element, such as:

```
    $foo{$bar}
    $ref->{"susie"}[12]
```

%s 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"}
```

%s 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.

%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*.

(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 G_KEEPERR in *perlcall*.

<> should be quotes

(F) You wrote `require <file>` when you should have written `require 'file'`.

Attempt to join self

(F) You tried to join a thread from within itself, which is an impossible task. You may be joining the wrong thread, or you may need to move the join() to some other thread.

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 realloc() ignored

(S) An internal routine called realloc() on something that had never been malloc()ed in the first place. Mandatory, but can be disabled by setting environment variable `PERL_BADFREE` to 1.

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.

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 *perlport* for more on portability concerns.

Bit vector size > 32 non-portable

(W portable) Using bit vector sizes larger than 32 is non-portable.

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.

Can't check filesystem of script "%s"

(P) For some reason you can't check the filesystem of the script for nosuid.

Can't declare class for non-scalar %s in "%s"

(S) Currently, only scalar variables can declared with a specific class qualifier in a "my" or "our" declaration. The semantics may be extended for other types of variables in future.

Can't declare %s in "%s"

(F) Only scalar, array, and hash variables may be declared as "my" or "our" variables. They must have ordinary identifiers as names.

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 modify non-lvalue subroutine call

(F) Subroutines meant to be used in lvalue context should be declared as such, see Lvalue subroutines in *perlsub*.

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 *perlvms*) so that environ is not searched.

Can't remove %s: %s, skipping file

(S) 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 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 weaken a nonreference

(F) You attempted to weaken something that was not a reference. Only references can be weakened.

Character class [:%s: unknown]

(F) The class in the character class [: :] syntax is unknown. See *perlre*.

Character class syntax [%s belongs inside] **character classes**

(W unsafe) 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.

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 Constant Functions in *perlsub* and *constant*.

constant(%s): %s

(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` or `charnames` pragma? See *charnames* and *overload*.

CORE::%s is not a keyword

(F) The CORE:: namespace is reserved for Perl keywords.

defined(@array) is deprecated

(D) defined() is not usually 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.

defined(%hash) is deprecated

(D) defined() is not usually useful on hashes because it checks for an undefined *scalar* value. If you want to see if the hash is empty, just use `if (%hash) { # not empty }` for example.

Did not produce a valid header

See Server error.

(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.

Document contains no data

See Server error.

entering effective %s failed

(F) While under the `use filetest` pragma, switching the real and effective uids or gids failed.

false [range "%s" in regexp]

(W regexp) 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 "-". Consider quoting the "-", "\-". See *perlre*.

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 "<" or nothing. If you intended only to read from the file, use "<". See open in *perlfunc*.

flock() on closed filehandle %s

(W closed) The filehandle you're attempting to flock() got itself closed some time before now. Check your logic flow. flock() operates on filehandles. Are you attempting to call flock() on a dirhandle by the same name?

Global symbol "%s" requires explicit package name

(F) You've said "use strict vars", which indicates that all variables must either be lexically scoped (using "my"), declared beforehand using "our", or explicitly qualified to say which package the global variable is in (using "::").

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 *perlport* for more on portability concerns.

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.

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 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).

Integer overflow in %s number

(W 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.

Invalid %s attribute: %s

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

The indicated attributes for a subroutine or variable were not recognized by Perl or by a user-supplied handler. See *attributes*.

invalid [range "%s" in regexp]

The offending range is now explicitly displayed.

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 subroutine attribute list

(F) Something other than a colon or whitespace was seen between the elements of a subroutine attribute list. If the previous attribute had a parenthesised parameter list, perhaps that list was terminated too soon.

leaving effective %s failed

(F) While under the `use filetest` pragma, switching the real and effective uids or gids failed.

Lvalue subs returning %s not implemented yet

(F) Due to limitations in the current implementation, array and hash values cannot be returned in subroutines used in lvalue context. See Lvalue subroutines in *perlsub*.

Method %s not permitted

See Server error.

Missing %sbrace%s on \N{}

(F) Wrong syntax of character name literal `\N{charname}` within double-quotish context.

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 name in "my sub"

(F) The reserved syntax for lexically scoped subroutines requires that they have a name with which they can be found.

No %s specified for -%c

(F) The indicated command line switch needs a mandatory argument, but you haven't specified one.

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 semantics. Such syntax is reserved for future extensions.

No space allowed after -%c

(F) The argument to the indicated command line switch must follow immediately after the switch, without intervening spaces.

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.

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 *perlport* for more on portability concerns.

See also *perlport* for writing portable code.

panic: del_backref

(P) Failed an internal consistency check while trying to reset a weak reference.

panic: kid popen errno read

(F) forked child returned an incomprehensible message about its errno.

panic: magic_killbackrefs

(P) Failed an internal consistency check while trying to reset all weak references to an object.

Parentheses missing around "%s" list

(W parenthesis) You said something like

```
    my $foo, $bar = @_;
```

when you meant

```
    my ($foo, $bar) = @_;
```

Remember that "my", "our", and "local" bind tighter than comma.

Possible unintended interpolation of %s in string

(W ambiguous) It used to be that Perl would try to guess whether you wanted an array interpolated or a literal @. It no longer does this; arrays are now *always* interpolated into strings. This means that if you try something like:

```
        print "fred@example.com";
```

and the array `@example` doesn't exist, Perl is going to print `fred.com`, which is probably not what you wanted. To get a literal `@` sign in a string, put a backslash before it, just as you would to get a literal `$` sign.

Possible Y2K bug: %s

(W y2k) You are concatenating the number 19 with another number, which could be a potential Year 2000 problem.

pragma "attrs" is deprecated, use "sub NAME : ATTRS" instead

(W deprecated) You have written something like this:

```
    sub doit
    {
        use attrs qw(locked);
    }
```

You should use the new declaration syntax instead.

```
    sub doit : locked
    {
        ...
```

The `use attrs` pragma is now obsolete, and is only provided for backward-compatibility. See Subroutine Attributes in *perlsub*.

Premature end of script headers

See Server error.

Repeat count in pack overflows

(F) You can't specify a repeat count so large that it overflows your signed integers. See pack in *perlfunc*.

Repeat count in unpack overflows

(F) You can't specify a repeat count so large that it overflows your signed integers. See unpack in *perlfunc*.

realloc() of freed memory ignored

(S) An internal routine called realloc() on something that had already been freed.

Reference is already weak

(W misc) You have attempted to weaken a reference that is already weak. Doing so has no effect.

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.

Strange *+?{} on zero-length expression

(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}/`.

switching effective %s is not implemented

(F) While under the `use filetest` pragma, we cannot switch the real and effective uids or gids.

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 *perlvms*) so that the environ array isn't the target of the change to %ENV which produced the warning.

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.

Unknown open() mode '%s'

(F) The second argument of 3-argument open() is not among the list of valid modes: <, >, >>, +<, +>, +>>, -|, |-.

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.

Unrecognized escape \\%c passed through

(W misc) You used a backslash-character combination which is not recognized by Perl. The character was understood literally.

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 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 subroutine attribute list

(F) The lexer saw an opening (left) parenthesis character while parsing a subroutine 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.

Unterminated subroutine attribute list

(F) The lexer found something other than a simple identifier at the start of a subroutine 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.

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.

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.

91.12 New tests

lib/attrs

Compatibility tests for `sub : attrs` vs the older `use attrs`.

lib/env

Tests for new environment scalar capability (e.g., `use Env qw($BAR);`).

lib/env-array

Tests for new environment array capability (e.g., `use Env qw(@PATH);`).

lib/io_const

IO constants (SEEK_*, _IO*).

lib/io_dir

Directory-related IO methods (new, read, close, rewind, tied delete).

lib/io_multihomed

INET sockets with multi-homed hosts.

lib/io_poll

IO poll().

lib/io_unix

UNIX sockets.

op/attrs

Regression tests for `my ($x,@y,%z) : attrs` and <sub : attrs>.

op/filetest

File test operators.

op/lex_assign

Verify operations that access pad objects (lexicals and temporaries).

op/exists_sub

Verify `exists &sub` operations.

91.13 Incompatible Changes

91.13.1 Perl Source Incompatibilities

Beware that any new warnings that have been added or old ones that have been enhanced are **not** considered incompatible changes.

Since all new warnings must be explicitly requested via the `-w` switch or the `warnings` pragma, it is ultimately the programmer's responsibility to ensure that warnings are enabled judiciously.

CHECK is a new keyword

All subroutine definitions named CHECK are now special. See `/"Support for CHECK blocks"` for more information.

Treatment of list slices of undef has changed

There is a potential incompatibility in the behavior of list slices that are comprised entirely of undefined values. See §91.10.5.

Format of $ English::PERL_VERSION is different

The English module now sets $PERL_VERSION to $ˆV (a string value) rather than `$]` (a numeric value). This is a potential incompatibility. Send us a report via perlbug if you are affected by this.

See §91.3.7 for the reasons for this change.

Literals of the form `1.2.3` parse differently

Previously, numeric literals with more than one dot in them were interpreted as a floating point number concatenated with one or more numbers. Such "numbers" are now parsed as strings composed of the specified ordinals.

For example, `print 97.98.99` used to output `97.9899` in earlier versions, but now prints `abc`.

See §91.3.6.

Possibly changed pseudo-random number generator

Perl programs that depend on reproducing a specific set of pseudo-random numbers may now produce different output due to improvements made to the rand() builtin. You can use `sh Configure -Drandfunc=rand` to obtain the old behavior.

See §91.3.20.

Hashing function for hash keys has changed

Even though Perl hashes are not order preserving, the apparently random order encountered when iterating on the contents of a hash is actually determined by the hashing algorithm used. Improvements in the algorithm may yield a random order that is **different** from that of previous versions, especially when iterating on hashes.

See §91.3.22 for additional information.

`undef` fails on read only values

Using the `undef` operator on a readonly value (such as $1) has the same effect as assigning `undef` to the readonly value–it throws an exception.

Close-on-exec bit may be set on pipe and socket handles

Pipe and socket handles are also now subject to the close-on-exec behavior determined by the special variable $ˆF.

See §91.3.44.

Writing `"$$1"` to mean `"${$}1"` is unsupported

Perl 5.004 deprecated the interpretation of `$$1` and similar within interpolated strings to mean `$$ . "1"`, but still allowed it.

In Perl 5.6.0 and later, `"$$1"` always means `"${$1}"`.

delete(), each(), values() and `\(%h)`

operate on aliases to values, not copies

delete(), each(), values() and hashes (e.g. `\(%h)`) in a list context return the actual values in the hash, instead of copies (as they used to in earlier versions). Typical idioms for using these constructs copy the returned values, but this can make a significant difference when creating references to the returned values. Keys in the hash are still returned as copies when iterating on a hash.

See also §91.7.4.

vec(EXPR,OFFSET,BITS) enforces powers-of-two BITS

vec() generates a run-time error if the BITS argument is not a valid power-of-two integer.

Text of some diagnostic output has changed

Most references to internal Perl operations in diagnostics have been changed to be more descriptive. This may be an issue for programs that may incorrectly rely on the exact text of diagnostics for proper functioning.

`%@` has been removed

The undocumented special variable `%@` that used to accumulate "background" errors (such as those that happen in DESTROY()) has been removed, because it could potentially result in memory leaks.

Parenthesized not() behaves like a list operator

The `not` operator now falls under the "if it looks like a function, it behaves like a function" rule.

As a result, the parenthesized form can be used with `grep` and `map`. The following construct used to be a syntax error before, but it works as expected now:

```
    grep not($_), @things;
```

On the other hand, using `not` with a literal list slice may not work. The following previously allowed construct:

```
    print not (1,2,3)[0];
```

needs to be written with additional parentheses now:

```
    print not((1,2,3)[0]);
```

The behavior remains unaffected when `not` is not followed by parentheses.

Semantics of bareword prototype `(*)` have changed

The semantics of the bareword prototype * have changed. Perl 5.005 always coerced simple scalar arguments to a typeglob, which wasn't useful in situations where the subroutine must distinguish between a simple scalar and a typeglob. The new behavior is to not coerce bareword arguments to a typeglob. The value will always be visible as either a simple scalar or as a reference to a typeglob.

See §91.3.49.

Semantics of bit operators may have changed on 64-bit platforms

If your platform is either natively 64-bit or if Perl has been configured to used 64-bit integers, i.e., $Config{ivsize} is 8, there may be a potential incompatibility in the behavior of bitwise numeric operators (& | ^ ˜ << >>). These operators used to strictly operate on the lower 32 bits of integers in previous versions, but now operate over the entire native integral width. In particular, note that unary ˜ will produce different results on platforms that have different $Config{ivsize}. For portability, be sure to mask off the excess bits in the result of unary ˜, e.g., `~$x & 0xffffffff`.

See §91.3.47.

More builtins taint their results

As described in §91.3.48, there may be more sources of taint in a Perl program.

To avoid these new tainting behaviors, you can build Perl with the Configure option `-Accflags=-DINCOMPLETE_TAINTS`. Beware that the ensuing perl binary may be insecure.

91.13.2 C Source Incompatibilities

`PERL_POLLUTE`

Release 5.005 grandfathered old global symbol names by providing preprocessor macros for extension source compatibility. As of release 5.6.0, these preprocessor definitions are not available by default. You need to explicitly compile perl with `-DPERL_POLLUTE` to get these definitions. For extensions still using the old symbols, this option can be specified via MakeMaker:

```
    perl Makefile.PL POLLUTE=1
```

`PERL_IMPLICIT_CONTEXT`

This new build option provides a set of macros for all API functions such that an implicit interpreter/thread context argument is passed to every API function. As a result of this, something like `sv_setsv(foo,bar)` amounts to a macro invocation that actually translates to something like `Perl_sv_setsv(my_perl,foo,bar)`. While this is generally expected to not have any significant source compatibility issues, the difference between a macro and a real function call will need to be considered.

This means that there **is** a source compatibility issue as a result of this if your extensions attempt to use pointers to any of the Perl API functions.

Note that the above issue is not relevant to the default build of Perl, whose interfaces continue to match those of prior versions (but subject to the other options described here).

See The Perl API in *perlguts* for detailed information on the ramifications of building Perl with this option.

```
        NOTE: PERL_IMPLICIT_CONTEXT is automatically enabled whenever Perl is built
        with one of -Dusethreads, -Dusemultiplicity, or both.  It is not
        intended to be enabled by users at this time.
```

PERL_POLLUTE_MALLOC

Enabling Perl's malloc in release 5.005 and earlier caused the namespace of the system's malloc family of functions to be usurped by the Perl versions, since by default they used the same names. Besides causing problems on platforms that do not allow these functions to be cleanly replaced, this also meant that the system versions could not be called in programs that used Perl's malloc. Previous versions of Perl have allowed this behaviour to be suppressed with the HIDEMYMALLOC and EMBEDMYMALLOC preprocessor definitions.

As of release 5.6.0, Perl's malloc family of functions have default names distinct from the system versions. You need to explicitly compile perl with `-DPERL_POLLUTE_MALLOC` to get the older behaviour. HIDEMYMALLOC and EMBEDMYMALLOC have no effect, since the behaviour they enabled is now the default.

Note that these functions do **not** constitute Perl's memory allocation API. See Memory Allocation in *perlguts* for further information about that.

91.13.3 Compatible C Source API Changes

PATCHLEVEL is now PERL_VERSION

The cpp macros `PERL_REVISION`, `PERL_VERSION`, and `PERL_SUBVERSION` are now available by default from perl.h, and reflect the base revision, patchlevel, and subversion respectively. `PERL_REVISION` had no prior equivalent, while `PERL_VERSION` and `PERL_SUBVERSION` were previously available as `PATCHLEVEL` and `SUBVERSION`.

The new names cause less pollution of the **cpp** namespace and reflect what the numbers have come to stand for in common practice. For compatibility, the old names are still supported when *patchlevel.h* is explicitly included (as required before), so there is no source incompatibility from the change.

91.13.4 Binary Incompatibilities

In general, the default build of this release is expected to be binary compatible for extensions built with the 5.005 release or its maintenance versions. However, specific platforms may have broken binary compatibility due to changes in the defaults used in hints files. Therefore, please be sure to always check the platform-specific README files for any notes to the contrary.

The usethreads or usemultiplicity builds are **not** binary compatible with the corresponding builds in 5.005.

On platforms that require an explicit list of exports (AIX, OS/2 and Windows, among others), purely internal symbols such as parser functions and the run time opcodes are not exported by default. Perl 5.005 used to export all functions irrespective of whether they were considered part of the public API or not.

For the full list of public API functions, see *perlapi*.

91.14 Known Problems

91.14.1 Localizing a tied hash element may leak memory

As of the 5.6.1 release, there is a known leak when code such as this is executed:

```
    use Tie::Hash;
    tie my %tie_hash => 'Tie::StdHash';

    ...

    local($tie_hash{Foo}) = 1; # leaks
```

91.14.2 Known test failures

- 64-bit builds

  Subtest #15 of lib/b.t may fail under 64-bit builds on platforms such as HP-UX PA64 and Linux IA64. The issue is still being investigated.

  The lib/io_multihomed test may hang in HP-UX if Perl has been configured to be 64-bit. Because other 64-bit platforms do not hang in this test, HP-UX is suspect. All other tests pass in 64-bit HP-UX. The test attempts to create and connect to "multihomed" sockets (sockets which have multiple IP addresses).

  Note that 64-bit support is still experimental.

- Failure of Thread tests

  The subtests 19 and 20 of lib/thr5005.t test are known to fail due to fundamental problems in the 5.005 threading implementation. These are not new failures–Perl 5.005_0x has the same bugs, but didn't have these tests. (Note that support for 5.005-style threading remains experimental.)

- NEXTSTEP 3.3 POSIX test failure

  In NEXTSTEP 3.3p2 the implementation of the strftime(3) in the operating system libraries is buggy: the %j format numbers the days of a month starting from zero, which, while being logical to programmers, will cause the subtests 19 to 27 of the lib/posix test may fail.

- Tru64 (aka Digital UNIX, aka DEC OSF/1) lib/sdbm test failure with gcc

  If compiled with gcc 2.95 the lib/sdbm test will fail (dump core). The cure is to use the vendor cc, it comes with the operating system and produces good code.

91.14.3 EBCDIC platforms not fully supported

In earlier releases of Perl, EBCDIC environments like OS390 (also known as Open Edition MVS) and VM-ESA were supported. Due to changes required by the UTF-8 (Unicode) support, the EBCDIC platforms are not supported in Perl 5.6.0.

The 5.6.1 release improves support for EBCDIC platforms, but they are not fully supported yet.

91.14.4 UNICOS/mk CC failures during Configure run

In UNICOS/mk the following errors may appear during the Configure run:

```
        Guessing which symbols your C compiler and preprocessor define...
        CC-20 cc: ERROR File = try.c, Line = 3
        ...
          bad switch yylook 79bad switch yylook 79bad switch yylook 79bad switch yylook 79#ifdef A29K
        ...
        4 errors detected in the compilation of "try.c".
```

The culprit is the broken awk of UNICOS/mk. The effect is fortunately rather mild: Perl itself is not adversely affected by the error, only the h2ph utility coming with Perl, and that is rather rarely needed these days.

91.14.5 Arrow operator and arrays

When the left argument to the arrow operator -> is an array, or the `scalar` operator operating on an array, the result of the operation must be considered erroneous. For example:

```
    @x->[2]
    scalar(@x)->[2]
```

These expressions will get run-time errors in some future release of Perl.

91.14.6 Experimental features

As discussed above, many features are still experimental. Interfaces and implementation of these features are subject to change, and in extreme cases, even subject to removal in some future release of Perl. These features include the following:

Threads

Unicode

64-bit support

Lvalue subroutines

Weak references

The pseudo-hash data type

The Compiler suite

Internal implementation of file globbing

The DB module

The regular expression code constructs:

`(?{ code })` and `(??{ code })`

91.15 Obsolete Diagnostics

Character class syntax [: : is reserved] **for future extensions**

(W) 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: "\[:" and ":\]".

Ill-formed logical name |%s| in prime_env_iter

(W) A warning peculiar to VMS. A logical name was encountered when preparing to iterate over %ENV which violates the syntactic rules governing logical names. Because it cannot be translated normally, it is skipped, and will not appear in %ENV. This may be a benign occurrence, as some software packages might directly modify logical name tables and introduce nonstandard names, or it may indicate that a logical name table has been corrupted.

In string, @%s now must be written as \@%s

The description of this error used to say:

```
        (Someday it will simply assume that an unbackslashed @
         interpolates an array.)
```

That day has come, and this fatal error has been removed. It has been replaced by a non-fatal warning instead. See Arrays now always interpolate into double-quoted strings for details.

Probable precedence problem on %s

(W) 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;
```

regexp too big

(F) The current implementation of regular expressions uses shorts as address offsets within a string. Unfortunately this means that if the regular expression compiles to longer than 32767, it'll blow up. Usually when you want a regular expression this big, there is a better way to do it with multiple statements. See *perlre*.

Use of "$ $ <digit>" to mean "$ {$ }<digit>" is deprecated

(D) Perl versions before 5.004 misinterpreted any type marker followed by "$" and a digit. For example, "$$0" was incorrectly taken to mean "${$}0" instead of "${$0}". This bug is (mostly) fixed in Perl 5.004.

However, the developers of Perl 5.004 could not fix this bug completely, because at least two widely-used modules depend on the old meaning of "$$0" in a string. So Perl 5.004 still interprets "$$<digit>" in the old (broken) way inside strings; but it generates this message as a warning. And in Perl 5.005, this special treatment will cease.

91.16 Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup. There may also be information at http://www.perl.com/ , the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to perlbug@perl.org to be analysed by the Perl porting team.

91.17 SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

91.18 HISTORY

Written by Gurusamy Sarathy <*gsar@ActiveState.com*>, with many contributions from The Perl Porters.

Send omissions or corrections to <*perlbug@perl.org*>.

Chapter 92

perl56delta

What's new for perl v5.6.0

92.1 DESCRIPTION

This document describes differences between the 5.005 release and the 5.6.0 release.

92.2 Core Enhancements

92.2.1 Interpreter cloning, threads, and concurrency

Perl 5.6.0 introduces the beginnings of support for running multiple interpreters concurrently in different threads. In conjunction with the perl_clone() API call, which can be used to selectively duplicate the state of any given interpreter, it is possible to compile a piece of code once in an interpreter, clone that interpreter one or more times, and run all the resulting interpreters in distinct threads.

On the Windows platform, this feature is used to emulate fork() at the interpreter level. See *perlfork* for details about that.

This feature is still in evolution. It is eventually meant to be used to selectively clone a subroutine and data reachable from that subroutine in a separate interpreter and run the cloned subroutine in a separate thread. Since there is no shared data between the interpreters, little or no locking will be needed (unless parts of the symbol table are explicitly shared). This is obviously intended to be an easy-to-use replacement for the existing threads support.

Support for cloning interpreters and interpreter concurrency can be enabled using the -Dusethreads Configure option (see win32/Makefile for how to enable it on Windows.) The resulting perl executable will be functionally identical to one that was built with -Dmultiplicity, but the perl_clone() API call will only be available in the former.

-Dusethreads enables the cpp macro USE_ITHREADS by default, which in turn enables Perl source code changes that provide a clear separation between the op tree and the data it operates with. The former is immutable, and can therefore be shared between an interpreter and all of its clones, while the latter is considered local to each interpreter, and is therefore copied for each clone.

Note that building Perl with the -Dusemultiplicity Configure option is adequate if you wish to run multiple **independent** interpreters concurrently in different threads. -Dusethreads only provides the additional functionality of the perl_clone() API call and other support for running **cloned** interpreters concurrently.

```
    NOTE: This is an experimental feature.  Implementation details are
    subject to change.
```

92.2.2 Lexically scoped warning categories

You can now control the granularity of warnings emitted by perl at a finer level using the `use warnings` pragma. *warnings* and *perllexwarn* have copious documentation on this feature.

92.2.3 Unicode and UTF-8 support

Perl now uses UTF-8 as its internal representation for character strings. The `utf8` and `bytes` pragmas are used to control this support in the current lexical scope. See *perlunicode*, *utf8* and *bytes* for more information.

This feature is expected to evolve quickly to support some form of I/O disciplines that can be used to specify the kind of input and output data (bytes or characters). Until that happens, additional modules from CPAN will be needed to complete the toolkit for dealing with Unicode.

```
    NOTE: This should be considered an experimental feature.  Implementation
    details are subject to change.
```

92.2.4 Support for interpolating named characters

The new `\N` escape interpolates named characters within strings. For example, `"Hi!  \N{WHITE SMILING FACE}"` evaluates to a string with a unicode smiley face at the end.

92.2.5 "our" declarations

An "our" declaration introduces a value that can be best understood as a lexically scoped symbolic alias to a global variable in the package that was current where the variable was declared. This is mostly useful as an alternative to the `vars` pragma, but also provides the opportunity to introduce typing and other attributes for such variables. See our in *perlfunc*.

92.2.6 Support for strings represented as a vector of ordinals

Literals of the form `v1.2.3.4` are now parsed as a string composed of characters with the specified ordinals. This is an alternative, more readable way to construct (possibly unicode) strings instead of interpolating characters, as in `"\x{1}\x{2}\x{3}\x{4}"`. The leading `v` may be omitted if there are more than two ordinals, so `1.2.3` is parsed the same as `v1.2.3`.

Strings written in this form are also useful to represent version "numbers". It is easy to compare such version "numbers" (which are really just plain strings) using any of the usual string comparison operators `eq`, `ne`, `lt`, `gt`, etc., or perform bitwise string operations on them using |, &, etc.

In conjunction with the new `$^V` magic variable (which contains the perl version as a string), such literals can be used as a readable way to check if you're running a particular version of Perl:

```
    # this will parse in older versions of Perl also
    if ($^V and $^V gt v5.6.0) {
        # new features supported
    }
```

`require` and `use` also have some special magic to support such literals, but this particular usage should be avoided because it leads to misleading error messages under versions of Perl which don't support vector strings. Using a true version number will ensure correct behavior in all versions of Perl:

```
    require 5.006;    # run time check for v5.6
    use 5.006_001;    # compile time check for v5.6.1
```

Also, `sprintf` and `printf` support the Perl-specific format flag `%v` to print ordinals of characters in arbitrary strings:

```
    printf "v%vd", $^V;         # prints current version, such as "v5.5.650"
    printf "%*vX", ":", $addr;  # formats IPv6 address
    printf "%*vb", " ", $bits;  # displays bitstring
```

See Scalar value constructors in *perldata* for additional information.

92.2.7 Improved Perl version numbering system

Beginning with Perl version 5.6.0, the version number convention has been changed to a "dotted integer" scheme that is more commonly found in open source projects.

Maintenance versions of v5.6.0 will be released as v5.6.1, v5.6.2 etc. The next development series following v5.6.0 will be numbered v5.7.x, beginning with v5.7.0, and the next major production release following v5.6.0 will be v5.8.0.

The English module now sets $PERL_VERSION to $^V (a string value) rather than `$]` (a numeric value). (This is a potential incompatibility. Send us a report via perlbug if you are affected by this.)

The v1.2.3 syntax is also now legal in Perl. See Support for strings represented as a vector of ordinals for more on that.

To cope with the new versioning system's use of at least three significant digits for each version component, the method used for incrementing the subversion number has also changed slightly. We assume that versions older than v5.6.0 have been incrementing the subversion component in multiples of 10. Versions after v5.6.0 will increment them by 1. Thus, using the new notation, 5.005_03 is the "same" as v5.5.30, and the first maintenance version following v5.6.0 will be v5.6.1 (which should be read as being equivalent to a floating point value of 5.006_001 in the older format, stored in `$]`).

92.2.8 New syntax for declaring subroutine attributes

Formerly, if you wanted to mark a subroutine as being a method call or as requiring an automatic lock() when it is entered, you had to declare that with a `use attrs` pragma in the body of the subroutine. That can now be accomplished with declaration syntax, like this:

```
sub mymethod : locked method ;
...
sub mymethod : locked method {
    ...
}

sub othermethod :locked :method ;
...
sub othermethod :locked :method {
    ...
}
```

(Note how only the first `:` is mandatory, and whitespace surrounding the `:` is optional.)

AutoSplit.pm and *SelfLoader.pm* have been updated to keep the attributes with the stubs they provide. See *attributes*.

92.2.9 File and directory handles can be autovivified

Similar to how constructs such as `$x->[0]` autovivify a reference, handle constructors (open(), opendir(), pipe(), socketpair(), sysopen(), socket(), and accept()) now autovivify a file or directory handle 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
}
```

92.2.10 open() with more than two arguments

If open() is passed three arguments instead of two, the second argument is used as the mode and the third argument is taken to be the file name. This is primarily useful for protecting against unintended magic behavior of the traditional two-argument form. See open in *perlfunc*.

92.2.11 64-bit support

Any platform that has 64-bit integers either

```
        (1) natively as longs or ints
        (2) via special compiler flags
        (3) using long long or int64_t
```

is able to use "quads" (64-bit integers) as follows:

- constants (decimal, hexadecimal, octal, binary) in the code
- arguments to oct() and hex()
- arguments to print(), printf() and sprintf() (flag prefixes ll, L, q)
- printed as such
- pack() and unpack() "q" and "Q" formats
- in basic arithmetics: + - * / % (NOTE: operating close to the limits of the integer values may produce surprising results)
- in bit arithmetics: & | ˆ ˜ << >> (NOTE: these used to be forced to be 32 bits wide but now operate on the full native width.)
- vec()

Note that unless you have the case (a) you will have to configure and compile Perl using the -Duse64bitint Configure flag.

```
    NOTE: The Configure flags -Duselonglong and -Duse64bits have been
    deprecated.  Use -Duse64bitint instead.
```

There are actually two modes of 64-bitness: the first one is achieved using Configure -Duse64bitint and the second one using Configure -Duse64bitall. The difference is that the first one is minimal and the second one maximal. The first works in more places than the second.

The `use64bitint` does only as much as is required to get 64-bit integers into Perl (this may mean, for example, using "long longs") while your memory may still be limited to 2 gigabytes (because your pointers could still be 32-bit). Note that the name `64bitint` does not imply that your C compiler will be using 64-bit `int`s (it might, but it doesn't have to): the `use64bitint` means that you will be able to have 64 bits wide scalar values.

The `use64bitall` goes all the way by attempting to switch also integers (if it can), longs (and pointers) to being 64-bit. This may create an even more binary incompatible Perl than -Duse64bitint: the resulting executable may not run at all in a 32-bit box, or you may have to reboot/reconfigure/rebuild your operating system to be 64-bit aware.

Natively 64-bit systems like Alpha and Cray need neither -Duse64bitint nor -Duse64bitall.

Last but not least: note that due to Perl's habit of always using floating point numbers, the quads are still not true integers. When quads overflow their limits (0...18_446_744_073_709_551_615 unsigned, -9_223_372_036_854_775_808...9_223_372_036_854_775_807 signed), they are silently promoted to floating point numbers, after which they will start losing precision (in their lower digits).

```
    NOTE: 64-bit support is still experimental on most platforms.
    Existing support only covers the LP64 data model.  In particular, the
    LLP64 data model is not yet supported.  64-bit libraries and system
    APIs on many platforms have not stabilized--your mileage may vary.
```

92.2.12 Large file support

If you have filesystems that support "large files" (files larger than 2 gigabytes), you may now also be able to create and access them from Perl.

```
    NOTE: The default action is to enable large file support, if
    available on the platform.
```

If the large file support is on, and you have a Fcntl constant O_LARGEFILE, the O_LARGEFILE is automatically added to the flags of sysopen().

Beware that unless your filesystem also supports "sparse files" seeking to umpteen petabytes may be inadvisable.

Note that in addition to requiring a proper file system to do large files you may also need to adjust your per-process (or your per-system, or per-process-group, or per-user-group) maximum filesize limits before running Perl scripts that try to handle large files, especially if you intend to write such files.

Finally, in addition to your process/process group maximum filesize limits, you may have quota limits on your filesystems that stop you (your user id or your user group id) from using large files.

Adjusting your process/user/group/file system/operating system limits is outside the scope of Perl core language. For process limits, you may try increasing the limits using your shell's limits/limit/ulimit command before running Perl. The BSD::Resource extension (not included with the standard Perl distribution) may also be of use, it offers the getrlimit/setrlimit interface that can be used to adjust process resource usage limits, including the maximum filesize limit.

92.2.13 Long doubles

In some systems you may be able to use long doubles to enhance the range and precision of your double precision floating point numbers (that is, Perl's numbers). Use Configure -Duselongdouble to enable this support (if it is available).

92.2.14 "more bits"

You can "Configure -Dusemorebits" to turn on both the 64-bit support and the long double support.

92.2.15 Enhanced support for sort() subroutines

Perl subroutines with a prototype of `($$)`, and XSUBs in general, can now be used as sort subroutines. In either case, the two elements to be compared are passed as normal parameters in @_. See sort in *perlfunc*.

For unprototyped sort subroutines, the historical behavior of passing the elements to be compared as the global variables $a and $b remains unchanged.

92.2.16 `sort $coderef @foo` allowed

sort() did not accept a subroutine reference as the comparison function in earlier versions. This is now permitted.

92.2.17 File globbing implemented internally

Perl now uses the File::Glob implementation of the glob() operator automatically. This avoids using an external csh process and the problems associated with it.

```
    NOTE: This is currently an experimental feature.  Interfaces and
    implementation are subject to change.
```

92.2.18 Support for CHECK blocks

In addition to `BEGIN`, `INIT`, `END`, `DESTROY` and `AUTOLOAD`, subroutines named `CHECK` are now special. These are queued up during compilation and behave similar to END blocks, except they are called at the end of compilation rather than at the end of execution. They cannot be called directly.

92.2.19 POSIX character class syntax [: :] supported

For example to match alphabetic characters use /[[:alpha:]]/. See *perlre* for details.

92.2.20 Better pseudo-random number generator

In 5.005_0x and earlier, perl's rand() function used the C library rand(3) function. As of 5.005_52, Configure tests for drand48(), random(), and rand() (in that order) and picks the first one it finds.

These changes should result in better random numbers from rand().

92.2.21 Improved `qw//` operator

The `qw//` operator is now evaluated at compile time into a true list instead of being replaced with a run time call to `split()`. This removes the confusing misbehaviour of `qw//` in scalar context, which had inherited that behaviour from split().

Thus:

```
    $foo = ($bar) = qw(a b c); print "$foo|$bar\n";
```

now correctly prints "3|a", instead of "2|a".

92.2.22 Better worst-case behavior of hashes

Small changes in the hashing algorithm have been implemented in order to improve the distribution of lower order bits in the hashed value. This is expected to yield better performance on keys that are repeated sequences.

92.2.23 pack() format 'Z' supported

The new format type 'Z' is useful for packing and unpacking null-terminated strings. See pack in *perlfunc*.

92.2.24 pack() format modifier '!' supported

The new format type modifier '!' is useful for packing and unpacking native shorts, ints, and longs. See pack in *perlfunc*.

92.2.25 pack() and unpack() support counted strings

The template character '/' can be used to specify a counted string type to be packed or unpacked. See pack in *perlfunc*.

92.2.26 Comments in pack() templates

The '#' character in a template introduces a comment up to end of the line. This facilitates documentation of pack() templates.

92.2.27 Weak references

In previous versions of Perl, you couldn't cache objects so as to allow them to be deleted if the last reference from outside the cache is deleted. The reference in the cache would hold a reference count on the object and the objects would never be destroyed.

Another familiar problem is with circular references. When an object references itself, its reference count would never go down to zero, and it would not get destroyed until the program is about to exit.

Weak references solve this by allowing you to "weaken" any reference, that is, make it not count towards the reference count. When the last non-weak reference to an object is deleted, the object is destroyed and all the weak references to the object are automatically undef-ed.

To use this feature, you need the Devel::WeakRef package from CPAN, which contains additional documentation.

```
    NOTE: This is an experimental feature.  Details are subject to change.
```

92.2.28 Binary numbers supported

Binary numbers are now supported as literals, in s?printf formats, and `oct()`:

```
    $answer = 0b101010;
    printf "The answer is: %b\n", oct("0b101010");
```

92.2.29 Lvalue subroutines

Subroutines can now return modifiable lvalues. See Lvalue subroutines in *perlsub*.

```
    NOTE: This is an experimental feature.  Details are subject to change.
```

92.2.30 Some arrows may be omitted in calls through references

Perl now allows the arrow to be omitted in many constructs involving subroutine calls through references. For example, `$foo[10]->('foo')` may now be written `$foo[10]('foo')`. This is rather similar to how the arrow may be omitted from `$foo[10]->{'foo'}`. Note however, that the arrow is still required for `foo(10)->('bar')`.

92.2.31 Boolean assignment operators are legal lvalues

Constructs such as `($a ||= 2) += 1` are now allowed.

92.2.32 exists() is supported on subroutine names

The exists() builtin now works on subroutine names. A subroutine is considered to exist if it has been declared (even if implicitly). See exists in *perlfunc* for examples.

92.2.33 exists() and delete() are supported on array elements

The exists() and delete() builtins now work on simple arrays as well. The behavior is similar to that on hash elements.

exists() can be used to check whether an array element has been initialized. This avoids autovivifying array elements that don't exist. If the array is tied, the EXISTS() method in the corresponding tied package will be invoked.

delete() may be used to remove an element from the array and return it. The array element at that position returns to its uninitialized state, so that testing for the same element with exists() will return false. If the element happens to be the one at the end, the size of the array also shrinks up to the highest element that tests true for exists(), or 0 if none such is found. If the array is tied, the DELETE() method in the corresponding tied package will be invoked.

See exists in *perlfunc* and delete in *perlfunc* for examples.

92.2.34 Pseudo-hashes work better

Dereferencing some types of reference values in a pseudo-hash, such as `$ph->{foo}[1]`, was accidentally disallowed. This has been corrected.

When applied to a pseudo-hash element, exists() now reports whether the specified value exists, not merely if the key is valid.

delete() now works on pseudo-hashes. When given a pseudo-hash element or slice it deletes the values corresponding to the keys (but not the keys themselves). See Pseudo-hashes: Using an array as a hash in *perlref*.

Pseudo-hash slices with constant keys are now optimized to array lookups at compile-time.

List assignments to pseudo-hash slices are now supported.

The `fields` pragma now provides ways to create pseudo-hashes, via fields::new() and fields::phash(). See *fields*.

```
    NOTE: The pseudo-hash data type continues to be experimental.
    Limiting oneself to the interface elements provided by the
    fields pragma will provide protection from any future changes.
```

92.2.35 Automatic flushing of output buffers

fork(), exec(), system(), qx//, and pipe open()s now flush buffers of all files opened for output when the operation was attempted. This mostly eliminates confusing buffering mishaps suffered by users unaware of how Perl internally handles I/O.

This is not supported on some platforms like Solaris where a suitably correct implementation of fflush(NULL) isn't available.

92.2.36 Better diagnostics on meaningless filehandle operations

Constructs such as `open(<FH>)` and `close(<FH>)` are compile time errors. Attempting to read from filehandles that were opened only for writing will now produce warnings (just as writing to read-only filehandles does).

92.2.37 Where possible, buffered data discarded from duped input filehandle

`open(NEW, "<&OLD")` now attempts to discard any data that was previously read and buffered in `OLD` before duping the handle. On platforms where doing this is allowed, the next read operation on `NEW` will return the same data as the corresponding operation on `OLD`. Formerly, it would have returned the data from the start of the following disk block instead.

92.2.38 eof() has the same old magic as <>

`eof()` would return true if no attempt to read from <> had yet been made. `eof()` has been changed to have a little magic of its own, it now opens the <> files.

92.2.39 binmode() can be used to set :crlf and :raw modes

binmode() now accepts a second argument that specifies a discipline for the handle in question. The two pseudo-disciplines ":raw" and ":crlf" are currently supported on DOS-derivative platforms. See binmode in *perlfunc* and *open*.

92.2.40 `-T` filetest recognizes UTF-8 encoded files as "text"

The algorithm used for the `-T` filetest has been enhanced to correctly identify UTF-8 content as "text".

92.2.41 system(), backticks and pipe open now reflect exec() failure

On Unix and similar platforms, system(), qx() and open(FOO, "cmd |") etc., are implemented via fork() and exec(). When the underlying exec() fails, earlier versions did not report the error properly, since the exec() happened to be in a different process.

The child process now communicates with the parent about the error in launching the external command, which allows these constructs to return with their usual error value and set $!.

92.2.42 Improved diagnostics

Line numbers are no longer suppressed (under most likely circumstances) during the global destruction phase.

Diagnostics emitted from code running in threads other than the main thread are now accompanied by the thread ID.

Embedded null characters in diagnostics now actually show up. They used to truncate the message in prior versions.

$foo::a and $foo::b are now exempt from "possible typo" warnings only if sort() is encountered in package `foo`.

Unrecognized alphabetic escapes encountered when parsing quote constructs now generate a warning, since they may take on new semantics in later versions of Perl.

Many diagnostics now report the internal operation in which the warning was provoked, like so:

```
    Use of uninitialized value in concatenation (.) at (eval 1) line 1.
    Use of uninitialized value in print at (eval 1) line 1.
```

Diagnostics that occur within eval may also report the file and line number where the eval is located, in addition to the eval sequence number and the line number within the evaluated text itself. For example:

```
    Not enough arguments for scalar at (eval 4)[newlib/perl5db.pl:1411] line 2, at EOF
```

92.2.43 Diagnostics follow STDERR

Diagnostic output now goes to whichever file the `STDERR` handle is pointing at, instead of always going to the underlying C runtime library's `stderr`.

92.2.44 More consistent close-on-exec behavior

On systems that support a close-on-exec flag on filehandles, the flag is now set for any handles created by pipe(), socketpair(), socket(), and accept(), if that is warranted by the value of $^F that may be in effect. Earlier versions neglected to set the flag for handles created with these operators. See pipe in *perlfunc*, socketpair in *perlfunc*, socket in *perlfunc*, accept in *perlfunc*, and $^F in *perlvar*.

92.2.45 syswrite() ease-of-use

The length argument of `syswrite()` has become optional.

92.2.46 Better syntax checks on parenthesized unary operators

Expressions such as:

```
    print defined(&foo,&bar,&baz);
    print uc("foo","bar","baz");
    undef($foo,&bar);
```

used to be accidentally allowed in earlier versions, and produced unpredictable behaviour. Some produced ancillary warnings when used in this way; others silently did the wrong thing.

The parenthesized forms of most unary operators that expect a single argument now ensure that they are not called with more than one argument, making the cases shown above syntax errors. The usual behaviour of:

```
    print defined &foo, &bar, &baz;
    print uc "foo", "bar", "baz";
    undef $foo, &bar;
```

remains unchanged. See *perlop*.

92.2.47 Bit operators support full native integer width

The bit operators (& | ˆ ˜ << >>) now operate on the full native integral width (the exact size of which is available in $Config{ivsize}). For example, if your platform is either natively 64-bit or if Perl has been configured to use 64-bit integers, these operations apply to 8 bytes (as opposed to 4 bytes on 32-bit platforms). For portability, be sure to mask off the excess bits in the result of unary ˜, e.g., `˜$x & 0xffffffff`.

92.2.48 Improved security features

More potentially unsafe operations taint their results for improved security.

The `passwd` and `shell` fields returned by the getpwent(), getpwnam(), and getpwuid() are now tainted, because the user can affect their own encrypted password and login shell.

The variable modified by shmread(), and messages returned by msgrcv() (and its object-oriented interface IPC::SysV::Msg::rcv) are also tainted, because other untrusted processes can modify messages and shared memory segments for their own nefarious purposes.

92.2.49 More functional bareword prototype (*)

Bareword prototypes have been rationalized to enable them to be used to override builtins that accept barewords and interpret them in a special way, such as `require` or `do`.

Arguments prototyped as * will now be visible within the subroutine as either a simple scalar or as a reference to a typeglob. See Prototypes in *perlsub*.

92.2.50 `require` and do may be overridden

`require` and `do 'file'` operations may be overridden locally by importing subroutines of the same name into the current package (or globally by importing them into the CORE::GLOBAL:: namespace). Overriding `require` will also affect `use`, provided the override is visible at compile-time. See Overriding Built-in Functions in *perlsub*.

92.2.51 $ ˆX variables may now have names longer than one character

Formerly, $ˆX was synonymous with ${"\cX"}, but $ˆXY was a syntax error. Now variable names that begin with a control character may be arbitrarily long. However, for compatibility reasons, these variables *must* be written with explicit braces, as `${ˆXY}` for example. `${ˆXYZ}` is synonymous with ${"\cXYZ"}. Variable names with more than one control character, such as `${ˆXYˆZ}`, are illegal.

The old syntax has not changed. As before, 'ˆX' may be either a literal control-X character or the two-character sequence 'caret' plus 'X'. When braces are omitted, the variable name stops after the control character. Thus `"$ˆXYZ"` continues to be synonymous with `$ˆX . "YZ"` as before.

As before, lexical variables may not have names beginning with control characters. As before, variables whose names begin with a control character are always forced to be in package 'main'. All such variables are reserved for future extensions, except those that begin with `ˆ_`, which may be used by user programs and are guaranteed not to acquire special meaning in any future version of Perl.

92.2.52 New variable $ ˆC reflects `-c` switch

`$^C` has a boolean value that reflects whether perl is being run in compile-only mode (i.e. via the `-c` switch). Since BEGIN blocks are executed under such conditions, this variable enables perl code to determine whether actions that make sense only during normal running are warranted. See *perlvar*.

92.2.53 New variable $ ˆV contains Perl version as a string

`$^V` contains the Perl version number as a string composed of characters whose ordinals match the version numbers, i.e. v5.6.0. This may be used in string comparisons.

See `Support for strings represented as a vector of ordinals` for an example.

92.2.54 Optional Y2K warnings

If Perl is built with the cpp macro `PERL_Y2KWARN` defined, it emits optional warnings when concatenating the number 19 with another number.

This behavior must be specifically enabled when running Configure. See *INSTALL* and *README.Y2K*.

92.2.55 Arrays now always interpolate into double-quoted strings

In double-quoted strings, arrays now interpolate, no matter what. The behavior in earlier versions of perl 5 was that arrays would interpolate into strings if the array had been mentioned before the string was compiled, and otherwise Perl would raise a fatal compile-time error. In versions 5.000 through 5.003, the error was

```
        Literal @example now requires backslash
```

In versions 5.004_01 through 5.6.0, the error was

```
        In string, @example now must be written as \@example
```

The idea here was to get people into the habit of writing `"fred\@example.com"` when they wanted a literal `@` sign, just as they have always written `"Give me back my \$5"` when they wanted a literal `$` sign.

Starting with 5.6.1, when Perl now sees an `@` sign in a double-quoted string, it *always* attempts to interpolate an array, regardless of whether or not the array has been used or declared already. The fatal error has been downgraded to an optional warning:

```
        Possible unintended interpolation of @example in string
```

This warns you that `"fred@example.com"` is going to turn into `fred.com` if you don't backslash the `@`. See http://www.plover.com/˜mjd/perl/at-error.html for more details about the history here.

92.3 Modules and Pragmata

92.3.1 Modules

attributes

While used internally by Perl as a pragma, this module also provides a way to fetch subroutine and variable attributes. See *attributes*.

B

The Perl Compiler suite has been extensively reworked for this release. More of the standard Perl testsuite passes when run under the Compiler, but there is still a significant way to go to achieve production quality compiled executables.

```
    NOTE: The Compiler suite remains highly experimental.  The
    generated code may not be correct, even when it manages to execute
    without errors.
```

Benchmark

Overall, Benchmark results exhibit lower average error and better timing accuracy.

You can now run tests for *n* seconds instead of guessing the right number of tests to run: e.g., timethese(-5, ...) will run each code for at least 5 CPU seconds. Zero as the "number of repetitions" means "for at least 3 CPU seconds". The output format has also changed. For example:

```
   use Benchmark;$x=3;timethese(-5,{a=>sub{$x*$x},b=>sub{$x**2}})
```

will now output something like this:

```
   Benchmark: running a, b, each for at least 5 CPU seconds...
            a:  5 wallclock secs ( 5.77 usr +  0.00 sys =  5.77 CPU) @ 200551.91/s (n=1156516)
            b:  4 wallclock secs ( 5.00 usr +  0.02 sys =  5.02 CPU) @ 159605.18/s (n=800686)
```

New features: "each for at least N CPU seconds...", "wallclock secs", and the "@ operations/CPU second (n=operations)".

timethese() now returns a reference to a hash of Benchmark objects containing the test results, keyed on the names of the tests.

timethis() now returns the iterations field in the Benchmark result object instead of 0.

timethese(), timethis(), and the new cmpthese() (see below) can also take a format specifier of 'none' to suppress output.

A new function countit() is just like timeit() except that it takes a TIME instead of a COUNT.

A new function cmpthese() prints a chart comparing the results of each test returned from a timethese() call. For each possible pair of tests, the percentage speed difference (iters/sec or seconds/iter) is shown.

For other details, see *Benchmark*.

ByteLoader

The ByteLoader is a dedicated extension to generate and run Perl bytecode. See *ByteLoader*.

constant

References can now be used.

The new version also allows a leading underscore in constant names, but disallows a double leading underscore (as in "__LINE__"). Some other names are disallowed or warned against, including BEGIN, END, etc. Some names which were forced into main:: used to fail silently in some cases; now they're fatal (outside of main::) and an optional warning (inside of main::). The ability to detect whether a constant had been set with a given name has been added.

See *constant*.

charnames

This pragma implements the `\N` string escape. See *charnames*.

Data::Dumper

A `Maxdepth` setting can be specified to avoid venturing too deeply into deep data structures. See *Data::Dumper*.

The XSUB implementation of Dump() is now automatically called if the `Useqq` setting is not in use.

Dumping `qr//` objects works correctly.

DB

`DB` is an experimental module that exposes a clean abstraction to Perl's debugging API.

DB_File

DB_File can now be built with Berkeley DB versions 1, 2 or 3. See `ext/DB_File/Changes`.

Devel::DProf

Devel::DProf, a Perl source code profiler has been added. See *Devel::DProf* and *dprofpp*.

Devel::Peek

The Devel::Peek module provides access to the internal representation of Perl variables and data. It is a data debugging tool for the XS programmer.

Dumpvalue

The Dumpvalue module provides screen dumps of Perl data.

DynaLoader

DynaLoader now supports a dl_unload_file() function on platforms that support unloading shared objects using dlclose().

Perl can also optionally arrange to unload all extension shared objects loaded by Perl. To enable this, build Perl with the Configure option `-Accflags=-DDL_UNLOAD_ALL_AT_EXIT`. (This maybe useful if you are using Apache with mod_perl.)

English

$PERL_VERSION now stands for `$^V` (a string value) rather than for `$]` (a numeric value).

Env

Env now supports accessing environment variables like PATH as array variables.

Fcntl

More Fcntl constants added: F_SETLK64, F_SETLKW64, O_LARGEFILE for large file (more than 4GB) access (NOTE: the O_LARGEFILE is automatically added to sysopen() flags if large file support has been configured, as is the default), Free/Net/OpenBSD locking behaviour flags F_FLOCK, F_POSIX, Linux F_SHLCK, and O_ACCMODE: the combined mask of O_RDONLY, O_WRONLY, and O_RDWR. The seek()/sysseek() constants SEEK_SET, SEEK_CUR, and SEEK_END are available via the `:seek` tag. The chmod()/stat() S_IF* constants and S_IS* functions are available via the `:mode` tag.

File::Compare

A compare_text() function has been added, which allows custom comparison functions. See *File::Compare*.

File::Find

File::Find now works correctly when the wanted() function is either autoloaded or is a symbolic reference.

A bug that caused File::Find to lose track of the working directory when pruning top-level directories has been fixed.

File::Find now also supports several other options to control its behavior. It can follow symbolic links if the `follow` option is specified. Enabling the `no_chdir` option will make File::Find skip changing the current directory when walking directories. The `untaint` flag can be useful when running with taint checks enabled.

See *File::Find*.

File::Glob

This extension implements BSD-style file globbing. By default, it will also be used for the internal implementation of the glob() operator. See *File::Glob*.

File::Spec

New methods have been added to the File::Spec module: devnull() returns the name of the null device (/dev/null on Unix) and tmpdir() the name of the temp directory (normally /tmp on Unix). There are now also methods to convert between absolute and relative filenames: abs2rel() and rel2abs(). For compatibility with operating systems that specify volume names in file paths, the splitpath(), splitdir(), and catdir() methods have been added.

File::Spec::Functions

The new File::Spec::Functions modules provides a function interface to the File::Spec module. Allows shorthand

```
    $fullname = catfile($dir1, $dir2, $file);
```

instead of

```
    $fullname = File::Spec->catfile($dir1, $dir2, $file);
```

Getopt::Long

Getopt::Long licensing has changed to allow the Perl Artistic License as well as the GPL. It used to be GPL only, which got in the way of non-GPL applications that wanted to use Getopt::Long.

Getopt::Long encourages the use of Pod::Usage to produce help messages. For example:

```
    use Getopt::Long;
    use Pod::Usage;
    my $man = 0;
    my $help = 0;
    GetOptions('help|?' => \$help, man => \$man) or pod2usage(2);
    pod2usage(1) if $help;
    pod2usage(-exitstatus => 0, -verbose => 2) if $man;

    __END__

    =head1 NAME

    sample - Using Getopt::Long and Pod::Usage

    =head1 SYNOPSIS

    sample [options] [file ...]

     Options:
       -help            brief help message
       -man             full documentation

    =head1 OPTIONS

    =over 8

    =item B<-help>

    Print a brief help message and exits.

    =item B<-man>

    Prints the manual page and exits.

    =back

    =head1 DESCRIPTION
    B<This program> will read the given input file(s) and do something
    useful with the contents thereof.

    =cut
```

See *Pod::Usage* for details.

A bug that prevented the non-option call-back <> from being specified as the first argument has been fixed.

To specify the characters < and > as option starters, use ><. Note, however, that changing option starters is strongly deprecated.

IO

write() and syswrite() will now accept a single-argument form of the call, for consistency with Perl's syswrite().

You can now create a TCP-based IO::Socket::INET without forcing a connect attempt. This allows you to configure its options (like making it non-blocking) and then call connect() manually.

A bug that prevented the IO::Socket::protocol() accessor from ever returning the correct value has been corrected.

IO::Socket::connect now uses non-blocking IO instead of alarm() to do connect timeouts.

IO::Socket::accept now uses select() instead of alarm() for doing timeouts.

IO::Socket::INET->new now sets $! correctly on failure. $@ is still set for backwards compatibility.

JPL

Java Perl Lingo is now distributed with Perl. See jpl/README for more information.

lib

`use lib` now weeds out any trailing duplicate entries. `no lib` removes all named entries.

Math::BigInt

The bitwise operations <<, >>, &, |, and ˜ are now supported on bigints.

Math::Complex

The accessor methods Re, Im, arg, abs, rho, and theta can now also act as mutators (accessor $z->Re(), mutator $z->Re(3)).

The class method `display_format` and the corresponding object method `display_format`, in addition to accepting just one argument, now can also accept a parameter hash. Recognized keys of a parameter hash are `"style"`, which corresponds to the old one parameter case, and two new parameters: `"format"`, which is a printf()-style format string (defaults usually to `"%.15g"`, you can revert to the default by setting the format string to `undef`) used for both parts of a complex number, and `"polar_pretty_print"` (defaults to true), which controls whether an attempt is made to try to recognize small multiples and rationals of pi (2pi, pi/2) at the argument (angle) of a polar complex number.

The potentially disruptive change is that in list context both methods now *return the parameter hash*, instead of only the value of the `"style"` parameter.

Math::Trig

A little bit of radial trigonometry (cylindrical and spherical), radial coordinate conversions, and the great circle distance were added.

Pod::Parser, Pod::InputObjects

Pod::Parser is a base class for parsing and selecting sections of pod documentation from an input stream. This module takes care of identifying pod paragraphs and commands in the input and hands off the parsed paragraphs and commands to user-defined methods which are free to interpret or translate them as they see fit.

Pod::InputObjects defines some input objects needed by Pod::Parser, and for advanced users of Pod::Parser that need more about a command besides its name and text.

As of release 5.6.0 of Perl, Pod::Parser is now the officially sanctioned "base parser code" recommended for use by all pod2xxx translators. Pod::Text (pod2text) and Pod::Man (pod2man) have already been converted to use

Pod::Parser and efforts to convert Pod::HTML (pod2html) are already underway. For any questions or comments about pod parsing and translating issues and utilities, please use the pod-people@perl.org mailing list.

For further information, please see *Pod::Parser* and *Pod::InputObjects*.

Pod::Checker, podchecker

This utility checks pod files for correct syntax, according to *perlpod*. Obvious errors are flagged as such, while warnings are printed for mistakes that can be handled gracefully. The checklist is not complete yet. See *Pod::Checker*.

Pod::ParseUtils, Pod::Find

These modules provide a set of gizmos that are useful mainly for pod translators. Pod::Find traverses directory structures and returns found pod files, along with their canonical names (like `File::Spec::Unix`). Pod::ParseUtils contains **Pod::List** (useful for storing pod list information), **Pod::Hyperlink** (for parsing the contents of `L<>` sequences) and **Pod::Cache** (for caching information about pod files, e.g., link nodes).

Pod::Select, podselect

Pod::Select is a subclass of Pod::Parser which provides a function named "podselect()" to filter out user-specified sections of raw pod documentation from an input stream. podselect is a script that provides access to Pod::Select from other scripts to be used as a filter. See *Pod::Select*.

Pod::Usage, pod2usage

Pod::Usage provides the function "pod2usage()" to print usage messages for a Perl script based on its embedded pod documentation. The pod2usage() function is generally useful to all script authors since it lets them write and maintain a single source (the pods) for documentation, thus removing the need to create and maintain redundant usage message text consisting of information already in the pods.

There is also a pod2usage script which can be used from other kinds of scripts to print usage messages from pods (even for non-Perl scripts with pods embedded in comments).

For details and examples, please see *Pod::Usage*.

Pod::Text and Pod::Man

Pod::Text has been rewritten to use Pod::Parser. While pod2text() is still available for backwards compatibility, the module now has a new preferred interface. See *Pod::Text* for the details. The new Pod::Text module is easily subclassed for tweaks to the output, and two such subclasses (Pod::Text::Termcap for man-page-style bold and underlining using termcap information, and Pod::Text::Color for markup with ANSI color sequences) are now standard.

pod2man has been turned into a module, Pod::Man, which also uses Pod::Parser. In the process, several outstanding bugs related to quotes in section headers, quoting of code escapes, and nested lists have been fixed. pod2man is now a wrapper script around this module.

SDBM_File

An EXISTS method has been added to this module (and sdbm_exists() has been added to the underlying sdbm library), so one can now call exists on an SDBM_File tied hash and get the correct result, rather than a runtime error.

A bug that may have caused data loss when more than one disk block happens to be read from the database in a single FETCH() has been fixed.

Sys::Syslog

Sys::Syslog now uses XSUBs to access facilities from syslog.h so it no longer requires syslog.ph to exist.

Sys::Hostname

Sys::Hostname now uses XSUBs to call the C library's gethostname() or uname() if they exist.

Term::ANSIColor

Term::ANSIColor is a very simple module to provide easy and readable access to the ANSI color and highlighting escape sequences, supported by most ANSI terminal emulators. It is now included standard.

Time::Local

The timelocal() and timegm() functions used to silently return bogus results when the date fell outside the machine's integer range. They now consistently croak() if the date falls in an unsupported range.

Win32

The error return value in list context has been changed for all functions that return a list of values. Previously these functions returned a list with a single element `undef` if an error occurred. Now these functions return the empty list in these situations. This applies to the following functions:

```
Win32::FsType
Win32::GetOSVersion
```

The remaining functions are unchanged and continue to return `undef` on error even in list context.

The Win32::SetLastError(ERROR) function has been added as a complement to the Win32::GetLastError() function.

The new Win32::GetFullPathName(FILENAME) returns the full absolute pathname for FILENAME in scalar context. In list context it returns a two-element list containing the fully qualified directory name and the filename. See *Win32*.

XSLoader

The XSLoader extension is a simpler alternative to DynaLoader. See *XSLoader*.

DBM Filters

A new feature called "DBM Filters" has been added to all the DBM modules–DB_File, GDBM_File, NDBM_File, ODBM_File, and SDBM_File. DBM Filters add four new methods to each DBM module:

```
filter_store_key
filter_store_value
filter_fetch_key
filter_fetch_value
```

These can be used to filter key-value pairs before the pairs are written to the database or just after they are read from the database. See *perldbmfilter* for further information.

92.3.2 Pragmata

`use attrs` is now obsolete, and is only provided for backward-compatibility. It's been replaced by the `sub : attributes` syntax. See Subroutine Attributes in *perlsub* and *attributes*.

Lexical warnings pragma, `use warnings;`, to control optional warnings. See *perllexwarn*.

`use filetest` to control the behaviour of filetests (`-r` `-w` ...). Currently only one subpragma implemented, "use filetest 'access';", that uses access(2) or equivalent to check permissions instead of using stat(2) as usual. This matters in filesystems where there are ACLs (access control lists): the stat(2) might lie, but access(2) knows better.

The `open` pragma can be used to specify default disciplines for handle constructors (e.g. open()) and for qx//. The two pseudo-disciplines `:raw` and `:crlf` are currently supported on DOS-derivative platforms (i.e. where binmode is not a no-op). See also §92.2.39.

92.4 Utility Changes

92.4.1 dprofpp

`dprofpp` is used to display profile data generated using `Devel::DProf`. See *dprofpp*.

92.4.2 find2perl

The `find2perl` utility now uses the enhanced features of the File::Find module. The -depth and -follow options are supported. Pod documentation is also included in the script.

92.4.3 h2xs

The `h2xs` tool can now work in conjunction with `C::Scan` (available from CPAN) to automatically parse real-life header files. The `-M`, `-a`, `-k`, and `-o` options are new.

92.4.4 perlcc

`perlcc` now supports the C and Bytecode backends. By default, it generates output from the simple C backend rather than the optimized C backend.

Support for non-Unix platforms has been improved.

92.4.5 perldoc

`perldoc` has been reworked to avoid possible security holes. It will not by default let itself be run as the superuser, but you may still use the **-U** switch to try to make it drop privileges first.

92.4.6 The Perl Debugger

Many bug fixes and enhancements were added to *perl5db.pl*, the Perl debugger. The help documentation was rearranged. New commands include `< ?`, `> ?`, and `{ ?` to list out current actions, `man` *`docpage`* to run your doc viewer on some perl docset, and support for quoted options. The help information was rearranged, and should be viewable once again if you're using **less** as your pager. A serious security hole was plugged–you should immediately remove all older versions of the Perl debugger as installed in previous releases, all the way back to perl3, from your system to avoid being bitten by this.

92.5 Improved Documentation

Many of the platform-specific README files are now part of the perl installation. See *perl* for the complete list.

perlapi.pod

The official list of public Perl API functions.

perlboot.pod

A tutorial for beginners on object-oriented Perl.

perlcompile.pod

An introduction to using the Perl Compiler suite.

perldbmfilter.pod

A howto document on using the DBM filter facility.

perldebug.pod

All material unrelated to running the Perl debugger, plus all low-level guts-like details that risked crushing the casual user of the debugger, have been relocated from the old manpage to the next entry below.

perldebguts.pod

This new manpage contains excessively low-level material not related to the Perl debugger, but slightly related to debugging Perl itself. It also contains some arcane internal details of how the debugging process works that may only be of interest to developers of Perl debuggers.

perlfork.pod

Notes on the fork() emulation currently available for the Windows platform.

perlfilter.pod

An introduction to writing Perl source filters.

perlhack.pod

Some guidelines for hacking the Perl source code.

perlintern.pod

A list of internal functions in the Perl source code. (List is currently empty.)

perllexwarn.pod

Introduction and reference information about lexically scoped warning categories.

perlnumber.pod

Detailed information about numbers as they are represented in Perl.

perlopentut.pod

A tutorial on using open() effectively.

perlreftut.pod

A tutorial that introduces the essentials of references.

perltootc.pod

A tutorial on managing class data for object modules.

perltodo.pod

Discussion of the most often wanted features that may someday be supported in Perl.

perlunicode.pod

An introduction to Unicode support features in Perl.

92.6 Performance enhancements

92.6.1 Simple sort() using { $ a <=> $ b } and the like are optimized

Many common sort() operations using a simple inlined block are now optimized for faster performance.

92.6.2 Optimized assignments to lexical variables

Certain operations in the RHS of assignment statements have been optimized to directly set the lexical variable on the LHS, eliminating redundant copying overheads.

92.6.3 Faster subroutine calls

Minor changes in how subroutine calls are handled internally provide marginal improvements in performance.

92.6.4 delete(), each(), values() and hash iteration are faster

The hash values returned by delete(), each(), values() and hashes in a list context are the actual values in the hash, instead of copies. This results in significantly better performance, because it eliminates needless copying in most situations.

92.7 Installation and Configuration Improvements

92.7.1 -Dusethreads means something different

The -Dusethreads flag now enables the experimental interpreter-based thread support by default. To get the flavor of experimental threads that was in 5.005 instead, you need to run Configure with "-Dusethreads -Duse5005threads".

As of v5.6.0, interpreter-threads support is still lacking a way to create new threads from Perl (i.e., `use Thread;` will not work with interpreter threads). `use Thread;` continues to be available when you specify the -Duse5005threads option to Configure, bugs and all.

```
    NOTE: Support for threads continues to be an experimental feature.
    Interfaces and implementation are subject to sudden and drastic changes.
```

92.7.2 New Configure flags

The following new flags may be enabled on the Configure command line by running Configure with `-Dflag`.

```
    usemultiplicity
    usethreads useithreads      (new interpreter threads: no Perl API yet)
    usethreads use5005threads   (threads as they were in 5.005)

    use64bitint                 (equal to now deprecated 'use64bits')
    use64bitall

    uselongdouble
    usemorebits
    uselargefiles
    usesocks                    (only SOCKS v5 supported)
```

92.7.3 Threadedness and 64-bitness now more daring

The Configure options enabling the use of threads and the use of 64-bitness are now more daring in the sense that they no more have an explicit list of operating systems of known threads/64-bit capabilities. In other words: if your operating system has the necessary APIs and datatypes, you should be able just to go ahead and use them, for threads by Configure -Dusethreads, and for 64 bits either explicitly by Configure -Duse64bitint or implicitly if your system has 64-bit wide datatypes. See also §92.2.11.

92.7.4 Long Doubles

Some platforms have "long doubles", floating point numbers of even larger range than ordinary "doubles". To enable using long doubles for Perl's scalars, use -Duselongdouble.

92.7.5 -Dusemorebits

You can enable both -Duse64bitint and -Duselongdouble with -Dusemorebits. See also §92.2.11.

92.7.6 -Duselargefiles

Some platforms support system APIs that are capable of handling large files (typically, files larger than two gigabytes). Perl will try to use these APIs if you ask for -Duselargefiles.

See §92.2.12 for more information.

92.7.7 installusrbinperl

You can use "Configure -Uinstallusrbinperl" which causes installperl to skip installing perl also as /usr/bin/perl. This is useful if you prefer not to modify /usr/bin for some reason or another but harmful because many scripts assume to find Perl in /usr/bin/perl.

92.7.8 SOCKS support

You can use "Configure -Dusesocks" which causes Perl to probe for the SOCKS proxy protocol library (v5, not v4). For more information on SOCKS, see:

```
    http://www.socks.nec.com/
```

92.7.9 `-A` flag

You can "post-edit" the Configure variables using the Configure `-A` switch. The editing happens immediately after the platform specific hints files have been processed but before the actual configuration process starts. Run `Configure -h` to find out the full `-A` syntax.

92.7.10 Enhanced Installation Directories

The installation structure has been enriched to improve the support for maintaining multiple versions of perl, to provide locations for vendor-supplied modules, scripts, and manpages, and to ease maintenance of locally-added modules, scripts, and manpages. See the section on Installation Directories in the INSTALL file for complete details. For most users building and installing from source, the defaults should be fine.

If you previously used `Configure -Dsitelib` or `-Dsitearch` to set special values for library directories, you might wish to consider using the new `-Dsiteprefix` setting instead. Also, if you wish to re-use a config.sh file from an earlier version of perl, you should be sure to check that Configure makes sensible choices for the new directories. See INSTALL for complete details.

92.8 Platform specific changes

92.8.1 Supported platforms

- The Mach CThreads (NEXTSTEP, OPENSTEP) are now supported by the Thread extension.
- GNU/Hurd is now supported.
- Rhapsody/Darwin is now supported.
- EPOC is now supported (on Psion 5).
- The cygwin port (formerly cygwin32) has been greatly improved.

92.8.2 DOS

- Perl now works with djgpp 2.02 (and 2.03 alpha).
- Environment variable names are not converted to uppercase any more.
- Incorrect exit codes from backticks have been fixed.
- This port continues to use its own builtin globbing (not File::Glob).

92.8.3 OS390 (OpenEdition MVS)

Support for this EBCDIC platform has not been renewed in this release. There are difficulties in reconciling Perl's standardization on UTF-8 as its internal representation for characters with the EBCDIC character set, because the two are incompatible.

It is unclear whether future versions will renew support for this platform, but the possibility exists.

92.8.4 VMS

Numerous revisions and extensions to configuration, build, testing, and installation process to accommodate core changes and VMS-specific options.

Expand %ENV-handling code to allow runtime mapping to logical names, CLI symbols, and CRTL environ array.

Extension of subprocess invocation code to accept filespecs as command "verbs".

Add to Perl command line processing the ability to use default file types and to recognize Unix-style `2>&1`.

Expansion of File::Spec::VMS routines, and integration into ExtUtils::MM_VMS.

Extension of ExtUtils::MM_VMS to handle complex extensions more flexibly.

Barewords at start of Unix-syntax paths may be treated as text rather than only as logical names.

Optional secure translation of several logical names used internally by Perl.

Miscellaneous bugfixing and porting of new core code to VMS.

Thanks are gladly extended to the many people who have contributed VMS patches, testing, and ideas.

92.8.5 Win32

Perl can now emulate fork() internally, using multiple interpreters running in different concurrent threads. This support must be enabled at build time. See *perlfork* for detailed information.

When given a pathname that consists only of a drivename, such as `A:`, opendir() and stat() now use the current working directory for the drive rather than the drive root.

The builtin XSUB functions in the Win32:: namespace are documented. See *Win32*.

$^X now contains the full path name of the running executable.

A Win32::GetLongPathName() function is provided to complement Win32::GetFullPathName() and Win32::GetShortPathName(). See *Win32*.

POSIX::uname() is supported.

system(1,...) now returns true process IDs rather than process handles. kill() accepts any real process id, rather than strictly return values from system(1,...).

For better compatibility with Unix, `kill(0, $pid)` can now be used to test whether a process exists.

The `Shell` module is supported.

Better support for building Perl under command.com in Windows 95 has been added.

Scripts are read in binary mode by default to allow ByteLoader (and the filter mechanism in general) to work properly. For compatibility, the DATA filehandle will be set to text mode if a carriage return is detected at the end of the line containing the __END__ or __DATA__ token; if not, the DATA filehandle will be left open in binary mode. Earlier versions always opened the DATA filehandle in text mode.

The glob() operator is implemented via the `File::Glob` extension, which supports glob syntax of the C shell. This increases the flexibility of the glob() operator, but there may be compatibility issues for programs that relied on the older globbing syntax. If you want to preserve compatibility with the older syntax, you might want to run perl with `-MFile::DosGlob`. For details and compatibility information, see *File::Glob*.

92.9 Significant bug fixes

92.9.1 <HANDLE> on empty files

With `$/` set to `undef`, "slurping" an empty file returns a string of zero length (instead of `undef`, as it used to) the first time the HANDLE is read after `$/` is set to `undef`. Further reads yield `undef`.

This means that the following will append "foo" to an empty file (it used to do nothing):

```
    perl -0777 -pi -e 's/^/foo/' empty_file
```

The behaviour of:

```
    perl -pi -e 's/^/foo/' empty_file
```

is unchanged (it continues to leave the file empty).

92.9.2 `eval '...'` improvements

Line numbers (as reflected by caller() and most diagnostics) within `eval '...'` were often incorrect where here documents were involved. This has been corrected.

Lexical lookups for variables appearing in `eval '...'` within functions that were themselves called within an `eval '...'` were searching the wrong place for lexicals. The lexical search now correctly ends at the subroutine's block boundary.

The use of `return` within `eval {...}` caused $@ not to be reset correctly when no exception occurred within the eval. This has been fixed.

Parsing of here documents used to be flawed when they appeared as the replacement expression in `eval 's/.../.../e'`. This has been fixed.

92.9.3 All compilation errors are true errors

Some "errors" encountered at compile time were by necessity generated as warnings followed by eventual termination of the program. This enabled more such errors to be reported in a single run, rather than causing a hard stop at the first error that was encountered.

The mechanism for reporting such errors has been reimplemented to queue compile-time errors and report them at the end of the compilation as true errors rather than as warnings. This fixes cases where error messages leaked through in the form of warnings when code was compiled at run time using `eval STRING`, and also allows such errors to be reliably trapped using `eval "..."`.

92.9.4 Implicitly closed filehandles are safer

Sometimes implicitly closed filehandles (as when they are localized, and Perl automatically closes them on exiting the scope) could inadvertently set $? or $!. This has been corrected.

92.9.5 Behavior of list slices is more consistent

When taking a slice of a literal list (as opposed to a slice of an array or hash), Perl used to return an empty list if the result happened to be composed of all undef values.

The new behavior is to produce an empty list if (and only if) the original list was empty. Consider the following example:

```
    @a = (1,undef,undef,2)[2,1,2];
```

The old behavior would have resulted in @a having no elements. The new behavior ensures it has three undefined elements.

Note in particular that the behavior of slices of the following cases remains unchanged:

```
    @a = ()[1,2];
    @a = (getpwent)[7,0];
    @a = (anything_returning_empty_list())[2,1,2];
    @a = @b[2,1,2];
    @a = @c{'a','b','c'};
```

See *perldata*.

92.9.6 `(\$)` prototype and `$foo{a}`

A scalar reference prototype now correctly allows a hash or array element in that slot.

92.9.7 `goto &sub` and AUTOLOAD

The `goto &sub` construct works correctly when `&sub` happens to be autoloaded.

92.9.8 `-bareword` allowed under `use integer`

The autoquoting of barewords preceded by `-` did not work in prior versions when the `integer` pragma was enabled. This has been fixed.

92.9.9 Failures in DESTROY()

When code in a destructor threw an exception, it went unnoticed in earlier versions of Perl, unless someone happened to be looking in $@ just after the point the destructor happened to run. Such failures are now visible as warnings when warnings are enabled.

92.9.10 Locale bugs fixed

printf() and sprintf() previously reset the numeric locale back to the default "C" locale. This has been fixed.

Numbers formatted according to the local numeric locale (such as using a decimal comma instead of a decimal dot) caused "isn't numeric" warnings, even while the operations accessing those numbers produced correct results. These warnings have been discontinued.

92.9.11 Memory leaks

The `eval 'return sub {...}'` construct could sometimes leak memory. This has been fixed.

Operations that aren't filehandle constructors used to leak memory when used on invalid filehandles. This has been fixed.

Constructs that modified `@_` could fail to deallocate values in `@_` and thus leak memory. This has been corrected.

92.9.12 Spurious subroutine stubs after failed subroutine calls

Perl could sometimes create empty subroutine stubs when a subroutine was not found in the package. Such cases stopped later method lookups from progressing into base packages. This has been corrected.

92.9.13 Taint failures under `-U`

When running in unsafe mode, taint violations could sometimes cause silent failures. This has been fixed.

92.9.14 END blocks and the `-c` switch

Prior versions used to run BEGIN **and** END blocks when Perl was run in compile-only mode. Since this is typically not the expected behavior, END blocks are not executed anymore when the `-c` switch is used, or if compilation fails.

See §92.2.18 for how to run things when the compile phase ends.

92.9.15 Potential to leak DATA filehandles

Using the `__DATA__` token creates an implicit filehandle to the file that contains the token. It is the program's responsibility to close it when it is done reading from it.

This caveat is now better explained in the documentation. See *perldata*.

92.10 New or Changed Diagnostics

"%s" variable %s masks earlier declaration in same %s

(W misc) A "my" or "our" 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 referents to it are destroyed.

"my sub" not yet implemented

(F) Lexically scoped subroutines are not yet implemented. Don't try that yet.

"our" variable %s redeclared

(W misc) You seem to have already declared the same global once before in the current lexical scope.

'!' allowed only after types %s

(F) The '!' is allowed in pack() and unpack() only after certain types. See pack in *perlfunc*.

/ cannot take a count

(F) You had an unpack template indicating a counted-length string, but you have also specified an explicit size for the string. See pack in *perlfunc*.

/ must be followed by a, A or Z

(F) You had an unpack template indicating a counted-length string, which must be followed by one of the letters a, A or Z to indicate what sort of string is to be unpacked. See pack in *perlfunc*.

/ must be followed by a*, A* or Z*

(F) You had a pack template indicating a counted-length string, Currently the only things that can have their length counted are a*, A* or Z*. See pack in *perlfunc*.

/ must follow a numeric type

(F) You had an unpack template that contained a '#', but this did not follow some numeric unpack specification. See pack in *perlfunc*.

/%s/: Unrecognized escape \\%c passed through

(W regexp) You used a backslash-character combination which is not recognized by Perl. This combination appears in an interpolated variable or a '-delimited regular expression. The character was understood literally.

/%s/: Unrecognized escape \\%c in character class passed through

(W regexp) You used a backslash-character combination which is not recognized by Perl inside character classes. The character was understood literally.

/%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.

%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 *perlsub*.

%s argument is not a HASH or ARRAY element

(F) The argument to exists() must be a hash or array element, such as:

```
$foo{$bar}
$ref->{"susie"}[12]
```

%s 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"}
```

%s 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.

%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*.

(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 G_KEEPERR in *perlcall*.

<> should be quotes

(F) You wrote `require <file>` when you should have written `require 'file'`.

Attempt to join self

(F) You tried to join a thread from within itself, which is an impossible task. You may be joining the wrong thread, or you may need to move the join() to some other thread.

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 realloc() ignored

(S) An internal routine called realloc() on something that had never been malloc()ed in the first place. Mandatory, but can be disabled by setting environment variable `PERL_BADFREE` to 1.

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.

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 *perlport* for more on portability concerns.

Bit vector size > 32 non-portable

(W portable) Using bit vector sizes larger than 32 is non-portable.

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.

Can't check filesystem of script "%s"

(P) For some reason you can't check the filesystem of the script for nosuid.

Can't declare class for non-scalar %s in "%s"

(S) Currently, only scalar variables can declared with a specific class qualifier in a "my" or "our" declaration. The semantics may be extended for other types of variables in future.

Can't declare %s in "%s"

(F) Only scalar, array, and hash variables may be declared as "my" or "our" variables. They must have ordinary identifiers as names.

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 modify non-lvalue subroutine call

(F) Subroutines meant to be used in lvalue context should be declared as such, see Lvalue subroutines in *perlsub*.

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 *perlvms*) so that environ is not searched.

Can't remove %s: %s, skipping file

(S) 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 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 weaken a nonreference

(F) You attempted to weaken something that was not a reference. Only references can be weakened.

Character class [:%s: unknown]

(F) The class in the character class [: :] syntax is unknown. See *perlre*.

Character class syntax [%s belongs inside] **character classes**

(W unsafe) 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.

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 Constant Functions in *perlsub* and *constant*.

constant(%s): %s

(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` or `charnames` pragma? See *charnames* and *overload*.

CORE::%s is not a keyword

(F) The CORE:: namespace is reserved for Perl keywords.

defined(@array) is deprecated

(D) defined() is not usually 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.

defined(%hash) is deprecated

(D) defined() is not usually useful on hashes because it checks for an undefined *scalar* value. If you want to see if the hash is empty, just use `if (%hash) { # not empty }` for example.

Did not produce a valid header

See Server error.

(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.

Document contains no data

See Server error.

entering effective %s failed

(F) While under the `use filetest` pragma, switching the real and effective uids or gids failed.

false [range "%s" in regexp]

(W regexp) 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 "-". Consider quoting the "-", "\-". See *perlre*.

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 "<" or nothing. If you intended only to read from the file, use "<". See open in *perlfunc*.

flock() on closed filehandle %s

(W closed) The filehandle you're attempting to flock() got itself closed some time before now. Check your logic flow. flock() operates on filehandles. Are you attempting to call flock() on a dirhandle by the same name?

Global symbol "%s" requires explicit package name

(F) You've said "use strict vars", which indicates that all variables must either be lexically scoped (using "my"), declared beforehand using "our", or explicitly qualified to say which package the global variable is in (using "::").

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 *perlport* for more on portability concerns.

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.

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 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).

Integer overflow in %s number

(W 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.

Invalid %s attribute: %s

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

The indicated attributes for a subroutine or variable were not recognized by Perl or by a user-supplied handler. See *attributes*.

invalid [range "%s" in regexp]

The offending range is now explicitly displayed.

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 subroutine attribute list

(F) Something other than a colon or whitespace was seen between the elements of a subroutine attribute list. If the previous attribute had a parenthesised parameter list, perhaps that list was terminated too soon.

leaving effective %s failed

(F) While under the `use filetest` pragma, switching the real and effective uids or gids failed.

Lvalue subs returning %s not implemented yet

(F) Due to limitations in the current implementation, array and hash values cannot be returned in subroutines used in lvalue context. See Lvalue subroutines in *perlsub*.

Method %s not permitted

See Server error.

Missing %sbrace%s on \N{}

(F) Wrong syntax of character name literal `\N{charname}` within double-quotish context.

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 name in "my sub"

(F) The reserved syntax for lexically scoped subroutines requires that they have a name with which they can be found.

No %s specified for -%c

(F) The indicated command line switch needs a mandatory argument, but you haven't specified one.

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 semantics. Such syntax is reserved for future extensions.

No space allowed after -%c

(F) The argument to the indicated command line switch must follow immediately after the switch, without intervening spaces.

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.

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 *perlport* for more on portability concerns.

See also *perlport* for writing portable code.

panic: del_backref

(P) Failed an internal consistency check while trying to reset a weak reference.

panic: kid popen errno read

(F) forked child returned an incomprehensible message about its errno.

panic: magic_killbackrefs

(P) Failed an internal consistency check while trying to reset all weak references to an object.

Parentheses missing around "%s" list

(W parenthesis) You said something like

```
my $foo, $bar = @_;
```

when you meant

```
my ($foo, $bar) = @_;
```

Remember that "my", "our", and "local" bind tighter than comma.

Possible unintended interpolation of %s in string

(W ambiguous) It used to be that Perl would try to guess whether you wanted an array interpolated or a literal @. It no longer does this; arrays are now *always* interpolated into strings. This means that if you try something like:

```
    print "fred@example.com";
```

and the array `@example` doesn't exist, Perl is going to print `fred.com`, which is probably not what you wanted. To get a literal `@` sign in a string, put a backslash before it, just as you would to get a literal `$` sign.

Possible Y2K bug: %s

(W y2k) You are concatenating the number 19 with another number, which could be a potential Year 2000 problem.

pragma "attrs" is deprecated, use "sub NAME : ATTRS" instead

(W deprecated) You have written something like this:

```
sub doit
{
    use attrs qw(locked);
}
```

You should use the new declaration syntax instead.

```
sub doit : locked
{
    ...
```

The `use attrs` pragma is now obsolete, and is only provided for backward-compatibility. See Subroutine Attributes in *perlsub*.

Premature end of script headers

See Server error.

Repeat count in pack overflows

(F) You can't specify a repeat count so large that it overflows your signed integers. See pack in *perlfunc*.

Repeat count in unpack overflows

(F) You can't specify a repeat count so large that it overflows your signed integers. See unpack in *perlfunc*.

realloc() of freed memory ignored

(S) An internal routine called realloc() on something that had already been freed.

Reference is already weak

(W misc) You have attempted to weaken a reference that is already weak. Doing so has no effect.

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.

Strange *+?{} on zero-length expression

(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}/`.

switching effective %s is not implemented

(F) While under the `use filetest` pragma, we cannot switch the real and effective uids or gids.

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 *perlvms*) so that the environ array isn't the target of the change to %ENV which produced the warning.

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.

Unknown open() mode '%s'

(F) The second argument of 3-argument open() is not among the list of valid modes: <, >, >>, +<, +>, +>>, -|, |-.

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.

Unrecognized escape \\%c passed through

(W misc) You used a backslash-character combination which is not recognized by Perl. The character was understood literally.

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 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 subroutine attribute list

(F) The lexer saw an opening (left) parenthesis character while parsing a subroutine 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.

Unterminated subroutine attribute list

(F) The lexer found something other than a simple identifier at the start of a subroutine 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.

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.

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.

92.11 New tests

lib/attrs

Compatibility tests for `sub : attrs` vs the older `use attrs`.

lib/env

Tests for new environment scalar capability (e.g., `use Env qw($BAR);`).

lib/env-array

Tests for new environment array capability (e.g., `use Env qw(@PATH);`).

lib/io_const

IO constants (SEEK_*, _IO*).

lib/io_dir

Directory-related IO methods (new, read, close, rewind, tied delete).

lib/io_multihomed

INET sockets with multi-homed hosts.

lib/io_poll

IO poll().

lib/io_unix

UNIX sockets.

op/attrs

Regression tests for `my ($x,@y,%z) : attrs` and <sub : attrs>.

op/filetest

File test operators.

op/lex_assign

Verify operations that access pad objects (lexicals and temporaries).

op/exists_sub

Verify `exists &sub` operations.

92.12 Incompatible Changes

92.12.1 Perl Source Incompatibilities

Beware that any new warnings that have been added or old ones that have been enhanced are **not** considered incompatible changes.

Since all new warnings must be explicitly requested via the `-w` switch or the `warnings` pragma, it is ultimately the programmer's responsibility to ensure that warnings are enabled judiciously.

CHECK is a new keyword

All subroutine definitions named CHECK are now special. See `/"Support for CHECK blocks"` for more information.

Treatment of list slices of undef has changed

There is a potential incompatibility in the behavior of list slices that are comprised entirely of undefined values. See §92.9.5.

Format of $ English::PERL_VERSION is different

The English module now sets $PERL_VERSION to $^V (a string value) rather than `$]` (a numeric value). This is a potential incompatibility. Send us a report via perlbug if you are affected by this.

See §92.2.7 for the reasons for this change.

Literals of the form `1.2.3` parse differently

Previously, numeric literals with more than one dot in them were interpreted as a floating point number concatenated with one or more numbers. Such "numbers" are now parsed as strings composed of the specified ordinals.

For example, `print 97.98.99` used to output `97.9899` in earlier versions, but now prints `abc`.

See §92.2.6.

Possibly changed pseudo-random number generator

Perl programs that depend on reproducing a specific set of pseudo-random numbers may now produce different output due to improvements made to the rand() builtin. You can use `sh Configure -Drandfunc=rand` to obtain the old behavior.

See §92.2.20.

Hashing function for hash keys has changed

Even though Perl hashes are not order preserving, the apparently random order encountered when iterating on the contents of a hash is actually determined by the hashing algorithm used. Improvements in the algorithm may yield a random order that is **different** from that of previous versions, especially when iterating on hashes.

See §92.2.22 for additional information.

`undef` fails on read only values

Using the `undef` operator on a readonly value (such as $1) has the same effect as assigning `undef` to the readonly value–it throws an exception.

Close-on-exec bit may be set on pipe and socket handles

Pipe and socket handles are also now subject to the close-on-exec behavior determined by the special variable $^F.

See §92.2.44.

Writing `"$$1"` to mean `"${$}1"` is unsupported

Perl 5.004 deprecated the interpretation of `$$1` and similar within interpolated strings to mean `$$ . "1"`, but still allowed it.

In Perl 5.6.0 and later, `"$$1"` always means `"${$1}"`.

delete(), each(), values() and `\(%h)`

operate on aliases to values, not copies

delete(), each(), values() and hashes (e.g. `\(%h)`) in a list context return the actual values in the hash, instead of copies (as they used to in earlier versions). Typical idioms for using these constructs copy the returned values, but this can make a significant difference when creating references to the returned values. Keys in the hash are still returned as copies when iterating on a hash.

See also §92.6.4.

vec(EXPR,OFFSET,BITS) enforces powers-of-two BITS

vec() generates a run-time error if the BITS argument is not a valid power-of-two integer.

Text of some diagnostic output has changed

Most references to internal Perl operations in diagnostics have been changed to be more descriptive. This may be an issue for programs that may incorrectly rely on the exact text of diagnostics for proper functioning.

`%@` has been removed

The undocumented special variable `%@` that used to accumulate "background" errors (such as those that happen in DESTROY()) has been removed, because it could potentially result in memory leaks.

Parenthesized not() behaves like a list operator

The `not` operator now falls under the "if it looks like a function, it behaves like a function" rule.

As a result, the parenthesized form can be used with `grep` and `map`. The following construct used to be a syntax error before, but it works as expected now:

```
grep not($_), @things;
```

On the other hand, using `not` with a literal list slice may not work. The following previously allowed construct:

```
print not (1,2,3)[0];
```

needs to be written with additional parentheses now:

```
print not((1,2,3)[0]);
```

The behavior remains unaffected when `not` is not followed by parentheses.

Semantics of bareword prototype `(*)` have changed

The semantics of the bareword prototype * have changed. Perl 5.005 always coerced simple scalar arguments to a typeglob, which wasn't useful in situations where the subroutine must distinguish between a simple scalar and a typeglob. The new behavior is to not coerce bareword arguments to a typeglob. The value will always be visible as either a simple scalar or as a reference to a typeglob.

See §92.2.49.

Semantics of bit operators may have changed on 64-bit platforms

If your platform is either natively 64-bit or if Perl has been configured to used 64-bit integers, i.e., $Config{ivsize} is 8, there may be a potential incompatibility in the behavior of bitwise numeric operators (& | ^ ~ << >>). These operators used to strictly operate on the lower 32 bits of integers in previous versions, but now operate over the entire native integral width. In particular, note that unary ~ will produce different results on platforms that have different $Config{ivsize}. For portability, be sure to mask off the excess bits in the result of unary ~, e.g., `~$x & 0xffffffff`.

See §92.2.47.

More builtins taint their results

As described in §92.2.48, there may be more sources of taint in a Perl program.

To avoid these new tainting behaviors, you can build Perl with the Configure option `-Accflags=-DINCOMPLETE_TAINTS`. Beware that the ensuing perl binary may be insecure.

92.12.2 C Source Incompatibilities

`PERL_POLLUTE`

Release 5.005 grandfathered old global symbol names by providing preprocessor macros for extension source compatibility. As of release 5.6.0, these preprocessor definitions are not available by default. You need to explicitly compile perl with `-DPERL_POLLUTE` to get these definitions. For extensions still using the old symbols, this option can be specified via MakeMaker:

```
perl Makefile.PL POLLUTE=1
```

`PERL_IMPLICIT_CONTEXT`

This new build option provides a set of macros for all API functions such that an implicit interpreter/thread context argument is passed to every API function. As a result of this, something like `sv_setsv(foo,bar)` amounts to a macro invocation that actually translates to something like `Perl_sv_setsv(my_perl,foo,bar)`. While this is generally expected to not have any significant source compatibility issues, the difference between a macro and a real function call will need to be considered.

This means that there **is** a source compatibility issue as a result of this if your extensions attempt to use pointers to any of the Perl API functions.

Note that the above issue is not relevant to the default build of Perl, whose interfaces continue to match those of prior versions (but subject to the other options described here).

See The Perl API in *perlguts* for detailed information on the ramifications of building Perl with this option.

```
NOTE: PERL_IMPLICIT_CONTEXT is automatically enabled whenever Perl is built
with one of -Dusethreads, -Dusemultiplicity, or both.  It is not
intended to be enabled by users at this time.
```

`PERL_POLLUTE_MALLOC`

Enabling Perl's malloc in release 5.005 and earlier caused the namespace of the system's malloc family of functions to be usurped by the Perl versions, since by default they used the same names. Besides causing problems on platforms that do not allow these functions to be cleanly replaced, this also meant that the system versions could not be called in programs that used Perl's malloc. Previous versions of Perl have allowed this behaviour to be suppressed with the HIDEMYMALLOC and EMBEDMYMALLOC preprocessor definitions.

As of release 5.6.0, Perl's malloc family of functions have default names distinct from the system versions. You need to explicitly compile perl with `-DPERL_POLLUTE_MALLOC` to get the older behaviour. HIDEMYMALLOC and EMBEDMYMALLOC have no effect, since the behaviour they enabled is now the default.

Note that these functions do **not** constitute Perl's memory allocation API. See Memory Allocation in *perlguts* for further information about that.

92.12.3 Compatible C Source API Changes

`PATCHLEVEL` is now `PERL_VERSION`

The cpp macros `PERL_REVISION`, `PERL_VERSION`, and `PERL_SUBVERSION` are now available by default from perl.h, and reflect the base revision, patchlevel, and subversion respectively. `PERL_REVISION` had no prior equivalent, while `PERL_VERSION` and `PERL_SUBVERSION` were previously available as `PATCHLEVEL` and `SUBVERSION`.

The new names cause less pollution of the **cpp** namespace and reflect what the numbers have come to stand for in common practice. For compatibility, the old names are still supported when *patchlevel.h* is explicitly included (as required before), so there is no source incompatibility from the change.

92.12.4 Binary Incompatibilities

In general, the default build of this release is expected to be binary compatible for extensions built with the 5.005 release or its maintenance versions. However, specific platforms may have broken binary compatibility due to changes in the defaults used in hints files. Therefore, please be sure to always check the platform-specific README files for any notes to the contrary.

The usethreads or usemultiplicity builds are **not** binary compatible with the corresponding builds in 5.005.

On platforms that require an explicit list of exports (AIX, OS/2 and Windows, among others), purely internal symbols such as parser functions and the run time opcodes are not exported by default. Perl 5.005 used to export all functions irrespective of whether they were considered part of the public API or not.

For the full list of public API functions, see *perlapi*.

92.13 Known Problems

92.13.1 Thread test failures

The subtests 19 and 20 of lib/thr5005.t test are known to fail due to fundamental problems in the 5.005 threading implementation. These are not new failures–Perl 5.005_0x has the same bugs, but didn't have these tests.

92.13.2 EBCDIC platforms not supported

In earlier releases of Perl, EBCDIC environments like OS390 (also known as Open Edition MVS) and VM-ESA were supported. Due to changes required by the UTF-8 (Unicode) support, the EBCDIC platforms are not supported in Perl 5.6.0.

92.13.3 In 64-bit HP-UX the lib/io_multihomed test may hang

The lib/io_multihomed test may hang in HP-UX if Perl has been configured to be 64-bit. Because other 64-bit platforms do not hang in this test, HP-UX is suspect. All other tests pass in 64-bit HP-UX. The test attempts to create and connect to "multihomed" sockets (sockets which have multiple IP addresses).

92.13.4 NEXTSTEP 3.3 POSIX test failure

In NEXTSTEP 3.3p2 the implementation of the strftime(3) in the operating system libraries is buggy: the %j format numbers the days of a month starting from zero, which, while being logical to programmers, will cause the subtests 19 to 27 of the lib/posix test may fail.

92.13.5 Tru64 (aka Digital UNIX, aka DEC OSF/1) lib/sdbm test failure with gcc

If compiled with gcc 2.95 the lib/sdbm test will fail (dump core). The cure is to use the vendor cc, it comes with the operating system and produces good code.

92.13.6 UNICOS/mk CC failures during Configure run

In UNICOS/mk the following errors may appear during the Configure run:

```
Guessing which symbols your C compiler and preprocessor define...
CC-20 cc: ERROR File = try.c, Line = 3
...
  bad switch yylook 79bad switch yylook 79bad switch yylook 79bad switch yylook 79#ifdef A29K
...
4 errors detected in the compilation of "try.c".
```

The culprit is the broken awk of UNICOS/mk. The effect is fortunately rather mild: Perl itself is not adversely affected by the error, only the h2ph utility coming with Perl, and that is rather rarely needed these days.

92.13.7 Arrow operator and arrays

When the left argument to the arrow operator `->` is an array, or the `scalar` operator operating on an array, the result of the operation must be considered erroneous. For example:

```
    @x->[2]
    scalar(@x)->[2]
```

These expressions will get run-time errors in some future release of Perl.

92.13.8 Experimental features

As discussed above, many features are still experimental. Interfaces and implementation of these features are subject to change, and in extreme cases, even subject to removal in some future release of Perl. These features include the following:

Threads

Unicode

64-bit support

Lvalue subroutines

Weak references

The pseudo-hash data type

The Compiler suite

Internal implementation of file globbing

The DB module

The regular expression code constructs:
`(?{ code })` and `(??{ code })`

92.14 Obsolete Diagnostics

Character class syntax [: : is reserved] **for future extensions**
(W) 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: "\[:" and ":\]".

Ill-formed logical name |%s| in prime_env_iter
(W) A warning peculiar to VMS. A logical name was encountered when preparing to iterate over %ENV which violates the syntactic rules governing logical names. Because it cannot be translated normally, it is skipped, and will not appear in %ENV. This may be a benign occurrence, as some software packages might directly modify logical name tables and introduce nonstandard names, or it may indicate that a logical name table has been corrupted.

In string, @%s now must be written as \@%s
The description of this error used to say:

```
        (Someday it will simply assume that an unbackslashed @
         interpolates an array.)
```

That day has come, and this fatal error has been removed. It has been replaced by a non-fatal warning instead. See Arrays now always interpolate into double-quoted strings for details.

Probable precedence problem on %s

(W) 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;
```

regexp too big

(F) The current implementation of regular expressions uses shorts as address offsets within a string. Unfortunately this means that if the regular expression compiles to longer than 32767, it'll blow up. Usually when you want a regular expression this big, there is a better way to do it with multiple statements. See *perlre*.

Use of "$ $ <digit>" to mean "$ {$ }<digit>" is deprecated

(D) Perl versions before 5.004 misinterpreted any type marker followed by "$" and a digit. For example, "$$0" was incorrectly taken to mean "${$}0" instead of "${$0}". This bug is (mostly) fixed in Perl 5.004.

However, the developers of Perl 5.004 could not fix this bug completely, because at least two widely-used modules depend on the old meaning of "$$0" in a string. So Perl 5.004 still interprets "$$<digit>" in the old (broken) way inside strings; but it generates this message as a warning. And in Perl 5.005, this special treatment will cease.

92.15 Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup. There may also be information at http://www.perl.com/perl/ , the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to perlbug@perl.org to be analysed by the Perl porting team.

92.16 SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

92.17 HISTORY

Written by Gurusamy Sarathy <*gsar@activestate.com*>, with many contributions from The Perl Porters.

Send omissions or corrections to <*perlbug@perl.org*>.

Chapter 93

perl5005delta

What's new for perl5.005

93.1 DESCRIPTION

This document describes differences between the 5.004 release and this one.

93.2 About the new versioning system

Perl is now developed on two tracks: a maintenance track that makes small, safe updates to released production versions with emphasis on compatibility; and a development track that pursues more aggressive evolution. Maintenance releases (which should be considered production quality) have subversion numbers that run from `1` to `49`, and development releases (which should be considered "alpha" quality) run from `50` to `99`.

Perl 5.005 is the combined product of the new dual-track development scheme.

93.3 Incompatible Changes

93.3.1 WARNING: This version is not binary compatible with Perl 5.004.

Starting with Perl 5.004_50 there were many deep and far-reaching changes to the language internals. If you have dynamically loaded extensions that you built under perl 5.003 or 5.004, you can continue to use them with 5.004, but you will need to rebuild and reinstall those extensions to use them 5.005. See *INSTALL* for detailed instructions on how to upgrade.

93.3.2 Default installation structure has changed

The new Configure defaults are designed to allow a smooth upgrade from 5.004 to 5.005, but you should read *INSTALL* for a detailed discussion of the changes in order to adapt them to your system.

93.3.3 Perl Source Compatibility

When none of the experimental features are enabled, there should be very few user-visible Perl source compatibility issues.

If threads are enabled, then some caveats apply. `@_` and `$_` become lexical variables. The effect of this should be largely transparent to the user, but there are some boundary conditions under which user will need to be aware of the issues. For example, `local(@_)` results in a "Can't localize lexical variable @_ ..." message. This may be enabled in a future version.

Some new keywords have been introduced. These are generally expected to have very little impact on compatibility. See New `INIT` keyword, New `lock` keyword, and New `qr//` operator.

Certain barewords are now reserved. Use of these will provoke a warning if you have asked for them with the `-w` switch. See `our` is now a reserved word.

93.3.4 C Source Compatibility

There have been a large number of changes in the internals to support the new features in this release.

- Core sources now require ANSI C compiler

  An ANSI C compiler is now **required** to build perl. See *INSTALL*.

- All Perl global variables must now be referenced with an explicit prefix

  All Perl global variables that are visible for use by extensions now have a `PL_` prefix. New extensions should `not` refer to perl globals by their unqualified names. To preserve sanity, we provide limited backward compatibility for globals that are being widely used like `sv_undef` and `na` (which should now be written as `PL_sv_undef`, `PL_na` etc.)

  If you find that your XS extension does not compile anymore because a perl global is not visible, try adding a `PL_` prefix to the global and rebuild.

  It is strongly recommended that all functions in the Perl API that don't begin with `perl` be referenced with a `Perl_` prefix. The bare function names without the `Perl_` prefix are supported with macros, but this support may cease in a future release.

  See *perlapi*.

- Enabling threads has source compatibility issues

  Perl built with threading enabled requires extensions to use the new `dTHR` macro to initialize the handle to access per-thread data. If you see a compiler error that talks about the variable `thr` not being declared (when building a module that has XS code), you need to add `dTHR;` at the beginning of the block that elicited the error.

  The API function `perl_get_sv("@",FALSE)` should be used instead of directly accessing perl globals as `GvSV(errgv)`. The API call is backward compatible with existing perls and provides source compatibility with threading is enabled.

  See §93.3.4 for more information.

93.3.5 Binary Compatibility

This version is NOT binary compatible with older versions. All extensions will need to be recompiled. Further binaries built with threads enabled are incompatible with binaries built without. This should largely be transparent to the user, as all binary incompatible configurations have their own unique architecture name, and extension binaries get installed at unique locations. This allows coexistence of several configurations in the same directory hierarchy. See *INSTALL*.

93.3.6 Security fixes may affect compatibility

A few taint leaks and taint omissions have been corrected. This may lead to "failure" of scripts that used to work with older versions. Compiling with -DINCOMPLETE_TAINTS provides a perl with minimal amounts of changes to the tainting behavior. But note that the resulting perl will have known insecurities.

Oneliners with the `-e` switch do not create temporary files anymore.

93.3.7 Relaxed new mandatory warnings introduced in 5.004

Many new warnings that were introduced in 5.004 have been made optional. Some of these warnings are still present, but perl's new features make them less often a problem. See New Diagnostics.

93.3.8 Licensing

Perl has a new Social Contract for contributors. See *Porting/Contract*.

The license included in much of the Perl documentation has changed. Most of the Perl documentation was previously under the implicit GNU General Public License or the Artistic License (at the user's choice). Now much of the documentation unambiguously states the terms under which it may be distributed. Those terms are in general much less restrictive than the GNU GPL. See *perl* and the individual perl manpages listed therein.

93.4 Core Changes

93.4.1 Threads

WARNING: Threading is considered an **experimental** feature. Details of the implementation may change without notice. There are known limitations and some bugs. These are expected to be fixed in future versions.

See *README.threads*.

93.4.2 Compiler

WARNING: The Compiler and related tools are considered **experimental**. Features may change without notice, and there are known limitations and bugs. Since the compiler is fully external to perl, the default configuration will build and install it.

The Compiler produces three different types of transformations of a perl program. The C backend generates C code that captures perl's state just before execution begins. It eliminates the compile-time overheads of the regular perl interpreter, but the run-time performance remains comparatively the same. The CC backend generates optimized C code equivalent to the code path at run-time. The CC backend has greater potential for big optimizations, but only a few optimizations are implemented currently. The Bytecode backend generates a platform independent bytecode representation of the interpreter's state just before execution. Thus, the Bytecode back end also eliminates much of the compilation overhead of the interpreter.

The compiler comes with several valuable utilities.

`B::Lint` is an experimental module to detect and warn about suspicious code, especially the cases that the `-w` switch does not detect.

`B::Deparse` can be used to demystify perl code, and understand how perl optimizes certain constructs.

`B::Xref` generates cross reference reports of all definition and use of variables, subroutines and formats in a program.

`B::Showlex` show the lexical variables used by a subroutine or file at a glance.

`perlcc` is a simple frontend for compiling perl.

See `ext/B/README`, *B*, and the respective compiler modules.

93.4.3 Regular Expressions

Perl's regular expression engine has been seriously overhauled, and many new constructs are supported. Several bugs have been fixed.

Here is an itemized summary:

Many new and improved optimizations

Changes in the RE engine:

```
Unneeded nodes removed;
Substrings merged together;
New types of nodes to process (SUBEXPR)* and similar expressions
    quickly, used if the SUBEXPR has no side effects and matches
    strings of the same length;
Better optimizations by lookup for constant substrings;
Better search for constants substrings anchored by $ ;
```

Changes in Perl code using RE engine:

```
More optimizations to s/longer/short/;
study() was not working;
/blah/ may be optimized to an analogue of index() if $& $‘ $’ not seen;
Unneeded copying of matched-against string removed;
Only matched part of the string is copying if $‘ $’ were not seen;
```

Many bug fixes

Note that only the major bug fixes are listed here. See *Changes* for others.

```
Backtracking might not restore start of $3.
No feedback if max count for * or + on "complex" subexpression
    was reached, similarly (but at compile time) for {3,34567}
Primitive restrictions on max count introduced to decrease a
    possibility of a segfault;
(ZERO-LENGTH)* could segfault;
(ZERO-LENGTH)* was prohibited;
Long REs were not allowed;
/RE/g could skip matches at the same position after a
  zero-length match;
```

New regular expression constructs

The following new syntax elements are supported:

```
(?<=RE)
(?<!RE)
(?{ CODE })
(?i-x)
(?i:RE)
(?(COND)YES_RE|NO_RE)
(?>RE)
\z
```

New operator for precompiled regular expressions

See New `qr//` operator.

Other improvements

```
Better debugging output (possibly with colors),
    even from non-debugging Perl;
RE engine code now looks like C, not like assembler;
Behaviour of RE modifiable by ‘use re’ directive;
Improved documentation;
Test suite significantly extended;
Syntax [:^upper:] etc., reserved inside character classes;
```

Incompatible changes

```
(?i) localized inside enclosing group;
$( is not interpolated into RE any more;
/RE/g may match at the same position (with non-zero length)
    after a zero-length match (bug fix).
```

See *perlre* and *perlop*.

93.4.4 Improved malloc()

See banner at the beginning of `malloc.c` for details.

93.4.5 Quicksort is internally implemented

Perl now contains its own highly optimized qsort() routine. The new qsort() is resistant to inconsistent comparison functions, so Perl's `sort()` will not provoke coredumps any more when given poorly written sort subroutines. (Some C library `qsort()`s that were being used before used to have this problem.) In our testing, the new `qsort()` required the minimal number of pair-wise compares on average, among all known `qsort()` implementations.

See `perlfunc/sort`.

93.4.6 Reliable signals

Perl's signal handling is susceptible to random crashes, because signals arrive asynchronously, and the Perl runtime is not reentrant at arbitrary times.

However, one experimental implementation of reliable signals is available when threads are enabled. See `Thread::Signal`. Also see *INSTALL* for how to build a Perl capable of threads.

93.4.7 Reliable stack pointers

The internals now reallocate the perl stack only at predictable times. In particular, magic calls never trigger reallocations of the stack, because all reentrancy of the runtime is handled using a "stack of stacks". This should improve reliability of cached stack pointers in the internals and in XSUBs.

93.4.8 More generous treatment of carriage returns

Perl used to complain if it encountered literal carriage returns in scripts. Now they are mostly treated like whitespace within program text. Inside string literals and here documents, literal carriage returns are ignored if they occur paired with linefeeds, or get interpreted as whitespace if they stand alone. This behavior means that literal carriage returns in files should be avoided. You can get the older, more compatible (but less generous) behavior by defining the preprocessor symbol `PERL_STRICT_CR` when building perl. Of course, all this has nothing whatever to do with how escapes like `\r` are handled within strings.

Note that this doesn't somehow magically allow you to keep all text files in DOS format. The generous treatment only applies to files that perl itself parses. If your C compiler doesn't allow carriage returns in files, you may still be unable to build modules that need a C compiler.

93.4.9 Memory leaks

`substr`, `pos` and `vec` don't leak memory anymore when used in lvalue context. Many small leaks that impacted applications that embed multiple interpreters have been fixed.

93.4.10 Better support for multiple interpreters

The build-time option `-DMULTIPLICITY` has had many of the details reworked. Some previously global variables that should have been per-interpreter now are. With care, this allows interpreters to call each other. See the `PerlInterp` extension on CPAN.

93.4.11 Behavior of local() on array and hash elements is now well-defined

See Temporary Values via local() in *perlsub*.

93.4.12 `%!` is transparently tied to the *Errno* module

See *perlvar*, and *Errno*.

93.4.13 Pseudo-hashes are supported

See *perlref*.

93.4.14 `EXPR foreach EXPR` is supported

See *perlsyn*.

93.4.15 Keywords can be globally overridden

See *perlsub*.

93.4.16 `$^E` is meaningful on Win32

See *perlvar*.

93.4.17 `foreach (1..1000000)` optimized

`foreach (1..1000000)` is now optimized into a counting loop. It does not try to allocate a 1000000-size list anymore.

93.4.18 `Foo::` can be used as implicitly quoted package name

Barewords caused unintuitive behavior when a subroutine with the same name as a package happened to be defined. Thus, `new Foo @args`, use the result of the call to `Foo()` instead of `Foo` being treated as a literal. The recommended way to write barewords in the indirect object slot is `new Foo:: @args`. Note that the method `new()` is called with a first argument of `Foo`, not `Foo::` when you do that.

93.4.19 `exists $Foo::{Bar::}` tests existence of a package

It was impossible to test for the existence of a package without actually creating it before. Now `exists $Foo::{Bar::}` can be used to test if the `Foo::Bar` namespace has been created.

93.4.20 Better locale support

See *perllocale*.

93.4.21 Experimental support for 64-bit platforms

Perl5 has always had 64-bit support on systems with 64-bit longs. Starting with 5.005, the beginnings of experimental support for systems with 32-bit long and 64-bit 'long long' integers has been added. If you add -DUSE_LONG_LONG to your ccflags in config.sh (or manually define it in perl.h) then perl will be built with 'long long' support. There will be many compiler warnings, and the resultant perl may not work on all systems. There are many other issues related to third-party extensions and libraries. This option exists to allow people to work on those issues.

93.4.22 prototype() returns useful results on builtins

See prototype in *perlfunc*.

93.4.23 Extended support for exception handling

`die()` now accepts a reference value, and `$@` gets set to that value in exception traps. This makes it possible to propagate exception objects. This is an undocumented **experimental** feature.

93.4.24 Re-blessing in DESTROY() supported for chaining DESTROY() methods

See Destructors in *perlobj*.

93.4.25 All `printf` format conversions are handled internally

See printf in *perlfunc*.

93.4.26 New `INIT` keyword

`INIT` subs are like `BEGIN` and `END`, but they get run just before the perl runtime begins execution. e.g., the Perl Compiler makes use of `INIT` blocks to initialize and resolve pointers to XSUBs.

93.4.27 New `lock` keyword

The `lock` keyword is the fundamental synchronization primitive in threaded perl. When threads are not enabled, it is currently a noop.

To minimize impact on source compatibility this keyword is "weak", i.e., any user-defined subroutine of the same name overrides it, unless a `use Thread` has been seen.

93.4.28 New `qr//` operator

The `qr//` operator, which is syntactically similar to the other quote-like operators, is used to create precompiled regular expressions. This compiled form can now be explicitly passed around in variables, and interpolated in other regular expressions. See *perlop*.

93.4.29 `our` is now a reserved word

Calling a subroutine with the name `our` will now provoke a warning when using the `-w` switch.

93.4.30 Tied arrays are now fully supported

See *Tie::Array*.

93.4.31 Tied handles support is better

Several missing hooks have been added. There is also a new base class for TIEARRAY implementations. See *Tie::Array*.

93.4.32 4th argument to substr

substr() can now both return and replace in one operation. The optional 4th argument is the replacement string. See substr in *perlfunc*.

93.4.33 Negative LENGTH argument to splice

splice() with a negative LENGTH argument now work similar to what the LENGTH did for substr(). Previously a negative LENGTH was treated as 0. See splice in *perlfunc*.

93.4.34 Magic lvalues are now more magical

When you say something like `substr($x, 5) = "hi"`, the scalar returned by substr() is special, in that any modifications to it affect $x. (This is called a 'magic lvalue' because an 'lvalue' is something on the left side of an assignment.) Normally, this is exactly what you would expect to happen, but Perl uses the same magic if you use substr(), pos(), or vec() in a context where they might be modified, like taking a reference with \ or as an argument to a sub that modifies `@_`. In previous versions, this 'magic' only went one way, but now changes to the scalar the magic refers to ($x in the above example) affect the magic lvalue too. For instance, this code now acts differently:

```
    $x = "hello";
    sub printit {
        $x = "g'bye";
        print $_[0], "\n";
    }
    printit(substr($x, 0, 5));
```

In previous versions, this would print "hello", but it now prints "g'bye".

93.4.35 <> now reads in records

If `$/` is a reference to an integer, or a scalar that holds an integer, <> will read in records instead of lines. For more info, see $/ in *perlvar*.

93.5 Supported Platforms

Configure has many incremental improvements. Site-wide policy for building perl can now be made persistent, via Policy.sh. Configure also records the command-line arguments used in *config.sh*.

93.5.1 New Platforms

BeOS is now supported. See *README.beos*.

DOS is now supported under the DJGPP tools. See *README.dos* (installed as *perldos* on some systems).

MiNT is now supported. See *README.mint*.

MPE/iX is now supported. See *README.mpeix*.

MVS (aka OS390, aka Open Edition) is now supported. See *README.os390* (installed as *perlos390* on some systems).

Stratus VOS is now supported. See *README.vos*.

93.5.2 Changes in existing support

Win32 support has been vastly enhanced. Support for Perl Object, a C++ encapsulation of Perl. GCC and EGCS are now supported on Win32. See *README.win32*, aka *perlwin32*.

VMS configuration system has been rewritten. See *README.vms* (installed as README_vms on some systems).

The hints files for most Unix platforms have seen incremental improvements.

93.6 Modules and Pragmata

93.6.1 New Modules

B

Perl compiler and tools. See *B*.

Data::Dumper

A module to pretty print Perl data. See *Data::Dumper*.

Dumpvalue

A module to dump perl values to the screen. See *Dumpvalue*.

Errno

A module to look up errors more conveniently. See *Errno*.

File::Spec

A portable API for file operations.

ExtUtils::Installed

Query and manage installed modules.

ExtUtils::Packlist

Manipulate .packlist files.

Fatal

Make functions/builtins succeed or die.

IPC::SysV

Constants and other support infrastructure for System V IPC operations in perl.

Test

A framework for writing testsuites.

Tie::Array

Base class for tied arrays.

Tie::Handle

Base class for tied handles.

Thread

Perl thread creation, manipulation, and support.

attrs

Set subroutine attributes.

fields

Compile-time class fields.

re

Various pragmata to control behavior of regular expressions.

93.6.2 Changes in existing modules

Benchmark

You can now run tests for *x* seconds instead of guessing the right number of tests to run.

Keeps better time.

Carp

Carp has a new function cluck(). cluck() warns, like carp(), but also adds a stack backtrace to the error message, like confess().

CGI

CGI has been updated to version 2.42.

Fcntl

More Fcntl constants added: F_SETLK64, F_SETLKW64, O_LARGEFILE for large (more than 4G) file access (the 64-bit support is not yet working, though, so no need to get overly excited), Free/Net/OpenBSD locking behaviour flags F_FLOCK, F_POSIX, Linux F_SHLCK, and O_ACCMODE: the mask of O_RDONLY, O_WRONLY, and O_RDWR.

Math::Complex

The accessors methods Re, Im, arg, abs, rho, theta, methods can ($z->Re()) now also act as mutators ($z->Re(3)).

Math::Trig

A little bit of radial trigonometry (cylindrical and spherical) added, for example the great circle distance.

POSIX

POSIX now has its own platform-specific hints files.

DB_File

DB_File supports version 2.x of Berkeley DB. See `ext/DB_File/Changes`.

MakeMaker

MakeMaker now supports writing empty makefiles, provides a way to specify that site umask() policy should be honored. There is also better support for manipulation of .packlist files, and getting information about installed modules.

Extensions that have both architecture-dependent and architecture-independent files are now always installed completely in the architecture-dependent locations. Previously, the shareable parts were shared both across architectures and across perl versions and were therefore liable to be overwritten with newer versions that might have subtle incompatibilities.

CPAN

See *perlmodinstall* and *CPAN*.

Cwd

Cwd::cwd is faster on most platforms.

93.7 Utility Changes

`h2ph` and related utilities have been vastly overhauled.

`perlcc`, a new experimental front end for the compiler is available.

The crude GNU `configure` emulator is now called `configure.gnu` to avoid trampling on `Configure` under case-insensitive filesystems.

`perldoc` used to be rather slow. The slower features are now optional. In particular, case-insensitive searches need the `-i` switch, and recursive searches need `-r`. You can set these switches in the `PERLDOC` environment variable to get the old behavior.

93.8 Documentation Changes

Config.pm now has a glossary of variables.

Porting/patching.pod has detailed instructions on how to create and submit patches for perl.

perlport specifies guidelines on how to write portably.

perlmodinstall describes how to fetch and install modules from `CPAN` sites.

Some more Perl traps are documented now. See *perltrap*.

perlopentut gives a tutorial on using open().

perlreftut gives a tutorial on references.

perlthrtut gives a tutorial on threads.

93.9 New Diagnostics

Ambiguous call resolved as CORE::%s(), qualify as such or use &

(W) 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 by declaring the subroutine to be an object method (see *attrs*).

Bad index while coercing array into hash

(F) The index looked up in the hash found as the 0'th element of a pseudo-hash is not legal. Index values must be at 1 or greater. See *perlref*.

Bareword "%s" refers to nonexistent package

(W) 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?

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 = 42;
process $BADREF 1,2,3;
$BADREF->process(1,2,3);
```

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 array into hash

(F) You used an array where a hash was expected, but the array has no information on how to map from keys to array indices. You can do that only with arrays that have a hash reference at index 0.

Can't goto subroutine from an eval-string

(F) The "goto subroutine" call can't be used to jump out of an eval "string". (You can use it to jump out of an eval {BLOCK}, but you probably don't want to.)

Can't localize pseudo-hash element

(F) You said something like `local $ar->{'key'}`, where $ar is a reference to a pseudo-hash. That hasn't been implemented yet, but you can get a similar effect by localizing the corresponding array element directly – `local $ar->[$ar->[0]{'key'}]`.

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.

Cannot 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`.

Character class syntax [. . is reserved] for future extensions

(W) 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: "\[." and ".\]".

Character class syntax [: : is reserved] for future extensions

(W) 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: "\[:" and ":\]".

Character class syntax [= = is reserved] for future extensions

(W) 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: "\[=" and "=\]".

%s: 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 (?{ code }) in *perlre*, and *perlsec*.

%s: Eval-group not allowed, use re 'eval'

(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 (?{ code }) in *perlre*.

%s: Eval-group not allowed at run time

(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 explicitly building the pattern from an interpolated string at run time and using that in an eval(). See (?{ code }) in *perlre*.

Explicit blessing to " (assuming package main)

(W) 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');

Illegal hex digit ignored

(W) You may have tried to use a character other than 0 - 9 or A - F in a hexadecimal number. Interpretation of the hexadecimal number stopped before the illegal character.

No such array field

(F) You tried to access an array as a hash, but the field name used is not defined. The hash at index 0 should map all valid field names to array indices for that to work.

No such field "%s" in variable %s of type %s

(F) You tried to access a field of a typed variable where the type does not know about the field name. The field names are looked up in the %FIELDS hash in the type package at compile time. The %FIELDS hash is usually set up with the 'fields' pragma.

Out of memory during ridiculously large request

(F) You can't allocate more than 2ˆ31+"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]`.

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.

Recursive inheritance detected while looking for method '%s' %s

(F) More than 100 levels of inheritance were encountered while invoking a method. Probably indicates an unintended loop in your inheritance hierarchy.

Reference found where even-sized list expected

(W) 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
```

Undefined value assigned to typeglob

(W) An undefined value was assigned to a typeglob, a la `*foo = undef`. This does nothing. It's possible that you really mean `undef *foo`.

Use of reserved word "%s" is deprecated

(D) The indicated bareword is a reserved word. Future versions of perl may use it as a keyword, so you're better off either explicitly quoting the word in a manner appropriate for its context of use, or using a different name altogether. The warning can be suppressed for subroutine names by either adding a & prefix, or using a package qualifier, e.g. `&our()`, or `Foo::our()`.

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 system administrator have set up the so-called variable 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, 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 LOCALE PROBLEMS in *perllocale*.

93.10 Obsolete Diagnostics

Can't mktemp()

(F) The mktemp() routine failed for some reason while trying to process a **-e** switch. Maybe your /tmp partition is full, or clobbered.

Removed because **-e** doesn't use temporary files any more.

Can't write to temp file for -e: %s

(F) The write routine failed for some reason while trying to process a **-e** switch. Maybe your /tmp partition is full, or clobbered.

Removed because **-e** doesn't use temporary files any more.

Cannot open temporary file

(F) The create routine failed for some reason while trying to process a **-e** switch. Maybe your /tmp partition is full, or clobbered.

Removed because **-e** doesn't use temporary files any more.

regexp too big

(F) The current implementation of regular expressions uses shorts as address offsets within a string. Unfortunately this means that if the regular expression compiles to longer than 32767, it'll blow up. Usually when you want a regular expression this big, there is a better way to do it with multiple statements. See *perlre*.

93.11 Configuration Changes

You can use "Configure -Uinstallusrbinperl" which causes installperl to skip installing perl also as /usr/bin/perl. This is useful if you prefer not to modify /usr/bin for some reason or another but harmful because many scripts assume to find Perl in /usr/bin/perl.

93.12 BUGS

If you find what you think is a bug, you might check the headers of recently posted articles in the comp.lang.perl.misc newsgroup. There may also be information at http://www.perl.com/perl/ , the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Make sure you trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to <*perlbug@perl.com*> to be analysed by the Perl porting team.

93.13 SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

93.14 HISTORY

Written by Gurusamy Sarathy <*gsar@activestate.com*>, with many contributions from The Perl Porters.

Send omissions or corrections to <*perlbug@perl.com*>.

Chapter 94

perl5004delta

What's new for perl5.004

94.1 DESCRIPTION

This document describes differences between the 5.003 release (as documented in *Programming Perl*, second edition–the Camel Book) and this one.

94.2 Supported Environments

Perl5.004 builds out of the box on Unix, Plan 9, LynxOS, VMS, OS/2, QNX, AmigaOS, and Windows NT. Perl runs on Windows 95 as well, but it cannot be built there, for lack of a reasonable command interpreter.

94.3 Core Changes

Most importantly, many bugs were fixed, including several security problems. See the *Changes* file in the distribution for details.

94.3.1 List assignment to %ENV works

`%ENV = ()` and `%ENV = @list` now work as expected (except on VMS where it generates a fatal error).

94.3.2 Change to "Can't locate Foo.pm in @INC" error

The error "Can't locate Foo.pm in @INC" now lists the contents of @INC for easier debugging.

94.3.3 Compilation option: Binary compatibility with 5.003

There is a new Configure question that asks if you want to maintain binary compatibility with Perl 5.003. If you choose binary compatibility, you do not have to recompile your extensions, but you might have symbol conflicts if you embed Perl in another application, just as in the 5.003 release. By default, binary compatibility is preserved at the expense of symbol table pollution.

94.3.4 $ PERL5OPT environment variable

You may now put Perl options in the $PERL5OPT environment variable. Unless Perl is running with taint checks, it will interpret this variable as if its contents had appeared on a "#!perl" line at the beginning of your script, except that hyphens are optional. PERL5OPT may only be used to set the following switches: **-[DIMUdmw]**.

94.3.5 Limitations on -M, -m, and -T options

The `-M` and `-m` options are no longer allowed on the `#!` line of a script. If a script needs a module, it should invoke it with the `use` pragma.

The **-T** option is also forbidden on the `#!` line of a script, unless it was present on the Perl command line. Due to the way `#!` works, this usually means that **-T** must be in the first argument. Thus:

```
#!/usr/bin/perl -T -w
```

will probably work for an executable script invoked as `scriptname`, while:

```
#!/usr/bin/perl -w -T
```

will probably fail under the same conditions. (Non-Unix systems will probably not follow this rule.) But `perl scriptname` is guaranteed to fail, since then there is no chance of **-T** being found on the command line before it is found on the `#!` line.

94.3.6 More precise warnings

If you removed the **-w** option from your Perl 5.003 scripts because it made Perl too verbose, we recommend that you try putting it back when you upgrade to Perl 5.004. Each new perl version tends to remove some undesirable warnings, while adding new warnings that may catch bugs in your scripts.

94.3.7 Deprecated: Inherited `AUTOLOAD` for non-methods

Before Perl 5.004, `AUTOLOAD` functions were looked up as methods (using the `@ISA` hierarchy), even when the function to be autoloaded was called as a plain function (e.g. `Foo::bar()`), not a method (e.g. `Foo->bar()` or `$obj->bar()`).

Perl 5.005 will use 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 5.004 issues an optional warning when a non-method uses an inherited `AUTOLOAD`.

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.

94.3.8 Previously deprecated %OVERLOAD is no longer usable

Using %OVERLOAD to define overloading was deprecated in 5.003. Overloading is now defined using the overload pragma. %OVERLOAD is still used internally but should not be used by Perl scripts. See *overload* for more details.

94.3.9 Subroutine arguments created only when they're modified

In Perl 5.004, nonexistent array and hash elements used as subroutine parameters are brought into existence only if they are actually assigned to (via `@_`).

Earlier versions of Perl vary in their handling of such arguments. Perl versions 5.002 and 5.003 always brought them into existence. Perl versions 5.000 and 5.001 brought them into existence only if they were not the first argument (which was almost certainly a bug). Earlier versions of Perl never brought them into existence.

For example, given this code:

```
undef @a; undef %a;
sub show { print $_[0] };
sub change { $_[0]++ };
show($a[2]);
change($a{b});
```

After this code executes in Perl 5.004, $a{b} exists but $a[2] does not. In Perl 5.002 and 5.003, both $a{b} and $a[2] would have existed (but $a[2]'s value would have been undefined).

94.3.10 Group vector changeable with `$)`

The `$)` special variable has always (well, in Perl 5, at least) reflected not only the current effective group, but also the group list as returned by the `getgroups()` C function (if there is one). However, until this release, there has not been a way to call the `setgroups()` C function from Perl.

In Perl 5.004, assigning to `$)` is exactly symmetrical with examining it: The first number in its string value is used as the effective gid; if there are any numbers after the first one, they are passed to the `setgroups()` C function (if there is one).

94.3.11 Fixed parsing of $ $ <digit>, &$ <digit>, etc.

Perl versions before 5.004 misinterpreted any type marker followed by "$" and a digit. For example, "$$0" was incorrectly taken to mean "${$}0" instead of "${$0}". This bug is (mostly) fixed in Perl 5.004.

However, the developers of Perl 5.004 could not fix this bug completely, because at least two widely-used modules depend on the old meaning of "$$0" in a string. So Perl 5.004 still interprets "$$<digit>" in the old (broken) way inside strings; but it generates this message as a warning. And in Perl 5.005, this special treatment will cease.

94.3.12 Fixed localization of $ <digit>, $ &, etc.

Perl versions before 5.004 did not always properly localize the regex-related special variables. Perl 5.004 does localize them, as the documentation has always said it should. This may result in $1, $2, etc. no longer being set where existing programs use them.

94.3.13 No resetting of $. on implicit close

The documentation for Perl 5.0 has always stated that `$.` is *not* reset when an already-open file handle is reopened with no intervening call to `close`. Due to a bug, perl versions 5.000 through 5.003 *did* reset `$.` under that circumstance; Perl 5.004 does not.

94.3.14 `wantarray` may return undef

The `wantarray` operator returns true if a subroutine is expected to return a list, and false otherwise. In Perl 5.004, `wantarray` can also return the undefined value if a subroutine's return value will not be used at all, which allows subroutines to avoid a time-consuming calculation of a return value if it isn't going to be used.

94.3.15 `eval EXPR` determines value of EXPR in scalar context

Perl (version 5) used to determine the value of EXPR inconsistently, sometimes incorrectly using the surrounding context for the determination. Now, the value of EXPR (before being parsed by eval) is always determined in a scalar context. Once parsed, it is executed as before, by providing the context that the scope surrounding the eval provided. This change makes the behavior Perl4 compatible, besides fixing bugs resulting from the inconsistent behavior. This program:

```
    @a = qw(time now is time);
    print eval @a;
    print '|', scalar eval @a;
```

used to print something like "timenowis881399109|4", but now (and in perl4) prints "4|4".

94.3.16 Changes to tainting checks

A bug in previous versions may have failed to detect some insecure conditions when taint checks are turned on. (Taint checks are used in setuid or setgid scripts, or when explicitly turned on with the `-T` invocation option.) Although it's unlikely, this may cause a previously-working script to now fail – which should be construed as a blessing, since that indicates a potentially-serious security hole was just plugged.

The new restrictions when tainting include:

No glob() or <*>

These operators may spawn the C shell (csh), which cannot be made safe. This restriction will be lifted in a future version of Perl when globbing is implemented without the use of an external program.

No spawning if tainted $ CDPATH, $ ENV, $ BASH_ENV

These environment variables may alter the behavior of spawned programs (especially shells) in ways that subvert security. So now they are treated as dangerous, in the manner of $IFS and $PATH.

No spawning if tainted $ TERM doesn't look like a terminal name

Some termcap libraries do unsafe things with $TERM. However, it would be unnecessarily harsh to treat all $TERM values as unsafe, since only shell metacharacters can cause trouble in $TERM. So a tainted $TERM is considered to be safe if it contains only alphanumerics, underscores, dashes, and colons, and unsafe if it contains other characters (including whitespace).

94.3.17 New Opcode module and revised Safe module

A new Opcode module supports the creation, manipulation and application of opcode masks. The revised Safe module has a new API and is implemented using the new Opcode module. Please read the new Opcode and Safe documentation.

94.3.18 Embedding improvements

In older versions of Perl it was not possible to create more than one Perl interpreter instance inside a single process without leaking like a sieve and/or crashing. The bugs that caused this behavior have all been fixed. However, you still must take care when embedding Perl in a C program. See the updated perlembed manpage for tips on how to manage your interpreters.

94.3.19 Internal change: FileHandle class based on IO::* classes

File handles are now stored internally as type IO::Handle. The FileHandle module is still supported for backwards compatibility, but it is now merely a front end to the IO::* modules – specifically, IO::Handle, IO::Seekable, and IO::File. We suggest, but do not require, that you use the IO::* modules in new code.

In harmony with this change, `*GLOB{FILEHANDLE}` is now just a backward-compatible synonym for `*GLOB{IO}`.

94.3.20 Internal change: PerlIO abstraction interface

It is now possible to build Perl with AT&T's sfio IO package instead of stdio. See *perlapio* for more details, and the *INSTALL* file for how to use it.

94.3.21 New and changed syntax

$ coderef->(PARAMS)

A subroutine reference may now be suffixed with an arrow and a (possibly empty) parameter list. This syntax denotes a call of the referenced subroutine, with the given parameters (if any).

This new syntax follows the pattern of `$hashref->{FOO}` and `$aryref->[$foo]`: You may now write `&$subref($foo)` as `$subref->($foo)`. All these arrow terms may be chained; thus, `&{$table->{FOO}}($bar)` may now be written `$table->{FOO}->($bar)`.

94.3.22 New and changed builtin constants

__PACKAGE__

The current package name at compile time, or the undefined value if there is no current package (due to a `package;` directive). Like `__FILE__` and `__LINE__`, `__PACKAGE__` does *not* interpolate into strings.

94.3.23 New and changed builtin variables

$ ˆE

Extended error message on some platforms. (Also known as $EXTENDED_OS_ERROR if you `use English`).

$ ˆH

The current set of syntax checks enabled by `use strict`. See the documentation of `strict` for more details. Not actually new, but newly documented. Because it is intended for internal use by Perl core components, there is no `use English` long name for this variable.

$ ˆM

By default, running out of memory it is not trappable. However, if compiled for this, Perl may use the contents of `$ˆM` as an emergency pool after die()ing with this message. 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 when in emergency. See the *INSTALL* file for information on how to enable this option. As a disincentive to casual use of this advanced feature, there is no `use English` long name for this variable.

94.3.24 New and changed builtin functions

delete on slices

This now works. (e.g. `delete @ENV{'PATH', 'MANPATH'}`)

flock

is now supported on more platforms, prefers fcntl to lockf when emulating, and always flushes before (un)locking.

printf and sprintf

Perl now implements these functions itself; it doesn't use the C library function sprintf() any more, except for floating-point numbers, and even then only known flags are allowed. As a result, it is now possible to know which conversions and flags will work, and what they will do.

The new conversions in Perl's sprintf() are:

```
   %i   a synonym for %d
   %p   a pointer (the address of the Perl value, in hexadecimal)
   %n   special: *stores* the number of characters output so far
        into the next variable in the parameter list
```

The new flags that go between the `%` and the conversion are:

```
   #    prefix octal with "0", hex with "0x"
   h    interpret integer as C type "short" or "unsigned short"
   V    interpret integer as Perl's standard integer type
```

Also, where a number would appear in the flags, an asterisk ("*") may be used instead, in which case Perl uses the next item in the parameter list as the given number (that is, as the field width or precision). If a field width obtained through "*" is negative, it has the same effect as the '-' flag: left-justification.

See sprintf in *perlfunc* for a complete list of conversion and flags.

keys as an lvalue

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. 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).

my() in Control Structures

You can now use my() (with or without the parentheses) in the control expressions of control structures such as:

```
while (defined(my $line = <>)) {
    $line = lc $line;
} continue {
    print $line;
}

if ((my $answer = <STDIN>) =~ /^y(es)?$/i) {
    user_agrees();
} elsif ($answer =~ /^n(o)?$/i) {
    user_disagrees();
} else {
    chomp $answer;
    die "'$answer' is neither 'yes' nor 'no'";
}
```

Also, you can declare a foreach loop control variable as lexical by preceding it with the word "my". For example, in:

```
foreach my $i (1, 2, 3) {
    some_function();
}
```

$i is a lexical variable, and the scope of $i extends to the end of the loop, but not beyond it.

Note that you still cannot use my() on global punctuation variables such as $_ and the like.

pack() and unpack()

A new format 'w' represents a BER compressed integer (as defined in ASN.1). Its format is a sequence of one or more bytes, each of which provides seven bits of the total value, with the most significant first. Bit eight of each byte is set, except for the last byte, in which bit eight is clear.

If 'p' or 'P' are given undef as values, they now generate a NULL pointer.

Both pack() and unpack() now fail when their templates contain invalid types. (Invalid types used to be ignored.)

sysseek()

The new sysseek() operator is a variant of seek() that sets and gets the file's system read/write position, using the lseek(2) system call. It is the only reliable way to seek before using sysread() or syswrite(). Its return value is the new position, or the undefined value on failure.

use VERSION

If the first argument to `use` is a number, it is treated as a version number instead of a module name. If the version of the Perl interpreter is less than VERSION, then an error message is printed and Perl exits immediately. Because `use` occurs at compile time, this check happens immediately during the compilation process, unlike `require VERSION`, which waits until runtime for the check. This is often useful if you need to check the current Perl version before `use`ing library modules which have changed in incompatible ways from older versions of Perl. (We try not to do this more than we have to.)

use Module VERSION LIST

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. (Note that there is not a comma after VERSION!)

This version-checking mechanism is similar to the one currently used in the Exporter module, but it is faster and can be used with modules that don't use the Exporter. It is the recommended method for new code.

prototype(FUNCTION)

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. (Not actually new; just never documented before.)

srand

The default seed for `srand`, which used to be `time`, has been changed. Now it's a heady mix of difficult-to-predict system-dependent values, which should be sufficient for most everyday purposes.

Previous to version 5.004, calling `rand` without first calling `srand` would yield the same sequence of random numbers on most or all machines. Now, when perl sees that you're calling `rand` and haven't yet called `srand`, it calls `srand` with the default seed. You should still call `srand` manually if your code might ever be run on a pre-5.004 system, of course, or if you want a seed other than the default.

$ _ as Default

Functions documented in the Camel to default to $_ now in fact do, and all those that do are so documented in *perlfunc*.

`m//gc` does not reset search position on failure

The `m//g` match iteration construct has always reset its target string's search position (which is visible through the `pos` operator) when a match fails; as a result, the next `m//g` match after a failure starts again at the beginning of the string. With Perl 5.004, this reset may be disabled by adding the "c" (for "continue") modifier, i.e. `m//gc`. This feature, in conjunction with the `\G` zero-width assertion, makes it possible to chain matches together. See *perlop* and *perlre*.

`m//x` ignores whitespace before ?*+{}

The `m//x` construct has always been intended to ignore all unescaped whitespace. However, before Perl 5.004, whitespace had the effect of escaping repeat modifiers like "*" or "?"; for example, `/a *b/x` was (mis)interpreted as `/a\*b/x`. This bug has been fixed in 5.004.

nested `sub{}` closures work now

Prior to the 5.004 release, nested anonymous functions didn't work right. They do now.

formats work right on changing lexicals

Just like anonymous functions that contain lexical variables that change (like a lexical index variable for a `foreach` loop), formats now work properly. For example, this silently failed before (printed only zeros), but is fine now:

```
    my $i;
    foreach $i ( 1 .. 10 ) {
        write;
    }
    format =
        my i is @#
        $i
    .
```

However, it still fails (without a warning) if the foreach is within a subroutine:

```
    my $i;
    sub foo {
      foreach $i ( 1 .. 10 ) {
        write;
      }
    }
    foo;
    format =
        my i is @#
        $i
    .
```

94.3.25 New builtin methods

The `UNIVERSAL` package automatically contains the following methods that are inherited by all other classes:

isa(CLASS)

`isa` returns *true* if its object is blessed into a subclass of `CLASS`

`isa` is also exportable and can be called as a sub with two arguments. This allows the ability to check what a reference points to. Example:

```
    use UNIVERSAL qw(isa);

    if(isa($ref, 'ARRAY')) {
       ...
    }
```

can(METHOD)

`can` checks to see if its object has a method called `METHOD`, if it does then a reference to the sub is returned; if it does not then *undef* is returned.

VERSION([NEED)]

`VERSION` 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 given package) not less than NEED; it will die if this is not the case. This method is normally called as a class method. This method is called automatically by the `VERSION` form of `use`.

```
    use A 1.2 qw(some imported subs);
    # implies:
    A->VERSION(1.2);
```

NOTE: `can` directly uses Perl's internal code for method lookup, and `isa` uses a very similar method and caching strategy. This may cause strange effects if the Perl code dynamically changes @ISA in any package.

You may add other methods to the UNIVERSAL class via Perl or XS code. You do not need to `use UNIVERSAL` in order to make these methods available to your program. This is necessary only if you wish to have `isa` available as a plain subroutine in the current package.

94.3.26 TIEHANDLE now supported

See *perltie* for other kinds of tie()s.

TIEHANDLE classname, LIST

This is the constructor for the class. That means it is expected to return an object of some sort. The reference can be used to hold some internal information.

```
sub TIEHANDLE {
    print "<shout>\n";
    my $i;
    return bless \$i, shift;
}
```

PRINT this, LIST

This method will be triggered every time the tied handle is printed to. Beyond its self reference it also expects the list that was passed to the print function.

```
sub PRINT {
    $r = shift;
    $$r++;
    return print join( $, => map {uc} @_), $\;
}
```

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, @_)."\n";
}
```

READ this LIST

This method will be called when the handle is read from via the `read` or `sysread` functions.

```
sub READ {
    $r = shift;
    my($buf,$len,$offset) = @_;
    print "READ called, \$buf=$buf, \$len=$len, \$offset=$offset";
}
```

READLINE this

This method will be called when the handle is read from. The method should return undef when there is no more data.

```
sub READLINE {
    $r = shift;
    return "PRINT 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"; }
```

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 for cleaning up.

```
        sub DESTROY {
            print "</shout>\n";
        }
```

94.3.27 Malloc enhancements

If perl is compiled with the malloc included with the perl distribution (that is, if `perl -V:d_mymalloc` is 'define') then you can print memory statistics at runtime by running Perl thusly:

```
  env PERL_DEBUG_MSTATS=2 perl your_script_here
```

The value of 2 means to print statistics after compilation and on exit; with a value of 1, the statistics are printed only on exit. (If you want the statistics at an arbitrary time, you'll need to install the optional module Devel::Peek.)

Three new compilation flags are recognized by malloc.c. (They have no effect if perl is compiled with system malloc().)

-DPERL_EMERGENCY_SBRK

If this macro is defined, running out of memory need not be a fatal error: a memory pool can allocated by assigning to the special variable `$^M`. See §**??**.

-DPACK_MALLOC

Perl memory allocation is by bucket with sizes close to powers of two. Because of these malloc overhead may be big, especially for data of size exactly a power of two. If `PACK_MALLOC` is defined, perl uses a slightly different algorithm for small allocations (up to 64 bytes long), which makes it possible to have overhead down to 1 byte for allocations which are powers of two (and appear quite often).

Expected memory savings (with 8-byte alignment in `alignbytes`) is about 20% for typical Perl usage. Expected slowdown due to additional malloc overhead is in fractions of a percent (hard to measure, because of the effect of saved memory on speed).

-DTWO_POT_OPTIMIZE

Similarly to `PACK_MALLOC`, this macro improves allocations of data with size close to a power of two; but this works for big allocations (starting with 16K by default). Such allocations are typical for big hashes and special-purpose scripts, especially image processing.

On recent systems, the fact that perl requires 2M from system for 1M allocation will not affect speed of execution, since the tail of such a chunk is not going to be touched (and thus will not require real memory). However, it may result in a premature out-of-memory error. So if you will be manipulating very large blocks with sizes close to powers of two, it would be wise to define this macro.

Expected saving of memory is 0-100% (100% in applications which require most memory in such 2**n chunks); expected slowdown is negligible.

94.3.28 Miscellaneous efficiency enhancements

Functions that have an empty prototype and that do nothing but return a fixed value are now inlined (e.g. `sub PI () { 3.14159 }`).

Each unique hash key is only allocated once, no matter how many hashes have an entry with that key. So even if you have 100 copies of the same hash, the hash keys never have to be reallocated.

94.4 Support for More Operating Systems

Support for the following operating systems is new in Perl 5.004.

94.4.1 Win32

Perl 5.004 now includes support for building a "native" perl under Windows NT, using the Microsoft Visual C++ compiler (versions 2.0 and above) or the Borland C++ compiler (versions 5.02 and above). The resulting perl can be used under Windows 95 (if it is installed in the same directory locations as it got installed in Windows NT). This port includes support for perl extension building tools like *MakeMaker* and *h2xs*, so that many extensions available on the Comprehensive Perl Archive Network (CPAN) can now be readily built under Windows NT. See http://www.perl.com/ for more information on CPAN and *README.win32* in the perl distribution for more details on how to get started with building this port.

There is also support for building perl under the Cygwin32 environment. Cygwin32 is a set of GNU tools that make it possible to compile and run many Unix programs under Windows NT by providing a mostly Unix-like interface for compilation and execution. See *README.cygwin32* in the perl distribution for more details on this port and how to obtain the Cygwin32 toolkit.

94.4.2 Plan 9

See *README.plan9* in the perl distribution.

94.4.3 QNX

See *README.qnx* in the perl distribution.

94.4.4 AmigaOS

See *README.amigaos* in the perl distribution.

94.5 Pragmata

Six new pragmatic modules exist:

use autouse MODULE => qw(sub1 sub2 sub3)

Defers `require MODULE` until someone calls one of the specified subroutines (which must be exported by MODULE). This pragma should be used with caution, and only when necessary.

use blib

use blib 'dir'

Looks for MakeMaker-like *'blib'* directory structure starting in *dir* (or current directory) and working back up to five levels of parent directories.

Intended for use on command line with **-M** option as a way of testing arbitrary scripts against an uninstalled version of a package.

use constant NAME => VALUE

Provides a convenient interface for creating compile-time constants, See Constant Functions in *perlsub*.

use locale

Tells the compiler to enable (or disable) the use of POSIX locales for builtin operations.

When `use locale` is in effect, the current LC_CTYPE locale is used for regular expressions and case mapping; LC_COLLATE for string ordering; and LC_NUMERIC for numeric formatting in printf and sprintf (but **not** in print). LC_NUMERIC is always used in write, since lexical scoping of formats is problematic at best.

Each `use locale` or `no locale` affects statements to the end of the enclosing BLOCK or, if not inside a BLOCK, to the end of the current file. Locales can be switched and queried with POSIX::setlocale().

See *perllocale* for more information.

use ops

Disable unsafe opcodes, or any named opcodes, when compiling Perl code.

use vmsish

Enable VMS-specific language features. Currently, there are three VMS-specific features available: 'status', which makes `$?` and `system` return genuine VMS status values instead of emulating POSIX; 'exit', which makes `exit` take a genuine VMS status value instead of assuming that `exit 1` is an error; and 'time', which makes all times relative to the local time zone, in the VMS tradition.

94.6 Modules

94.6.1 Required Updates

Though Perl 5.004 is compatible with almost all modules that work with Perl 5.003, there are a few exceptions:

```
    Module   Required Version for Perl 5.004
    ------   -------------------------------
    Filter   Filter-1.12
    LWP      libwww-perl-5.08
    Tk       Tk400.202 (-w makes noise)
```

Also, the majordomo mailing list program, version 1.94.1, doesn't work with Perl 5.004 (nor with perl 4), because it executes an invalid regular expression. This bug is fixed in majordomo version 1.94.2.

94.6.2 Installation directories

The *installperl* script now places the Perl source files for extensions in the architecture-specific library directory, which is where the shared libraries for extensions have always been. This change is intended to allow administrators to keep the Perl 5.004 library directory unchanged from a previous version, without running the risk of binary incompatibility between extensions' Perl source and shared libraries.

94.6.3 Module information summary

Brand new modules, arranged by topic rather than strictly alphabetically:

```
    CGI.pm              Web server interface ("Common Gateway Interface")
    CGI/Apache.pm       Support for Apache's Perl module
    CGI/Carp.pm         Log server errors with helpful context
    CGI/Fast.pm         Support for FastCGI (persistent server process)
    CGI/Push.pm         Support for server push
    CGI/Switch.pm       Simple interface for multiple server types
    CPAN                Interface to Comprehensive Perl Archive Network
    CPAN::FirstTime     Utility for creating CPAN configuration file
    CPAN::Nox           Runs CPAN while avoiding compiled extensions

    IO.pm               Top-level interface to IO::* classes
    IO/File.pm          IO::File extension Perl module
    IO/Handle.pm        IO::Handle extension Perl module
    IO/Pipe.pm          IO::Pipe extension Perl module
    IO/Seekable.pm      IO::Seekable extension Perl module
    IO/Select.pm        IO::Select extension Perl module
    IO/Socket.pm        IO::Socket extension Perl module

    Opcode.pm           Disable named opcodes when compiling Perl code

    ExtUtils/Embed.pm   Utilities for embedding Perl in C programs
    ExtUtils/testlib.pm Fixes up @INC to use just-built extension

    FindBin.pm          Find path of currently executing program

    Class/Struct.pm     Declare struct-like datatypes as Perl classes
    File/stat.pm        By-name interface to Perl's builtin stat
    Net/hostent.pm      By-name interface to Perl's builtin gethost*
    Net/netent.pm       By-name interface to Perl's builtin getnet*
    Net/protoent.pm     By-name interface to Perl's builtin getproto*
    Net/servent.pm      By-name interface to Perl's builtin getserv*
    Time/gmtime.pm      By-name interface to Perl's builtin gmtime
    Time/localtime.pm   By-name interface to Perl's builtin localtime
    Time/tm.pm          Internal object for Time::{gm,local}time
    User/grent.pm       By-name interface to Perl's builtin getgr*
    User/pwent.pm       By-name interface to Perl's builtin getpw*

    Tie/RefHash.pm      Base class for tied hashes with references as keys

    UNIVERSAL.pm        Base class for *ALL* classes
```

94.6.4 Fcntl

New constants in the existing Fcntl modules are now supported, provided that your operating system happens to support them:

```
    F_GETOWN F_SETOWN
    O_ASYNC O_DEFER O_DSYNC O_FSYNC O_SYNC
    O_EXLOCK O_SHLOCK
```

These constants are intended for use with the Perl operators sysopen() and fcntl() and the basic database modules like SDBM_File. For the exact meaning of these and other Fcntl constants please refer to your operating system's documentation for fcntl() and open().

In addition, the Fcntl module now provides these constants for use with the Perl operator flock():

```
        LOCK_SH LOCK_EX LOCK_NB LOCK_UN
```

These constants are defined in all environments (because where there is no flock() system call, Perl emulates it). However, for historical reasons, these constants are not exported unless they are explicitly requested with the ":flock" tag (e.g. `use Fcntl ':flock'`).

94.6.5 IO

The IO module provides a simple mechanism to load all the IO modules at one go. Currently this includes:

```
     IO::Handle
     IO::Seekable
     IO::File
     IO::Pipe
     IO::Socket
```

For more information on any of these modules, please see its respective documentation.

94.6.6 Math::Complex

The Math::Complex module has been totally rewritten, and now supports more operations. These are overloaded:

```
     + - * / ** <=> neg ~ abs sqrt exp log sin cos atan2 "" (stringify)
```

And these functions are now exported:

```
    pi i Re Im arg
    log10 logn ln cbrt root
    tan
    csc sec cot
    asin acos atan
    acsc asec acot
    sinh cosh tanh
    csch sech coth
    asinh acosh atanh
    acsch asech acoth
    cplx cplxe
```

94.6.7 Math::Trig

This new module provides a simpler interface to parts of Math::Complex for those who need trigonometric functions only for real numbers.

94.6.8 DB_File

There have been quite a few changes made to DB_File. Here are a few of the highlights:

- Fixed a handful of bugs.
- By public demand, added support for the standard hash function exists().
- Made it compatible with Berkeley DB 1.86.
- Made negative subscripts work with RECNO interface.
- Changed the default flags from O_RDWR to O_CREAT|O_RDWR and the default mode from 0640 to 0666.
- Made DB_File automatically import the open() constants (O_RDWR, O_CREAT etc.) from Fcntl, if available.
- Updated documentation.

Refer to the HISTORY section in DB_File.pm for a complete list of changes. Everything after DB_File 1.01 has been added since 5.003.

94.6.9 Net::Ping

Major rewrite - support added for both udp echo and real icmp pings.

94.6.10 Object-oriented overrides for builtin operators

Many of the Perl builtins returning lists now have object-oriented overrides. These are:

```
    File::stat
    Net::hostent
    Net::netent
    Net::protoent
    Net::servent
    Time::gmtime
    Time::localtime
    User::grent
    User::pwent
```

For example, you can now say

```
    use File::stat;
    use User::pwent;
    $his = (stat($filename)->st_uid == pwent($whoever)->pw_uid);
```

94.7 Utility Changes

94.7.1 pod2html

Sends converted HTML to standard output

The *pod2html* utility included with Perl 5.004 is entirely new. By default, it sends the converted HTML to its standard output, instead of writing it to a file like Perl 5.003's *pod2html* did. Use the **–outfile=FILENAME** option to write to a file.

94.7.2 xsubpp

`void` XSUBs now default to returning nothing

Due to a documentation/implementation bug in previous versions of Perl, XSUBs with a return type of `void` have actually been returning one value. Usually that value was the GV for the XSUB, but sometimes it was some already freed or reused value, which would sometimes lead to program failure.

In Perl 5.004, if an XSUB is declared as returning `void`, it actually returns no value, i.e. an empty list (though there is a backward-compatibility exception; see below). If your XSUB really does return an SV, you should give it a return type of `SV *`.

For backward compatibility, *xsubpp* tries to guess whether a `void` XSUB is really `void` or if it wants to return an `SV *`. It does so by examining the text of the XSUB: if *xsubpp* finds what looks like an assignment to `ST(0)`, it assumes that the XSUB's return type is really `SV *`.

94.8 C Language API Changes

`gv_fetchmethod` and `perl_call_sv`

The `gv_fetchmethod` function finds a method for an object, just like in Perl 5.003. The GV it returns may be a method cache entry. However, in Perl 5.004, method cache entries are not visible to users; therefore, they can no longer be passed directly to `perl_call_sv`. Instead, you should use the `GvCV` macro on the GV to extract its CV, and pass the CV to `perl_call_sv`.

The most likely symptom of passing the result of `gv_fetchmethod` to `perl_call_sv` is Perl's producing an "Undefined subroutine called" error on the *second* call to a given method (since there is no cache on the first call).

`perl_eval_pv`

A new function handy for eval'ing strings of Perl code inside C code. This function returns the value from the eval statement, which can be used instead of fetching globals from the symbol table. See *perlguts*, *perlembed* and *perlcall* for details and examples.

Extended API for manipulating hashes

Internal handling of hash keys has changed. The old hashtable API is still fully supported, and will likely remain so. The additions to the API allow passing keys as `SV*`s, so that `tied` hashes can be given real scalars as keys rather than plain strings (nontied hashes still can only use strings as keys). New extensions must use the new hash access functions and macros if they wish to use `SV*` keys. These additions also make it feasible to manipulate `HE*`s (hash entries), which can be more efficient. See *perlguts* for details.

94.9 Documentation Changes

Many of the base and library pods were updated. These new pods are included in section 1:

perldelta

This document.

perlfaq

Frequently asked questions.

perllocale

Locale support (internationalization and localization).

perltoot

Tutorial on Perl OO programming.

perlapio

Perl internal IO abstraction interface.

perlmodlib

Perl module library and recommended practice for module creation. Extracted from *perlmod* (which is much smaller as a result).

perldebug

Although not new, this has been massively updated.

perlsec

Although not new, this has been massively updated.

94.10 New Diagnostics

Several new conditions will trigger warnings that were silent before. Some only affect certain platforms. The following new warnings and errors outline these. These messages are classified as follows (listed in increasing order of desperation):

```
(W) A warning (optional).
(D) A deprecation (optional).
(S) A severe warning (mandatory).
(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).
```

"my" variable %s masks earlier declaration in same scope

(W) A lexical variable has been redeclared in the same scope, 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 referents to it are destroyed.

%s argument is not a HASH element or slice

(F) The argument to delete() must be either a hash element, such as

```
$foo{$bar}
$ref->[12]->{"susie"}
```

or a hash slice, such as

```
@foo{$bar, $baz, $xyzzy}
@{$ref->[12]}{"susie", "queue"}
```

Allocation too large: %lx

(X) You can't allocate more than 64K on an MS-DOS machine.

Allocation too large

(F) You can't allocate more than 2ˆ31+"small amount" bytes.

Applying %s to %s will act on scalar(%s)

(W) 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 grep in *perlfunc* and map in *perlfunc* for alternatives.

Attempt to free nonexistent shared string

(P) 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 use reference as lvalue in substr

(W) 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 substr in *perlfunc*.

Bareword "%s" refers to nonexistent package

(W) 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?

Can’t redefine active sort subroutine %s

(F) Perl optimizes the internal handling of sort subroutines and keeps pointers into them. You tried to redefine one such sort subroutine when it was currently active, which is not allowed. If you really want to do this, you should write `sort { &func } @x` instead of `sort func @x`.

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 *perlref*.

Cannot resolve method ‘%s’ overloading ‘%s’ in package ‘%s’

(P) Internal error trying to resolve overloading specified by a method name (as opposed to a subroutine reference).

Constant subroutine %s redefined

(S) You redefined a subroutine which had previously been eligible for inlining. See Constant Functions in *perlsub* for commentary and workarounds.

Constant subroutine %s undefined

(S) You undefined a subroutine which had previously been eligible for inlining. See Constant Functions in *perlsub* for commentary and workarounds.

Copy method did not return a reference

(F) The method which overloads "=" is buggy. See Copy Constructor in *overload*.

Died

(F) You passed die() an empty string (the equivalent of `die ""`) or you called it with no args and both `$@` and `$_` were empty.

Exiting pseudo-block via %s

(W) 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 sort in *perlfunc*.

Identifier too long

(F) Perl limits identifiers (names for variables, functions, etc.) to 252 characters for simple names, 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.

Illegal character %s (carriage return)

(F) A carriage return character was found in the input. This is an error, and not a warning, because carriage return characters can break multi-line strings, including here documents (e.g., `print <<EOF;`).

Illegal switch in PERL5OPT: %s

(X) The PERL5OPT environment variable may only be used to set the following switches: **-[DIMUdmw]**.

Integer overflow in hex number

(S) The literal hex number you have specified is too big for your architecture. On a 32-bit architecture the largest hex literal is 0xFFFFFFFF.

Integer overflow in octal number

(S) The literal octal number you have specified is too big for your architecture. On a 32-bit architecture the largest octal literal is 037777777777.

internal error: glob failed

(P) Something went wrong with the external program(s) used for `glob` and `<*.c>`. This may 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.

Invalid conversion in %s: "%s"

(W) Perl does not understand the given format conversion. See sprintf in *perlfunc*.

Invalid type in pack: '%s'

(F) The given character is not a valid pack type. See pack in *perlfunc*.

Invalid type in unpack: '%s'

(F) The given character is not a valid unpack type. See unpack in *perlfunc*.

Name "%s::%s" used only once: possible typo

(W) 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 `use vars` pragma is provided for just this purpose).

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 *perlform*.

Offset outside string

(F) You tried to do a read/write/send/recv operation with an offset pointing outside the buffer. This is difficult to imagine. The sole exception to this is that `sysread()`ing past the buffer will extend the buffer and zero pad the new area.

Out of memory!

(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*.

Out of memory during 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.

panic: frexp

(P) The library function frexp() failed, making printf("%f") impossible.

Possible attempt to put comments in qw() list

(W) 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() 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 !;
```

Scalar value @%s{%s} better written as $ %s{%s}

(W) 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.

Stub found while resolving method '%s' overloading '%s' in %s

(P) Overloading resolution over @ISA tree may be broken by importing stubs. Stubs should never be implicitly created, but explicit calls to `can` may break this.

Too late for "-T" option

(X) The #! line (or local equivalent) in a Perl script contains the **-T** option, but Perl was not invoked with **-T** in its argument list. 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.

untie attempted while %d inner references still exist

(W) A copy of the object returned from `tie` (or `tied`) was still valid when `untie` was called.

Unrecognized character %s

(F) The Perl parser has no idea what to do with the specified character in your Perl script (or eval). Perhaps you tried to run a compressed script, a binary program, or a directory as a Perl program.

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.

Use of "$ $ <digit>" to mean "$ {$ }<digit>" is deprecated

(D) Perl versions before 5.004 misinterpreted any type marker followed by "$" and a digit. For example, "$$0" was incorrectly taken to mean "${$}0" instead of "${$0}". This bug is (mostly) fixed in Perl 5.004.

However, the developers of Perl 5.004 could not fix this bug completely, because at least two widely-used modules depend on the old meaning of "$$0" in a string. So Perl 5.004 still interprets "$$<digit>" in the old (broken) way inside strings; but it generates this message as a warning. And in Perl 5.005, this special treatment will cease.

Value of %s can be "0"; test with defined()

(W) 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.

Variable "%s" may be unavailable

(W) An inner (nested) *anonymous* subroutine is inside a *named* subroutine, and outside that is another subroutine; and the anonymous (innermost) subroutine is referencing a lexical variable defined in the outermost subroutine. For example:

```
sub outermost { my $a; sub middle { sub { $a } } }
```

If the anonymous subroutine is called or referenced (directly or indirectly) from the outermost subroutine, it will share the variable as you would expect. But if the anonymous subroutine is called or referenced when the outermost subroutine is not active, it will see the value of the shared variable as it was before and during the *first* call to the outermost subroutine, which is probably not what you want.

In these circumstances, it is usually best to make the middle subroutine anonymous, using the `sub {}` syntax. Perl has specific support for shared variables in nested anonymous subroutines; a named subroutine in between interferes with this feature.

Variable "%s" will not stay shared

(W) An inner (nested) *named* subroutine is referencing a lexical variable defined in an outer subroutine.

When the inner subroutine is called, it will probably 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.

Furthermore, if the outer subroutine is anonymous and references a lexical variable outside itself, then the outer and inner subroutines will *never* share the given variable.

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 called or referenced, they are automatically rebound to the current values of such variables.

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.

Ill-formed logical name |%s| in prime_env_iter

(W) A warning peculiar to VMS. A logical name was encountered when preparing to iterate over %ENV which violates the syntactic rules governing logical names. Since it cannot be translated normally, it is skipped, and will not appear in %ENV. This may be a benign occurrence, as some software packages might directly modify logical name tables and introduce nonstandard names, or it may indicate that a logical name table has been corrupted.

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.

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 *README.os2*.

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 *README.os2*.

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 Signals in *perlipc*. See also "Process terminated by SIGTERM/SIGINT" in *README.os2*.

94.11 BUGS

If you find what you think is a bug, you might check the headers of recently posted articles in the comp.lang.perl.misc newsgroup. There may also be information at http://www.perl.com/perl/ , the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Make sure you trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to <*perlbug@perl.com*> to be analysed by the Perl porting team.

94.12 SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl. This file has been significantly updated for 5.004, so even veteran users should look through it.

The *README* file for general stuff.

The *Copying* file for copyright information.

94.13 HISTORY

Constructed by Tom Christiansen, grabbing material with permission from innumerable contributors, with kibitzing by more than a few Perl porters.

Last update: Wed May 14 11:14:09 EDT 1997

Chapter 95

perlartistic

The Perl Artistic License

95.1 SYNOPSIS

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

95.2 DESCRIPTION

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.

It is also one of the two licenses Perl allows itself to be redistributed and/or modified; for the other one, the GNU General Public License, see the *perlgpl*.

95.3 The "Artistic License"

95.3.1 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.

95.3.2 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.

95.3.3 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

Chapter 96

perlgpl

The GNU General Public License, version 2

96.1 SYNOPSIS

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

96.2 DESCRIPTION

This is **"The GNU General Public License, version 2"**. It's here so that modules, programs, etc., that want to declare this as their distribution license, can link to it.

It is also one of the two licenses Perl allows itself to be redistributed and/or modified; for the other one, the Perl Artistic License, see the *perlartistic*.

96.3 GNU GENERAL PUBLIC LICENSE

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.
                          59 Temple Place - Suite 330, Boston, MA
                          02111-1307, USA.
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble
```

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU 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. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), 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 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 show them these terms so they know 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.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

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 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 derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

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 License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

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.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

```
    a) You must cause the modified files to carry prominent notices
    stating that you changed the files and the date of any change.

    b) You must cause any work that you distribute or publish, that in
    whole or in part contains or is derived from the Program or any
    part thereof, to be licensed as a whole at no charge to all third
    parties under the terms of this License.

    c) If the modified program normally reads commands interactively
    when run, you must cause it, when started running for such
    interactive use in the most ordinary 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
    License.  (Exception: if the Program itself is interactive but
    does not normally print such an announcement, your work based on
    the Program is not required to print an announcement.)
```

–

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 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 Sections
    1 and 2 above on a medium customarily used for software interchange; or,

    b) Accompany it with a written offer, valid for at least three
    years, to give any third party, for a charge no more than your
    cost of physically performing source distribution, a complete
    machine-readable copy of the corresponding source code, to be
    distributed under the terms of Sections 1 and 2 above on a medium
    customarily used for software interchange; or,

    c) Accompany it with the information you received as to the offer
    to distribute corresponding source code.  (This alternative is
    allowed only for noncommercial distribution and only if you
    received the program in object code or executable form with such
    an offer, in accord with Subsection b above.)
```

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

–

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this

License. Therefore, by modifying or distributing 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 for copying, distributing or modifying the Program or works based on it.

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. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

–

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. 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 this 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 this License, you may choose any version ever published by the Free Software Foundation.

10. 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
```

11. 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.

12. 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 the public, 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 2 of the License, 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., 59 Temple Place - Suite 330, Boston, MA 02111-1307, 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) 19yy 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 is a sample; alter the names:

```
  Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  'Gnomovision' (which makes passes at compilers) written by James Hacker.

  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

[End.]

Part VI

Platform-Specific

Chapter 97

README.aix

Perl version 5 on IBM Unix (AIX) systems

97.1 DESCRIPTION

This document describes various features of IBM's Unix operating system (AIX) that will affect how Perl version 5 (hereafter just Perl) is compiled and/or runs.

97.1.1 Compiling Perl 5 on AIX

When compiling Perl, you must use an ANSI C compiler. AIX does not ship an ANSI compliant C-compiler with AIX by default, but binary builds of gcc for AIX are widely available.

At the moment of writing, AIX supports two different native C compilers, for which you have to pay: **xlC** and **vac**. If you decide to use either of these two (which is quite a lot easier than using gcc), be sure to upgrade to the latest available patch level. Currently:

```
    xlC.C     3.1.4.10 or 3.6.6.0 or 4.0.2.2 or 5.0.2.9 or 6.0.0.3
    vac.C     4.4.0.3  or 5.0.2.6 or 6.0.0.1
```

note that xlC has the OS version in the name as of version 4.0.2.0, so you will find xlC.C for AIX-5.0 as package

```
    xlC.aix50.rte   5.0.2.0 or 6.0.0.3
```

subversions are not the same 'latest' on all OS versions. For example, the latest xlC-5 on aix41 is 5.0.2.9, while on aix43, it is 5.0.2.7.

Perl can be compiled with either IBM's ANSI C compiler or with gcc. The former is recommended, as not only can it compile Perl with no difficulty, but also can take advantage of features listed later that require the use of IBM compiler-specific command-line flags.

The IBM's compiler patch levels 5.0.0.0 and 5.0.1.0 have compiler optimization bugs that affect compiling perl.c and regcomp.c, respectively. If Perl's configuration detects those compiler patch levels, optimization is turned off for the said source code files. Upgrading to at least 5.0.2.0 is recommended.

If you decide to use gcc, make sure your installation is recent and complete, and be sure to read the Perl README file for more gcc-specific details. Please report any hoops you had to jump through to the development team.

97.1.2 OS level

Before installing the patches to the IBM C-compiler you need to know the level of patching for the Operating System. IBM's command 'oslevel' will show the base, but is not always complete (in this example oslevel shows 4.3.NULL, whereas the system might run most of 4.3.THREE):

```
# oslevel
4.3.0.0
# lslpp -l | grep 'bos.rte '
bos.rte           4.3.3.75  COMMITTED  Base Operating System Runtime
bos.rte            4.3.2.0  COMMITTED  Base Operating System Runtime
#
```

The same might happen to AIX 5.1 or other OS levels. As a side note, perl cannot be built without bos.adt.syscalls and bos.adt.libm installed

```
# lslpp -l | egrep "syscalls|libm"
bos.adt.libm      5.1.0.25  COMMITTED  Base Application Development
bos.adt.syscalls  5.1.0.36  COMMITTED  System Calls Application
#
```

97.1.3 Building Dynamic Extensions on AIX

AIX supports dynamically loadable objects as well as shared libraries. Shared libraries by convention end with the suffix .a, which is a bit misleading, as an archive can contain static as well as dynamic members. For perl dynamically loaded objects we use the .so suffix also used on many other platforms.

Note that starting from Perl 5.7.2 (and consequently 5.8.0) and AIX 4.3 or newer Perl uses the AIX native dynamic loading interface in the so called runtime linking mode instead of the emulated interface that was used in Perl releases 5.6.1 and earlier or, for AIX releases 4.2 and earlier. This change does break backward compatibility with compiled modules from earlier perl releases. The change was made to make Perl more compliant with other applications like Apache/mod_perl which are using the AIX native interface. This change also enables the use of C++ code with static constructors and destructors in perl extensions, which was not possible using the emulated interface.

97.1.4 The IBM ANSI C Compiler

All defaults for Configure can be used.

If you've chosen to use vac 4, be sure to run 4.4.0.3. Older versions will turn up nasty later on. For vac 5 be sure to run at least 5.0.1.0, but vac 5.0.2.6 or up is highly recommended. Note that since IBM has removed vac 5.0.2.1 through 5.0.2.5 from the software depot, these versions should be considered obsolete.

Here's a brief lead of how to upgrade the compiler to the latest level. Of course this is subject to changes. You can only upgrade versions from ftp-available updates if the first three digit groups are the same (in where you can skip intermediate unlike the patches in the developer snapshots of perl), or to one version up where the 'base' is available. In other words, the AIX compiler patches are cumulative.

```
vac.C.4.4.0.1 => vac.C.4.4.0.3  is OK     (vac.C.4.4.0.2 not needed)
xlC.C.3.1.3.3 => xlC.C.3.1.4.10 is NOT OK (xlC.C.3.1.4.0 is not available)

# ftp ftp.software.ibm.com
Connected to service.boulder.ibm.com.
: welcome message ...
Name (ftp.software.ibm.com:merijn): anonymous
331 Guest login ok, send your complete e-mail address as password.
Password:
... accepted login stuff
```

```
ftp> cd /aix/fixes/v4/
ftp> dir other other.ll
output to local-file: other.ll? y
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
226 Transfer complete.
ftp> dir xlc xlc.ll
output to local-file: xlc.ll? y
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
226 Transfer complete.
ftp> bye
... goodbye messages
# ls -l *.ll
-rw-rw-rw-   1 merijn   system    1169432 Nov  2 17:29 other.ll
-rw-rw-rw-   1 merijn   system      29170 Nov  2 17:29 xlc.ll
```

On AIX 4.2 using xlC, we continue:

```
# lslpp -l | fgrep 'xlC.C '
  xlC.C                     3.1.4.9  COMMITTED  C for AIX Compiler
  xlC.C                     3.1.4.0  COMMITTED  C for AIX Compiler
# grep 'xlC.C.3.1.4.*.bff' xlc.ll
-rw-r--r--   1 45776101 1       6286336 Jul 22 1996  xlC.C.3.1.4.1.bff
-rw-rw-r--   1 45776101 1       6173696 Aug 24 1998  xlC.C.3.1.4.10.bff
-rw-r--r--   1 45776101 1       6319104 Aug 14 1996  xlC.C.3.1.4.2.bff
-rw-r--r--   1 45776101 1       6316032 Oct 21 1996  xlC.C.3.1.4.3.bff
-rw-r--r--   1 45776101 1       6315008 Dec 20 1996  xlC.C.3.1.4.4.bff
-rw-rw-r--   1 45776101 1       6178816 Mar 28 1997  xlC.C.3.1.4.5.bff
-rw-rw-r--   1 45776101 1       6188032 May 22 1997  xlC.C.3.1.4.6.bff
-rw-rw-r--   1 45776101 1       6191104 Sep  5 1997  xlC.C.3.1.4.7.bff
-rw-rw-r--   1 45776101 1       6185984 Jan 13 1998  xlC.C.3.1.4.8.bff
-rw-rw-r--   1 45776101 1       6169600 May 27 1998  xlC.C.3.1.4.9.bff
# wget ftp://ftp.software.ibm.com/aix/fixes/v4/xlc/xlC.C.3.1.4.10.bff
#
```

On AIX 4.3 using vac, we continue:

```
# lslpp -l | grep 'vac.C '
 vac.C                      5.0.2.2  COMMITTED  C for AIX Compiler
 vac.C                      5.0.2.0  COMMITTED  C for AIX Compiler
# grep 'vac.C.5.0.2.*.bff' other.ll
-rw-rw-r--   1 45776101 1       13592576 Apr 16 2001  vac.C.5.0.2.0.bff
-rw-rw-r--   1 45776101 1       14133248 Apr  9 2002  vac.C.5.0.2.3.bff
-rw-rw-r--   1 45776101 1       14173184 May 20 2002  vac.C.5.0.2.4.bff
-rw-rw-r--   1 45776101 1       14192640 Nov 22 2002  vac.C.5.0.2.6.bff
# wget ftp://ftp.software.ibm.com/aix/fixes/v4/other/vac.C.5.0.2.6.bff
#
```

Likewise on all other OS levels. Then execute the following command, and fill in its choices

```
# smit install_update
 -> Install and Update from LATEST Available Software
 * INPUT device / directory for software [ vac.C.5.0.2.6.bff    ]
 [ OK ]
 [ OK ]
```

Follow the messages ... and you're done.

If you like a more web-like approach, a good start point can be http://www14.software.ibm.com/webapp/download/downloadaz.jsp and click "C for AIX", and follow the instructions.

97.1.5 The usenm option

If linking miniperl

```
cc -o miniperl ... miniperlmain.o opmini.o perl.o ... -lm -lc ...
```

causes error like this

```
ld: 0711-317 ERROR: Undefined symbol: .aintl
ld: 0711-317 ERROR: Undefined symbol: .copysignl
ld: 0711-317 ERROR: Undefined symbol: .syscall
ld: 0711-317 ERROR: Undefined symbol: .eaccess
ld: 0711-317 ERROR: Undefined symbol: .setresuid
ld: 0711-317 ERROR: Undefined symbol: .setresgid
ld: 0711-317 ERROR: Undefined symbol: .setproctitle
ld: 0711-345 Use the -bloadmap or -bnoquiet option to obtain more information.
```

you could retry with

```
make realclean
rm config.sh
./Configure -Dusenm ...
```

which makes Configure to use the `nm` tool when scanning for library symbols, which usually is not done in AIX.

Related to this, you probably should not use the `-r` option of Configure in AIX, because that affects of how the `nm` tool is used.

97.1.6 Using GNU's gcc for building perl

Using gcc-3.x (tested with 3.0.4, 3.1, and 3.2) now works out of the box, as do recent gcc-2.9 builds available directly from IBM as part of their Linux compatibility packages, available here:

```
 http://www.ibm.com/servers/aix/products/aixos/linux/
```

97.1.7 Using Large Files with Perl

Should yield no problems.

97.1.8 Threaded Perl

Threads seem to work OK, though at the moment not all tests pass when threads are used in combination with 64-bit configurations.

You may get a warning when doing a threaded build:

```
 "pp_sys.c", line 4640.39: 1506-280 (W) Function argument assignment between types "unsigned char*" an
```

The exact line number may vary, but if the warning (W) comes from a line line this

```
 hent = PerlSock_gethostbyaddr(addr, (Netdb_hlen_t) addrlen, addrtype);
```

in the "pp_ghostent" function, you may ignore it safely. The warning is caused by the reentrant variant of gethostbyaddr() having a slightly different prototype than its non-reentrant variant, but the difference is not really significant here.

97.1.9 64-bit Perl

If your AIX is installed with 64-bit support, you can expect 64-bit configurations to work. In combination with threads some tests might still fail.

97.1.10 AIX 4.2 and extensions using C++ with statics

In AIX 4.2 Perl extensions that use C++ functions that use statics may have problems in that the statics are not getting initialized. In newer AIX releases this has been solved by linking Perl with the libC_r library, but unfortunately in AIX 4.2 the said library has an obscure bug where the various functions related to time (such as time() and gettimeofday()) return broken values, and therefore in AIX 4.2 Perl is not linked against the libC_r.

97.2 AUTHOR

H.Merijn Brand <h.m.brand@hccnet.nl>

97.3 DATE

Version 0.0.6: 23 Dec 2002

Chapter 98

perlamiga

Perl under Amiga OS

98.1 NOTE

Perl 5.8.0 cannot be built in AmigaOS. You can use either the maintenance release Perl 5.6.1 or the development release Perl 5.7.2 in AmigaOS. See §98.7 if you want to help fixing this problem.

98.2 SYNOPSIS

One can read this document in the following formats:

```
        man perlamiga
        multiview perlamiga.guide
```

to list some (not all may be available simultaneously), or it may be read *as is*: either as *README.amiga*, or *pod/perlamiga.pod*.

A recent version of perl for the Amiga can be found at the Geek Gadgets section of the Aminet:

```
      http://www.aminet.net/~aminet/dev/gg/index.html
```

98.3 DESCRIPTION

98.3.1 Prerequisites for Compiling Perl on AmigaOS

Unix emulation for AmigaOS: ixemul.library

You need the Unix emulation for AmigaOS, whose most important part is **ixemul.library**. For a minimum setup, get the latest versions of the following packages from the Aminet archives (http://www.aminet.net/˜aminet/):

```
        ixemul-bin
        ixemul-env-bin
        pdksh-bin
```

Note also that this is a minimum setup; you might want to add other packages of **ADE** (the *Amiga Developers Environment*).

Version of Amiga OS

You need at the very least AmigaOS version 2.0. Recommended is version 3.1.

98.3.2 Starting Perl programs under AmigaOS

Start your Perl program *foo* with arguments `arg1 arg2 arg3` the same way as on any other platform, by

```
perl foo arg1 arg2 arg3
```

If you want to specify perl options `-my_opts` to the perl itself (as opposed to your program), use

```
perl -my_opts foo arg1 arg2 arg3
```

Alternately, you can try to get a replacement for the system's **Execute** command that honors the #!/usr/bin/perl syntax in scripts and set the s-Bit of your scripts. Then you can invoke your scripts like under UNIX with

```
foo arg1 arg2 arg3
```

(Note that having *nixish full path to perl */usr/bin/perl* is not necessary, *perl* would be enough, but having full path would make it easier to use your script under *nix.)

98.3.3 Shortcomings of Perl under AmigaOS

Perl under AmigaOS lacks some features of perl under UNIX because of deficiencies in the UNIX-emulation, most notably:

- fork()
- some features of the UNIX filesystem regarding link count and file dates
- inplace operation (the -i switch) without backup file
- umask() works, but the correct permissions are only set when the file is finally close()d

98.4 INSTALLATION

Change to the installation directory (most probably ADE:), and extract the binary distribution:
lha -mraxe x perl-$VERSION-bin.lha
or
tar xvzpf perl-$VERSION-bin.tgz
(Of course you need lha or tar and gunzip for this.)
For installation of the Unix emulation, read the appropriate docs.

98.5 Accessing documentation

98.5.1 Manpages for Perl on AmigaOS

If you have `man` installed on your system, and you installed perl manpages, use something like this:

```
man perlfunc
man less
man ExtUtils.MakeMaker
```

to access documentation for different components of Perl. Start with

```
man perl
```

Note: You have to modify your man.conf file to search for manpages in the /ade/lib/perl5/man/man3 directory, or the man pages for the perl library will not be found.
Note that dot (.) is used as a package separator for documentation for packages, and as usual, sometimes you need to give the section - `3` above - to avoid shadowing by the *less(1) manpage*.

98.5.2 Perl HTML Documentation on AmigaOS

If you have some WWW browser available, you can build **HTML** docs. Cd to directory with *.pod* files, and do like this

```
        cd /ade/lib/perl5/pod
        pod2html
```

After this you can direct your browser the file *perl.html* in this directory, and go ahead with reading docs.
Alternatively you may be able to get these docs prebuilt from `CPAN`.

98.5.3 Perl GNU Info Files on AmigaOS

Users of `Emacs` would appreciate it very much, especially with `CPerl` mode loaded. You need to get latest `pod2info` from `CPAN`, or, alternately, prebuilt info pages.

98.5.4 Perl LaTeX Documentation on AmigaOS

Can be constructed using `pod2latex`.

98.6 BUILDING PERL ON AMIGAOS

Here we discuss how to build Perl under AmigaOS.

98.6.1 Build Prerequisites for Perl on AmigaOS

You need to have the latest **ixemul** (Unix emulation for Amiga) from Aminet.

98.6.2 Getting the Perl Source for AmigaOS

You can either get the latest perl-for-amiga source from Ninemoons and extract it with:

```
  tar xvzpf perl-$VERSION-src.tgz
```

or get the official source from CPAN:

```
  http://www.cpan.org/src/5.0
```

Extract it like this

```
  tar xvzpf perl-$VERSION.tar.gz
```

You will see a message about errors while extracting *Configure*. This is normal and expected. (There is a conflict with a similarly-named file *configure*, but it causes no harm.)

98.6.3 Making Perl on AmigaOS

Remember to use a hefty wad of stack (I use 2000000)

```
  sh configure.gnu --prefix=/gg
```

Now type

```
  make depend
```

Now!

```
  make
```

98.6.4 Testing Perl on AmigaOS

Now run

```
  make test
```

Some tests will be skipped because they need the fork() function:
io/pipe.t, op/fork.t, lib/filehand.t, lib/open2.t, lib/open3.t, lib/io_pipe.t, lib/io_sock.t

98.6.5 Installing the built Perl on AmigaOS

Run

```
  make install
```

98.7 PERL 5.8.0 BROKEN IN AMIGAOS

As told above, Perl 5.6.1 was still good in AmigaOS, as was 5.7.2. After Perl 5.7.2 (change #11423, see the Changes file, and the file pod/perlhack.pod for how to get the individual changes) Perl dropped its internal support for vfork(), and that was very probably the step that broke AmigaOS (since the ixemul library has only vfork). The build finally fails when the ext/DynaLoader is being built, and PERL ends up as "0" in the produced Makefile, trying to run "0" does not quite work. Also, executing miniperl in backticks seems to generate nothing: very probably related to the (v)fork problems. **Fixing the breakage requires someone quite familiar with the ixemul library, and how one is supposed to run external commands in AmigaOS without fork().**

98.8 AUTHORS

Norbert Pueschel, pueschel@imsdd.meb.uni-bonn.de Jan-Erik Karlsson, trg@privat.utfors.se

98.9 SEE ALSO

perl(1).

Chapter 99

README.apollo

Perl version 5 on Apollo DomainOS

99.1 DESCRIPTION

The following tests are known to fail as of Perl 5.005_03:

comp/decl..........FAILED at test 0 op/write...........FAILED at test 0 lib/filefind.......FAILED at test 2
lib/io_udp.........FAILED at test 2
lib/findbin........stat(/ressel/ABT/USER/vta/jk/proj.local/perl/perl5.005_03-MAINT_TRIAL_5/t/lib/): No such file or directory at ../lib/FindBin.pm line 162
stat(/ressel/ABT/USER/vta/jk/proj.local/perl/perl5.005_03-MAINT_TRIAL_5/t/lib/): No such file or directory at ../lib/FindBin.pm line 163 FAILED at test 1

99.2 AUTHOR

Johann Klasek <jk@auto.tuwien.ac.at>

Chapter 100

README.beos

Perl version 5 on BeOS

100.1 DESCRIPTION

Notes for building Perl under BeOS.

100.1.1 General Issues with Perl on BeOS

To compile perl under BeOS R4 x86:

```
        ./Configure -d
```

and hit ˆC when it asks you if you want to make changes to config.sh; edit config.sh and do the following: change d_socket='define' to ='undef'; remove SDBM, Errno, and Socket from dynamic_ext= and nonxs_ext=; add '#define bool short' to x2p/a2p.h;

```
        ../Configure -S; make; make install

        cd ~/config/lib; ln -s 5.00502/BeOS-BePC/CORE/libperl.so .
```

(substitute 5.00502 with the appropriate filename)

100.1.2 BeOS Release-specific Notes

R4 x86

Dynamic loading finally works! Yay! This means you can compile your own modules into perl. However, Sockets and Errno still don't work. (Hopefully, sockets will at least work by R5, if not sooner.)

R4 PPC

I have not tested this. I rather severely doubt that dynamic loading will work. (My BeBox is in pieces right now, following a nasty disk crash.) You may have to disable dynamic loading to get the thing to compile at all. (use './Configure' without -d, and say 'no' to 'Build a shared libperl.so'.)

100.1.3 Contact Information

If you have comments, problem reports, or even patches or bugfixes (gasp!) please email me.

28 Jan 1999 Tom Spindler dogcow@isi.net

100.1.4 Update 2002-05-30

The following tests fail on 5.8.0 Perl in BeOS Personal 5.03:

```
t/op/lfs............................FAILED at test 17
t/op/magic..........................FAILED at test 24
ext/Fcntl/t/syslfs..................FAILED at test 17
ext/File/Glob/t/basic...............FAILED at test 3
ext/POSIX/t/sigaction...............FAILED at test 13
ext/POSIX/t/waitpid.................FAILED at test 1
```

The reasons for the failures are as follows:

- The t/op/lfs and ext/Fcntl/t/syslfs failures indicate that the LFS (large file support, files larger than 2 gigabytes) doesn't work from Perl (BeFS itself is well capable of supporting large files). What fails is that trying to position the file pointer past 2 gigabytes doesn't work right, the position gets truncated to its lower 32 bits.
- The op/magic failures look like something funny going on with $0 and $^X that I can't now figure out: none of the generated pathnames are wrong as such, they just seem to accumulate "./" prefixes and infixes in ways that define logic.
- The Glob/t/basic indicates a bug in the getpw*() functions: they do not always return the correct user db entries.
- The sigaction #13 means that signal mask doesn't get properly restored if sigaction returns early.
- The waitpid failure means that after there are no more child processes, waitpid is supposed to start returning -1 (and set errno to ECHILD). In BeOS, it doesn't seem to.

Disclaimer: I just installed BeOS Personal Edition 5.0 and the Developer Tools, that is the whole extent of my BeOS expertise, so please don't ask me for further help in BeOS Perl problems.

jhi@iki.fi

Chapter 101

README.BS2000

Building and installing Perl for BS2000.

101.1 SYNOPSIS

This document will help you Configure, build, test and install Perl on BS2000 in the POSIX subsystem.

101.2 DESCRIPTION

This is a ported perl for the POSIX subsystem in BS2000 VERSION OSD V3.1A or later. It may work on other versions, but we started porting and testing it with 3.1A and are currently using Version V4.0A.

You may need the following GNU programs in order to install perl:

101.2.1 gzip on BS2000

We used version 1.2.4, which could be installed out of the box with one failure during 'make check'.

101.2.2 bison on BS2000

The yacc coming with BS2000 POSIX didn't work for us. So we had to use bison. We had to make a few changes to perl in order to use the pure (reentrant) parser of bison. We used version 1.25, but we had to add a few changes due to EBCDIC. See below for more details concerning yacc.

101.2.3 Unpacking Perl Distribution on BS2000

To extract an ASCII tar archive on BS2000 POSIX you need an ASCII filesystem (we used the mountpoint /usr/local/ascii for this). Now you extract the archive in the ASCII filesystem without I/O-conversion:

cd /usr/local/ascii export IO_CONVERSION=NO gunzip < /usr/local/src/perl.tar.gz | pax -r

You may ignore the error message for the first element of the archive (this doesn't look like a tar archive / skipping to next file...), it's only the directory which will be created automatically anyway.

After extracting the archive you copy the whole directory tree to your EBCDIC filesystem. **This time you use I/O-conversion**:

cd /usr/local/src IO_CONVERSION=YES cp -r /usr/local/ascii/perl5.005_02 ./

101.2.4 Compiling Perl on BS2000

There is a "hints" file for BS2000 called hints.posix-bc (because posix-bc is the OS name given by ‘uname‘) that specifies the correct values for most things. The major problem is (of course) the EBCDIC character set. We have german EBCDIC version.

Because of our problems with the native yacc we used GNU bison to generate a pure (=reentrant) parser for perly.y. So our yacc is really the following script:

—–8<—–/usr/local/bin/yacc—–8<—– #! /usr/bin/sh

# Bison as a reentrant yacc:

# save parameters: params="" while [[$# -gt 1]]; do params="$params $1" shift done

# add flag %pure_parser:

tmpfile=/tmp/bison.$$.y echo %pure_parser > $tmpfile cat $1 >> $tmpfile

# call bison:

echo "/usr/local/bin/bison –yacc $params $1\t\t\t(Pure Parser)" /usr/local/bin/bison –yacc $params $tmpfile

# cleanup:

rm -f $tmpfile —–8<———–-8<—–

We still use the normal yacc for a2p.y though!!! We made a softlink called byacc to distinguish between the two versions:

ln -s /usr/bin/yacc /usr/local/bin/byacc

We build perl using GNU make. We tried the native make once and it worked too.

101.2.5 Testing Perl on BS2000

We still got a few errors during `make test`. Some of them are the result of using bison. Bison prints *parser error* instead of *syntax error*, so we may ignore them. The following list shows our errors, your results may differ:

op/numconvert.......FAILED tests 1409-1440 op/regexp...........FAILED tests 483, 496 op/regexp_noamp.....FAILED tests 483, 496 pragma/overload.....FAILED tests 152-153, 170-171 pragma/warnings.....FAILED tests 14, 82, 129, 155, 192, 205, 207 lib/bigfloat........FAILED tests 351-352, 355 lib/bigfltpm........FAILED tests 354-355, 358 lib/complex.........FAILED tests 267, 487 lib/dumper..........FAILED tests 43, 45 Failed 11/231 test scripts, 95.24% okay. 57/10595 subtests failed, 99.46% okay.

101.2.6 Installing Perl on BS2000

We have no nroff on BS2000 POSIX (yet), so we ignored any errors while installing the documentation.

101.2.7 Using Perl in the Posix-Shell of BS2000

BS2000 POSIX doesn’t support the shebang notation (`#!/usr/local/bin/perl`), so you have to use the following lines instead:

: # use perl eval ’exec /usr/local/bin/perl -S $0 ${1+"$@"}’ if $running_under_some_shell;

101.2.8 Using Perl in "native" BS2000

We don’t have much experience with this yet, but try the following:

Copy your Perl executable to a BS2000 LLM using bs2cp:

```
bs2cp /usr/local/bin/perl 'bs2:perl(perl,l)'
```

Now you can start it with the following (SDF) command:

```
/START-PROG FROM-FILE=*MODULE(PERL,PERL),PROG-MODE=*ANY,RUN-MODE=*ADV
```

First you get the BS2000 commandline prompt (’*’). Here you may enter your parameters, e.g. `-e 'print "Hello World!\\n";'` (note the double backslash!) or `-w` and the name of your Perl script. Filenames starting with `/` are searched in the Posix filesystem, others are searched in the BS2000 filesystem. You may even use wildcards if you put a `%` in front of your filename (e.g. `-w checkfiles.pl %*.c`). Read your C/C++ manual for additional possibilities of the commandline prompt (look for PARAMETER-PROMPTING).

101.2.9 Floating point anomalies on BS2000

There appears to be a bug in the floating point implementation on BS2000 POSIX systems such that calling int() on the product of a number and a small magnitude number is not the same as calling int() on the quotient of that number and a large magnitude number. For example, in the following Perl code:

```
    my $x = 100000.0;
    my $y = int($x * 1e-5) * 1e5; # '0'
    my $z = int($x / 1e+5) * 1e5;  # '100000'
    print "\$y is $y and \$z is $z\n"; # $y is 0 and $z is 100000
```

Although one would expect the quantities $y and $z to be the same and equal to 100000 they will differ and instead will be 0 and 100000 respectively.

101.2.10 Using PerlIO and different encodings on ASCII and EBCDIC partitions

Since version 5.8 Perl uses the new PerlIO on BS2000. This enables you using 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(posix-bc)", "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 two files containing "Hello World!\n" in ASCII, EBCDIC, ISO Latin-1 (in this example identical to ASCII) respective UTF-EBCDIC (in this example identical to normal EBCDIC). See the documentation of Encode::PerlIO for details.

As the PerlIO layer uses raw IO internally, all this totally ignores the type of your filesystem (ASCII or EBCDIC) and the IO_CONVERSION environment variable. If you want to get the old behavior, that the BS2000 IO functions determine conversion depending on the filesystem PerlIO still is your friend. You use IO_CONVERSION as usual and tell Perl, that it should use the native IO layer:

```
    export IO_CONVERSION=YES
    export PERLIO=stdio
```

Now your IO would be ASCII on ASCII partitions and EBCDIC on EBCDIC partitions. See the documentation of PerlIO (without `Encode::`!) for further posibilities.

101.3 AUTHORS

Thomas Dorner

101.4 SEE ALSO

INSTALL, *perlport*.

101.4.1 Mailing list

If you are interested in the VM/ESA, z/OS (formerly known as OS/390) and POSIX-BC (BS2000) ports of Perl then see the perl-mvs mailing list. To subscribe, send an empty message to perl-mvs-subscribe@perl.org.

See also:

```
http://lists.perl.org/showlist.cgi?name=perl-mvs
```

There are web archives of the mailing list at:

```
http://www.xray.mpe.mpg.de/mailing-lists/perl-mvs/
http://archive.develooper.com/perl-mvs@perl.org/
```

101.5 HISTORY

This document was originally written by Thomas Dorner for the 5.005 release of Perl.

This document was podified for the 5.6 release of perl 11 July 2000.

Chapter 102

perlce

Perl for WinCE

102.1 DESCRIPTION

This file gives the instructions for building Perl5.8 and above for WinCE. Please read and understand the terms under which this software is distributed.

102.2 BUILD

This section describes the steps to be performed to build PerlCE. You may find additional and newer information about building perl for WinCE using following URL:

```
  http://perlce.sourceforge.net
```

There should also be pre-built binaries there.

Don't be confused by large size of downloaded distribution or constructed binaries: entire distribution could be large for WinCE ideology, but you may strip it at your wish and use only required parts.

102.2.1 Tools & SDK

For compiling, you need following:

- Microsoft Embedded Visual Tools
- Microsoft Visual C++
- Rainer Keuchel's celib-sources
- Rainer Keuchel's console-sources

Needed source files can be downloaded via: www.rainer-keuchel.de/wince/dirlist.html

102.2.2 Make

Please pay attention that starting from 5.8.0 miniperl *is* built and it facilitates in further building process. This means that in addition to compiler installation for mobile device you also need to have Microsoft Visual C++ installed as well.
On the bright side, you do not need to edit any files from ./win32 subdirectory. Normally you only need to edit ./wince/compile.bat to reflect your system and run it.
File ./wince/compile.bat is actually a wrapper to call nmake -f makefile.ce with appropriate parameters and it accepts extra parameters and forwards them to "nmake" command as additional arguments. You should pass target this way.
To prepare distribution you need to do following:

- go to ./wince subdirectory
- edit file compile.bat
- run compile.bat
- run compile.bat dist

makefile.ce has CROSS_NAME macro, and it is used further to refer to your cross-compilation scheme. You could assign a name to it, but this is not necessary, because by default it is assigned after your machine configuration name, such as "wince-sh3-hpc-wce211", and this is enough to distinguish different builds at the same time. This option could be handy for several different builds on same platform to perform, say, threaded build. In a following example we assume that all required environment variables are set properly for C cross-compiler (a special *.bat file could fit perfectly to this purpose) and your compile.bat has proper "MACHINE" parameter set, to, say, "wince-mips-pocket-wce300".

```
compile.bat
compile.bat dist
compile.bat CROSS_NAME=mips-wce300-thr "USE_ITHREADS=define" "USE_IMP_SYS=define" "USE_MULTI=define"
compile.bat CROSS_NAME=mips-wce300-thr "USE_ITHREADS=define" "USE_IMP_SYS=define" "USE_MULTI=define"
```

If all goes okay and no errors during a build, you'll get two independent distributions: "wince-mips-pocket-wce300" and "mips-wce300-thr".
Target 'dist' prepares distribution file set. Target 'zipdist' performs same as 'dist' but additionally compresses distribution files into zip archive.
NOTE: during a build there could be created a number (or one) of Config.pm for cross-compilation ("foreign" Config.pm) and those are hidden inside ../xlib/$(CROSS_NAME) with other auxilary files, but, and this is important to note, there should be *no* Config.pm for host miniperl. If you'll get an error that perl could not find Config.pm somewhere in building process this means something went wrong. Most probably you forgot to specify a cross-compilation when invoking miniperl.exe to Makefile.PL When building an extension for cross-compilation your command line should look like

```
..\miniperl.exe -I..\lib -MCross=mips-wce300-thr Makefile.PL
```

or just

```
..\miniperl.exe -I..\lib -MCross Makefile.PL
```

to refer a cross-compilation that was created last time.
If you decided to build with fcrypt.c file, please refer to README.win32 file, as long as all legal considerations and steps to do are exactly same in this case.
All questions related to building for WinCE devices could be asked in perlce-users@lists.sourceforge.net mailing list.

102.3 ACKNOWLEDGEMENTS

The port for Win32 was used as a reference.

102.4 AUTHORS

Rainer Keuchel (keuchel@netwave.de) Vadim Konovalov (vkonovalov@spb.lucent.com)

Chapter 103

README.cygwin

Perl for Cygwin

103.1 SYNOPSIS

This document will help you configure, make, test and install Perl on Cygwin. This document also describes features of Cygwin that will affect how Perl behaves at runtime.

NOTE: There are pre-built Perl packages available for Cygwin and a version of Perl is provided in the normal Cygwin install. If you do not need to customize the configuration, consider using one of those packages.

103.2 PREREQUISITES FOR COMPILING PERL ON CYGWIN

103.2.1 Cygwin = GNU+Cygnus+Windows (Don't leave UNIX without it)

The Cygwin tools are ports of the popular GNU development tools for Win32 platforms. They run thanks to the Cygwin library which provides the UNIX system calls and environment these programs expect. More information about this project can be found at:

```
  http://www.cygwin.com/
```

A recent net or commercial release of Cygwin is required.

At the time this document was last updated, Cygwin 1.5.2 was current.

103.2.2 Cygwin Configuration

While building Perl some changes may be necessary to your Cygwin setup so that Perl builds cleanly. These changes are **not** required for normal Perl usage.

NOTE: The binaries that are built will run on all Win32 versions. They do not depend on your host system (Win9x/WinME, WinNT/Win2K) or your Cygwin configuration (*ntea*, *ntsec*, binary/text mounts). The only dependencies come from hard-coded pathnames like `/usr/local`. However, your host system and Cygwin configuration will affect Perl's runtime behavior (see §**??**).

- `PATH`

  Set the `PATH` environment variable so that Configure finds the Cygwin versions of programs. Any Windows directories should be removed or moved to the end of your `PATH`.

- *nroff*

  If you do not have *nroff* (which is part of the *groff* package), Configure will **not** prompt you to install *man* pages.

- Permissions

  On WinNT with either the *ntea* or *ntsec* `CYGWIN` settings, directory and file permissions may not be set correctly. Since the build process creates directories and files, to be safe you may want to run a '`chmod -R +w *`' on the entire Perl source tree.

  Also, it is a well known WinNT "feature" that files created by a login that is a member of the *Administrators* group will be owned by the *Administrators* group. Depending on your umask, you may find that you can not write to files that you just created (because you are no longer the owner). When using the *ntsec* `CYGWIN` setting, this is not an issue because it "corrects" the ownership to what you would expect on a UNIX system.

103.3 CONFIGURE PERL ON CYGWIN

The default options gathered by Configure with the assistance of *hints/cygwin.sh* will build a Perl that supports dynamic loading (which requires a shared *libperl.dll*).

This will run Configure and keep a record:

```
./Configure 2>&1 | tee log.configure
```

If you are willing to accept all the defaults run Configure with **-de**. However, several useful customizations are available.

103.3.1 Stripping Perl Binaries on Cygwin

It is possible to strip the EXEs and DLLs created by the build process. The resulting binaries will be significantly smaller. If you want the binaries to be stripped, you can either add a **-s** option when Configure prompts you,

```
Any additional ld flags (NOT including libraries)? [none] -s
Any special flags to pass to gcc to use dynamic linking? [none] -s
Any special flags to pass to ld2 to create a dynamically loaded library?
[none] -s
```

or you can edit *hints/cygwin.sh* and uncomment the relevant variables near the end of the file.

103.3.2 Optional Libraries for Perl on Cygwin

Several Perl functions and modules depend on the existence of some optional libraries. Configure will find them if they are installed in one of the directories listed as being used for library searches. Pre-built packages for most of these are available from the Cygwin installer.

- `-lcrypt`

  The crypt package distributed with Cygwin is a Linux compatible 56-bit DES crypt port by Corinna Vinschen.

  Alternatively, the crypt libraries in GNU libc have been ported to Cygwin.

  The DES based Ultra Fast Crypt port was done by Alexey Truhan:

  ```
  ftp://ftp.uni-erlangen.de/pub/pc/gnuwin32/cygwin/porters/Okhapkin_Sergey/cw32crypt-dist-0.tgz
  ```

  NOTE: There are various export restrictions on DES implementations, see the glibc README for more details.

  The MD5 port was done by Andy Piper:

  ```
  ftp://ftp.uni-erlangen.de/pub/pc/gnuwin32/cygwin/porters/Okhapkin_Sergey/libcrypt.tgz
  ```

- `-lgdbm` (`use GDBM_File`)

  GDBM is available for Cygwin.

  NOTE: The GDBM library only works on NTFS partitions.

- `-ldb` (`use DB_File`)

  BerkeleyDB is available for Cygwin.

  NOTE: The BerkeleyDB library only completely works on NTFS partitions.

- `-lcygipc` (`use IPC::SysV`)

  A port of SysV IPC is available for Cygwin.

  NOTE: This has **not** been extensively tested. In particular, `d_semctl_semun` is undefined because it fails a Configure test and on Win9x the *shm*()* functions seem to hang. It also creates a compile time dependency because *perl.h* includes <*sys/ipc.h*> and <*sys/sem.h*> (which will be required in the future when compiling CPAN modules). CURRENTLY NOT SUPPORTED!

- `-lutil`

  Included with the standard Cygwin netrelease is the inetutils package which includes libutil.a.

103.3.3 Configure-time Options for Perl on Cygwin

The *INSTALL* document describes several Configure-time options. Some of these will work with Cygwin, others are not yet possible. Also, some of these are experimental. You can either select an option when Configure prompts you or you can define (undefine) symbols on the command line.

- `-Uusedl`

  Undefining this symbol forces Perl to be compiled statically.

- `-Uusemymalloc`

  By default Perl uses the `malloc()` included with the Perl source. If you want to force Perl to build with the system `malloc()` undefine this symbol.

- `-Uuseperlio`

  Undefining this symbol disables the PerlIO abstraction. PerlIO is now the default; it is not recommended to disable PerlIO.

- `-Dusemultiplicity`

  Multiplicity is required when embedding Perl in a C program and using more than one interpreter instance. This works with the Cygwin port.

- `-Duse64bitint`

  By default Perl uses 32 bit integers. If you want to use larger 64 bit integers, define this symbol.

- `-Duselongdouble`

  gcc supports long doubles (12 bytes). However, several additional long double math functions are necessary to use them within Perl (*{atan2, cos, exp, floor, fmod, frexp, isnan, log, modf, pow, sin, sqrt}l, strtold*). These are **not** yet available with Cygwin.

- `-Dusethreads`

  POSIX threads are implemented in Cygwin, define this symbol if you want a threaded perl.

- `-Duselargefiles`

  Cygwin uses 64-bit integers for internal size and position calculations, this will be correctly detected and defined by Configure.

- `-Dmksymlinks`

  Use this to build perl outside of the source tree. This works with Cygwin. Details can be found in the *INSTALL* document. This is the recommended way to build perl from sources.

103.3.4 Suspicious Warnings on Cygwin

You may see some messages during Configure that seem suspicious.

- *dlsym()*

  ld2 is needed to build dynamic libraries, but it does not exist when `dlsym()` checking occurs (it is not created until '`make`' runs). You will see the following message:

  ```
  Checking whether your C<dlsym()> needs a leading underscore ...
  ld2: not found
  I can't compile and run the test program.
  I'm guessing that dlsym doesn't need a leading underscore.
  ```

  Since the guess is correct, this is not a problem.

- Win9x and `d_eofnblk`

  Win9x does not correctly report `EOF` with a non-blocking read on a closed pipe. You will see the following messages:

  ```
  But it also returns -1 to signal EOF, so be careful!
  WARNING: you can't distinguish between EOF and no data!

  *** WHOA THERE!!! ***
      The recommended value for $d_eofnblk on this machine was "define"!
      Keep the recommended value? [y]
  ```

  At least for consistency with WinNT, you should keep the recommended value.

- Compiler/Preprocessor defines

  The following error occurs because of the Cygwin `#define` of `_LONG_DOUBLE`:

  ```
  Guessing which symbols your C compiler and preprocessor define...
  try.c:<line#>: missing binary operator
  ```

  This failure does not seem to cause any problems. With older gcc versions, "parse error" is reported instead of "missing binary operator".

103.4 MAKE ON CYGWIN

Simply run *make* and wait:

```
make 2>&1 | tee log.make
```

103.4.1 Errors on Cygwin

Errors like these are normal:

```
...
make: [extra.pods] Error 1 (ignored)
...
make: [extras.make] Error 1 (ignored)
```

103.4.2 ld2 on Cygwin

During '`make`', *ld2* will be created and installed in your $installbin directory (where you said to put public executables). It does not wait until the '`make install`' process to install the *ld2* script, this is because the remainder of the '`make`' refers to *ld2* without fully specifying its path and does this from multiple subdirectories. The assumption is that $installbin is in your current `PATH`. If this is not the case '`make`' will fail at some point. If this happens, just manually copy *ld2* from the source directory to somewhere in your `PATH`.

103.5 TEST ON CYGWIN

There are two steps to running the test suite:

```
make test 2>&1 | tee log.make-test
```

```
cd t;./perl harness 2>&1 | tee ../log.harness
```

The same tests are run both times, but more information is provided when running as '`./perl harness`'.

Test results vary depending on your host system and your Cygwin configuration. If a test can pass in some Cygwin setup, it is always attempted and explainable test failures are documented. It is possible for Perl to pass all the tests, but it is more likely that some tests will fail for one of the reasons listed below.

103.5.1 File Permissions on Cygwin

UNIX file permissions are based on sets of mode bits for {read,write,execute} for each {user,group,other}. By default Cygwin only tracks the Win32 read-only attribute represented as the UNIX file user write bit (files are always readable, files are executable if they have a *.{com,bat,exe}* extension or begin with `#!`, directories are always readable and executable). On WinNT with the *ntea* `CYGWIN` setting, the additional mode bits are stored as extended file attributes. On WinNT with the *ntsec* `CYGWIN` setting, permissions use the standard WinNT security descriptors and access control lists. Without one of these options, these tests will fail (listing not updated yet):

```
Failed Test           List of failed
------------------------------------
io/fs.t               5, 7, 9-10
lib/anydbm.t          2
lib/db-btree.t        20
lib/db-hash.t         16
lib/db-recno.t        18
lib/gdbm.t            2
lib/ndbm.t            2
lib/odbm.t            2
lib/sdbm.t            2
op/stat.t             9, 20 (.tmp not an executable extension)
```

103.5.2 NDBM_File and ODBM_File do not work on FAT filesystems

Do not use NDBM_File or ODBM_File on FAT filesystem. They can be built on a FAT filesystem, but many tests will fail:

```
../ext/NDBM_File/ndbm.t       13  3328    71   59  83.10%  1-2 4 16-71
../ext/ODBM_File/odbm.t      255 65280    ??   ??       %  ??
../lib/AnyDBM_File.t           2   512    12    2  16.67%  1 4
../lib/Memoize/t/errors.t      0   139    11    5  45.45%  7-11
../lib/Memoize/t/tie_ndbm.t   13  3328     4    4 100.00%  1-4
run/fresh_perl.t                          97    1   1.03%  91
```

If you intend to run only on FAT (or if using AnyDBM_File on FAT), run Configure with the -Ui_ndbm and -Ui_dbm options to prevent NDBM_File and ODBM_File being built.

With NTFS (and CYGWIN=ntsec), there should be no problems even if perl was built on FAT.

103.5.3 `fork()` failures in io_* tests

A `fork()` failure may result in the following tests failing:

```
ext/IO/lib/IO/t/io_multihomed.t
ext/IO/lib/IO/t/io_sock.t
ext/IO/lib/IO/t/io_unix.t
```

See comment on fork in *Miscellaneous* below.

103.5.4 Script Portability on Cygwin

Cygwin does an outstanding job of providing UNIX-like semantics on top of Win32 systems. However, in addition to the items noted above, there are some differences that you should know about. This is a very brief guide to portability, more information can be found in the Cygwin documentation.

- Pathnames

  Cygwin pathnames can be separated by forward (*/*) or backward (\\) slashes. They may also begin with drive letters (*C:*) or Universal Naming Codes (*//UNC*). DOS device names (*aux*, *con*, *prn*, *com**, *lpt?*, *nul*) are invalid as base filenames. However, they can be used in extensions (e.g., *hello.aux*). Names may contain all printable characters except these:

  ```
  : * ? " < > |
  ```

  File names are case insensitive, but case preserving. A pathname that contains a backslash or drive letter is a Win32 pathname (and not subject to the translations applied to POSIX style pathnames).

- Text/Binary

  When a file is opened it is in either text or binary mode. In text mode a file is subject to CR/LF/Ctrl-Z translations. With Cygwin, the default mode for an `open()` is determined by the mode of the mount that underlies the file. Perl provides a `binmode()` function to set binary mode on files that otherwise would be treated as text. `sysopen()` with the `O_TEXT` flag sets text mode on files that otherwise would be treated as binary:

  ```
      sysopen(FOO, "bar", O_WRONLY|O_CREAT|O_TEXT)
  ```

  `lseek()`, `tell()` and `sysseek()` only work with files opened in binary mode.

  The text/binary issue is covered at length in the Cygwin documentation.

- PerlIO

  PerlIO overrides the default Cygwin Text/Binary behaviour. A file will always treated as binary, regardless which mode of the mount it lives on, just like it is in UNIX. So CR/LF translation needs to be requested in either the `open()` call like this:

  ```
  open(FH, ">:crlf", "out.txt");
  ```

  which will do conversion from LF to CR/LF on the output, or in the environment settings (add this to your .bashrc):

  ```
  export PERLIO=crlf
  ```

  which will pull in the crlf PerlIO layer which does LF -> CRLF conversion on every output generated by perl.

- *.exe*

  The Cygwin `stat()`, `lstat()` and `readlink()` functions make the *.exe* extension transparent by looking for *foo.exe* when you ask for *foo* (unless a *foo* also exists). Cygwin does not require a *.exe* extension, but *gcc* adds it automatically when building a program. However, when accessing an executable as a normal file (e.g., *cp* in a makefile) the *.exe* is not transparent. The *install* included with Cygwin automatically appends a *.exe* when necessary.

- `chown()`

  On WinNT `chown()` can change a file's user and group IDs. On Win9x `chown()` is a no-op, although this is appropriate since there is no security model.

- Miscellaneous

  File locking using the `F_GETLK` command to `fcntl()` is a stub that returns `ENOSYS`.

  Win9x can not `rename()` an open file (although WinNT can).

  The Cygwin `chroot()` implementation has holes (it can not restrict file access by native Win32 programs).

  Inplace editing `perl -i` of files doesn't work without doing a backup of the file being edited `perl -i.bak` because of windowish restrictions, therefore Perl adds the suffix `.bak` automatically if you use `perl -i` without specifying a backup extension.

  Using `fork()` after loading multiple dlls may fail with an internal cygwin error like the following:

```
C:\CYGWIN\BIN\PERL.EXE: *** couldn't allocate memory 0x10000(4128768) for 'C:\CYGWIN\LIB\PERL5\5

  200 [main] perl 377147 sync_with_child: child -395691(0xB8) died before initialization with st
 1370 [main] perl 377147 sync_with_child: *** child state child loading dlls
```

  Use the rebase utility to resolve the conflicting dll addresses. The rebase package is included in the Cygwin netrelease. Use setup.exe from *http://www.cygwin.com/setup.exe* to install it and run rebaseall.

103.6 INSTALL PERL ON CYGWIN

This will install Perl, including *man* pages.

```
make install 2>&1 | tee log.make-install
```

NOTE: If `STDERR` is redirected '`make install`' will **not** prompt you to install *perl* into */usr/bin*.

You may need to be *Administrator* to run '`make install`'. If you are not, you must have write access to the directories in question.

Information on installing the Perl documentation in HTML format can be found in the *INSTALL* document.

103.7 MANIFEST ON CYGWIN

These are the files in the Perl release that contain references to Cygwin. These very brief notes attempt to explain the reason for all conditional code. Hopefully, keeping this up to date will allow the Cygwin port to be kept as clean as possible (listing not updated yet).

Documentation

```
INSTALL README.cygwin README.win32 MANIFEST
Changes Changes5.005 Changes5.004 Changes5.6
pod/perl.pod pod/perlport.pod pod/perlfaq3.pod
pod/perldelta.pod pod/perl5004delta.pod pod/perl56delta.pod
pod/perlhist.pod pod/perlmodlib.pod perl/buildtoc pod/perltoc.pod
```

Build, Configure, Make, Install

```
cygwin/Makefile.SHs
cygwin/ld2.in
cygwin/perlld.in
ext/IPC/SysV/hints/cygwin.pl
ext/NDBM_File/hints/cygwin.pl
ext/ODBM_File/hints/cygwin.pl
hints/cygwin.sh
Configure             - help finding hints from uname,
                        shared libperl required for dynamic loading
Makefile.SH           - linklibperl
Porting/patchls       - cygwin in port list
installman            - man pages with :: translated to .
installperl           - install dll/ld2/perlld, install to pods
makedepend.SH         - uwinfix
```

Tests

```
t/io/tell.t           - binmode
t/lib/b.t             - ignore Cwd from os_extras
t/lib/glob-basic.t    - Win32 directory list access differs from read mode
t/op/magic.t          - $^X/symlink WORKAROUND, s/.exe//
t/op/stat.t           - no /dev, skip Win32 ftCreationTime quirk
                        (cache manager sometimes preserves ctime of file
                        previously created and deleted), no -u (setuid)
```

Compiled Perl Source

```
EXTERN.h              - __declspec(dllimport)
XSUB.h                - __declspec(dllexport)
cygwin/cygwin.c       - os_extras (getcwd, spawn)
perl.c                - os_extras
perl.h                - binmode
doio.c                - win9x can not rename a file when it is open
pp_sys.c              - do not define h_errno, pp_system with spawn
util.c                - use setenv
```

Compiled Module Source

```
ext/POSIX/POSIX.xs    - tzname defined externally
ext/SDBM_File/sdbm/pair.c
                      - EXTCONST needs to be redefined from EXTERN.h
ext/SDBM_File/sdbm/sdbm.c
                      - binary open
```

Perl Modules/Scripts

```
lib/Cwd.pm            - hook to internal Cwd::cwd
lib/ExtUtils/MakeMaker.pm
                      - require MM_Cygwin.pm
lib/ExtUtils/MM_Cygwin.pm
                      - canonpath, cflags, manifypods, perl_archive
lib/File/Find.pm      - on remote drives stat() always sets st_nlink to 1
lib/File/Spec/Unix.pm - preserve //unc
lib/File/Temp.pm      - no directory sticky bit
lib/perl5db.pl        - use stdin not /dev/tty
utils/perldoc.PL      - version comment
```

103.8 BUGS ON CYGWIN

Support for swapping real and effective user and group IDs is incomplete. On WinNT Cygwin provides `setuid()`, `seteuid()`, `setgid()` and `setegid()`. However, additional Cygwin calls for manipulating WinNT access tokens and security contexts are required.

103.9 AUTHORS

Charles Wilson <cwilson@ece.gatech.edu>, Eric Fifer <egf7@columbia.edu>, alexander smishlajev <als@turnhere.com>, Steven Morlock <newspost@morlock.net>, Sebastien Barre <Sebastien.Barre@utc.fr>, Teun Burgers <burgers@ecn.nl>, Gerrit P. Haase <gp@familiehaase.de>.

103.10 HISTORY

Last updated: 2003-08-12

Chapter 104

perldgux

Perl under DG/UX.

104.1 SYNOPSIS

One can read this document in the following formats:

```
        man perldgux
        view perl perldgux
        explorer perldgux.html
        info perldgux
```

to list some (not all may be available simultaneously), or it may be read *as is*: as *README.dgux*.

104.2 DESCRIPTION

Perl 5.7/8.x for DG/UX ix86 R4.20MU0x

104.3 BUILDING PERL ON DG/UX

104.3.1 Non-threaded Perl on DG/UX

Just run ./Configure script from the top directory. Then give "make" to compile.

104.3.2 Threaded Perl on DG/UX

If you are using as compiler GCC-2.95.x rev(DG/UX) an easy solution for configuring perl in your DG/UX machine is to run the command:
./Configure -Dusethreads -Duseithreads -Dusedevel -des
This will automatically accept all the defaults and in particular /usr/local/ as installation directory. Note that GCC-2.95.x rev(DG/UX) knows the switch -pthread which allows it to link correctly DG/UX's -lthread library.
If you want to change the installation directory or have a standard DG/UX with C compiler GCC-2.7.2.x then you have no choice than to do an interactive build by issuing the command:
./Configure -Dusethreads -Duseithreads
In particular with GCC-2.7.2.x accept all the defaults and *watch* out for the message:

```
    Any additional ld flags (NOT including libraries)? [ -pthread]
```

Instead of -pthread put here -lthread. CGCC-2.7.2.x that comes with the DG/UX OS does NOT know the -pthread switch. So your build will fail if you choose the defaults. After configuration is done correctly give "make" to compile.

104.3.3 Testing Perl on DG/UX

Issuing a "make test" will run all the tests. If the test lib/ftmp-security gives you as a result something like

```
    lib/ftmp-security....File::Temp::_gettemp:
    Parent directory (/tmp/) is not safe (sticky bit not set
    when world writable?) at lib/ftmp-security.t line 100
```

don't panic and just set the sticky bit in your /tmp directory by doing the following as root:

cd / chmod +t /tmp (=set the sticky bit to /tmp).

Then rerun the tests. This time all must be OK.

104.3.4 Installing the built perl on DG/UX

Run the command "make install"

104.4 AUTHOR

Takis Psarogiannakopoulos Universirty of Cambridge Centre for Mathematical Sciences Department of Pure Mathematics Wilberforce road Cambridge CB3 0WB , UK email <takis@XFree86.Org>

104.5 SEE ALSO

perl(1).

Chapter 105

perldos

Perl under DOS, W31, W95.

105.1 SYNOPSIS

These are instructions for building Perl under DOS (or w??), using DJGPP v2.03 or later. Under w95 long filenames are supported.

105.2 DESCRIPTION

Before you start, you should glance through the README file found in the top-level directory where the Perl distribution was extracted. Make sure you read and understand the terms under which this software is being distributed.

This port currently supports MakeMaker (the set of modules that is used to build extensions to perl). Therefore, you should be able to build and install most extensions found in the CPAN sites.

Detailed instructions on how to build and install perl extension modules, including XS-type modules, is included. See 'BUILDING AND INSTALLING MODULES'.

105.2.1 Prerequisites for Compiling Perl on DOS

DJGPP

DJGPP is a port of GNU C/C++ compiler and development tools to 32-bit, protected-mode environment on Intel 32-bit CPUs running MS-DOS and compatible operating systems, by DJ Delorie <dj@delorie.com> and friends.

For more details (FAQ), check out the home of DJGPP at:

```
    http://www.delorie.com/djgpp/
```

If you have questions about DJGPP, try posting to the DJGPP newsgroup: comp.os.msdos.djgpp, or use the email gateway djgpp@delorie.com.

You can find the full DJGPP distribution on any SimTel.Net mirror all over the world. Like:

```
    ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2*
```

You need the following files to build perl (or add new modules):

```
    v2/djdev203.zip
    v2gnu/bnu2112b.zip
    v2gnu/gcc2953b.zip
    v2gnu/bsh204b.zip
    v2gnu/mak3791b.zip
    v2gnu/fil40b.zip
    v2gnu/sed3028b.zip
    v2gnu/txt20b.zip
    v2gnu/dif272b.zip
    v2gnu/grep24b.zip
    v2gnu/shl20jb.zip
    v2gnu/gwk306b.zip
    v2misc/csdpmi5b.zip
```

or possibly any newer version.

Pthreads

Thread support is not tested in this version of the djgpp perl.

105.2.2 Shortcomings of Perl under DOS

Perl under DOS lacks some features of perl under UNIX because of deficiencies in the UNIX-emulation, most notably:

- fork() and pipe()
- some features of the UNIX filesystem regarding link count and file dates
- in-place operation is a little bit broken with short filenames
- sockets

105.2.3 Building Perl on DOS

- Unpack the source package *perl5.8*.tar.gz* with djtarx. If you want to use long file names under w95 and also to get Perl to pass all its tests, don't forget to use

  ```
  set LFN=y
  set FNCASE=y
  ```

  before unpacking the archive.

- Create a "symlink" or copy your bash.exe to sh.exe in your `($DJDIR)/bin` directory.

  ```
  ln -s bash.exe sh.exe
  ```

  [If you have the recommended version of bash for DJGPP, this is already done for you.]

  And make the `SHELL` environment variable point to this *sh.exe*:

  ```
  set SHELL=c:/djgpp/bin/sh.exe (use full path name!)
  ```

  You can do this in *djgpp.env* too. Add this line BEFORE any section definition:

  ```
  +SHELL=%DJDIR%/bin/sh.exe
  ```

- If you have *split.exe* and *gsplit.exe* in your path, then rename *split.exe* to *djsplit.exe*, and *gsplit.exe* to *split.exe*. Copy or link *gecho.exe* to *echo.exe* if you don't have *echo.exe*. Copy or link *gawk.exe* to *awk.exe* if you don't have *awk.exe*.

  [If you have the recommended versions of djdev, shell utilities and gawk, all these are already done for you, and you will not need to do anything.]

- Chdir to the djgpp subdirectory of perl toplevel and type the following commands:

  ```
  set FNCASE=y
  configure.bat
  ```

  This will do some preprocessing then run the Configure script for you. The Configure script is interactive, but in most cases you just need to press ENTER. The "set" command ensures that DJGPP preserves the letter case of file names when reading directories. If you already issued this set command when unpacking the archive, and you are in the same DOS session as when you unpacked the archive, you don't have to issue the set command again. This command is necessary *before* you start to (re)configure or (re)build perl in order to ensure both that perl builds correctly and that building XS-type modules can succeed. See the DJGPP info entry for "_preserve_fncase" for more information:

  ```
  info libc alphabetical _preserve_fncase
  ```

  If the script says that your package is incomplete, and asks whether to continue, just answer with Y (this can only happen if you don't use long filenames or forget to issue "set FNCASE=y" first).

  When Configure asks about the extensions, I suggest IO and Fcntl, and if you want database handling then SDBM_File or GDBM_File (you need to install gdbm for this one). If you want to use the POSIX extension (this is the default), make sure that the stack size of your *cc1.exe* is at least 512kbyte (you can check this with: `stubedit cc1.exe`).

  You can use the Configure script in non-interactive mode too. When I built my *perl.exe*, I used something like this:

  ```
  configure.bat -des
  ```

  You can find more info about Configure's command line switches in the *INSTALL* file.

  When the script ends, and you want to change some values in the generated *config.sh* file, then run

  ```
  sh Configure -S
  ```

  after you made your modifications.

  IMPORTANT: if you use this `-S` switch, be sure to delete the CONFIG environment variable before running the script:

  ```
  set CONFIG=
  ```

- Now you can compile Perl. Type:

  ```
  make
  ```

105.2.4 Testing Perl on DOS

Type:

```
make test
```

If you're lucky you should see "All tests successful". But there can be a few failed subtests (less than 5 hopefully) depending on some external conditions (e.g. some subtests fail under linux/dosemu or plain dos with short filenames only).

105.2.5 Installation of Perl on DOS

Type:

```
        make install
```

This will copy the newly compiled perl and libraries into your DJGPP directory structure. Perl.exe and the utilities go into `($DJDIR)/bin`, and the library goes under `($DJDIR)/lib/perl5`. The pod documentation goes under `($DJDIR)/lib/perl5/pod`.

105.3 BUILDING AND INSTALLING MODULES ON DOS

105.3.1 Building Prerequisites for Perl on DOS

For building and installing non-XS modules, all you need is a working perl under DJGPP. Non-XS modules do not require re-linking the perl binary, and so are simpler to build and install.

XS-type modules do require re-linking the perl binary, because part of an XS module is written in "C", and has to be linked together with the perl binary to be executed. This is required because perl under DJGPP is built with the "static link" option, due to the lack of "dynamic linking" in the DJGPP environment.

Because XS modules require re-linking of the perl binary, you need both the perl binary distribution and the perl source distribution to build an XS extension module. In addition, you will have to have built your perl binary from the source distribution so that all of the components of the perl binary are available for the required link step.

105.3.2 Unpacking CPAN Modules on DOS

First, download the module package from CPAN (e.g., the "Comma Separated Value" text package, Text-CSV-0.01.tar.gz). Then expand the contents of the package into some location on your disk. Most CPAN modules are built with an internal directory structure, so it is usually safe to expand it in the root of your DJGPP installation. Some people prefer to locate source trees under /usr/src (i.e., `($DJDIR)/usr/src`), but you may put it wherever seems most logical to you, *EXCEPT* under the same directory as your perl source code. There are special rules that apply to modules which live in the perl source tree that do not apply to most of the modules in CPAN.

Unlike other DJGPP packages, which are normal "zip" files, most CPAN module packages are "gzipped tarballs". Recent versions of WinZip will safely unpack and expand them, *UNLESS* they have zero-length files. It is a known WinZip bug (as of v7.0) that it will not extract zero-length files.

From the command line, you can use the djtar utility provided with DJGPP to unpack and expand these files. For example:

```
        C:\djgpp>djtarx -v Text-CSV-0.01.tar.gz
```

This will create the new directory `($DJDIR)/Text-CSV-0.01`, filling it with the source for this module.

105.3.3 Building Non-XS Modules on DOS

To build a non-XS module, you can use the standard module-building instructions distributed with perl modules.

```
    perl Makefile.PL
    make
    make test
    make install
```

This is sufficient because non-XS modules install only ".pm" files and (sometimes) pod and/or man documentation. No re-linking of the perl binary is needed to build, install or use non-XS modules.

105.3.4 Building XS Modules on DOS

To build an XS module, you must use the standard module-building instructions distributed with perl modules *PLUS* three extra instructions specific to the DJGPP "static link" build environment.

```
    set FNCASE=y
    perl Makefile.PL
    make
    make perl
    make test
    make -f Makefile.aperl inst_perl MAP_TARGET=perl.exe
    make install
```

The first extra instruction sets DJGPP's FNCASE environment variable so that the new perl binary which you must build for an XS-type module will build correctly. The second extra instruction re-builds the perl binary in your module directory before you run "make test", so that you are testing with the new module code you built with "make". The third extra instruction installs the perl binary from your module directory into the standard DJGPP binary directory, `($DJDIR)/bin`, replacing your previous perl binary.

Note that the MAP_TARGET value *must* have the ".exe" extension or you will not create a "perl.exe" to replace the one in `($DJDIR)/bin`.

When you are done, the XS-module install process will have added information to your "perllocal" information telling that the perl binary has been replaced, and what module was installed. You can view this information at any time by using the command:

```
        perl -S perldoc perllocal
```

105.4 AUTHOR

Laszlo Molnar, *laszlo.molnar@eth.ericsson.se* [Installing/building perl]

Peter J. Farley III *pjfarley@banet.net* [Building/installing modules]

105.5 SEE ALSO

perl(1).

Chapter 106

README.epoc

Perl for EPOC

106.1 SYNOPSIS

Perl 5 README file for the EPOC Release 5 operating system.

106.2 INTRODUCTION

EPOC is an OS for palmtops and mobile phones. For more informations look at: http://www.symbian.com/

This is a port of perl to the epocemx SDK by Eberhard Mattes, which itself uses the SDK by symbian. Essentially epocemx it is a POSIX look alike environment for the EPOC OS. For more information look at: http://epocemx.sourceforge.net/

perl and epocemx runs on Epoc Release 5 machines: Psion 5mx, 5mx Pro, Psion Revo, Psion Netbook and on the Ericson M128. It may run on Epoc Release 3 Hardware (Series 5 classic), too. For more information about this hardware please refer to http://www.psion.com/

Vendors which like to have support for their devices are free to send me a sample.

106.3 INSTALLING PERL ON EPOC

You can download a ready-to-install version from http://www.oflebbe.de/oflebbe/perl/

You will need at least ˜6MB free space in order to install and run perl.

Please install the emxusr.sis package from http://epocemx.sourceforge.net/ first.

Install perl.sis on the EPOC machine. If you do not know how to do that, consult your PsiWin documentation.

Perl itself and its standard library is using 4 MB disk space. Unicode support and some other modules are left out. (For details, please look into epoc/createpkg.pl). If you like to use these modules, you are free to copy them from a current perl release.

106.4 STARTING PERL ON EPOC

Please use the epocemx shell to start perl. perl integrates with the conventions of epocemx.

106.4.1 Editors on Epoc

A suitable text editor can be downloaded from symbian http://www.symbian.com/developer/downloads/files/editor.zip

106.4.2 Features of Perl on Epoc

The built-in function EPOC::getcwd returns the current directory.

106.4.3 Restrictions of Perl on Epoc

Features are left out, because of restrictions of the POSIX support in EPOC:

- socket IO is only implemented poorly. You can only use sysread and syswrite on them. The commands read, write, print, <> do not work for sockets. This may change iff epocemx supports sockets.
- kill, alarm and signals. Do not try to use them. This may be impossible to implement on EPOC.
- select is missing.
- binmode does not exist. (No CR LF to LF translation for text files)
- EPOC does not handle the notion of current drive and current directory very well (i.e. not at all, but it tries hard to emulate one). See PATH.
- Heap is limited to 4MB.
- Dynamic loading is not implemented.

106.4.4 Compiling Perl 5 on the EPOC cross compiling environment

Sorry, this is far too short.

- You will need the epocemx SDK from Eberhard Mattes.
- Get the Perl sources from your nearest CPAN site.
- Unpack the sources.
- Build a native perl from this sources... Make sure to save the miniperl executable as miniperl.native.
  Start again from scratch

```
cp epoc/* .
./Configure -S
make
cp miniperl.native miniperl
touch miniperl.exe
make
perl createpkg.pl

emxsis perl.pkg perl.sis
```

106.5 SUPPORT STATUS OF PERL ON EPOC

I’m offering this port "as is". You can ask me questions, but I can’t guarantee I’ll be able to answer them. Since the port to epocemx is quite new, please check the web for updates first.
Very special thanks to Eberhard Mattes for epocemx.

106.6 AUTHOR

Olaf Flebbe <olaf@oflebbe.de> http://www.oflebbe.de/oflebbe/perl/

106.7 LAST UPDATE

2003-01-18

Chapter 107

README.freebsd

Perl version 5 on FreeBSD systems

107.1 DESCRIPTION

This document describes various features of FreeBSD that will affect how Perl version 5 (hereafter just Perl) is compiled and/or runs.

107.1.1 FreeBSD core dumps from readdir_r with ithreads

When perl is configured to use ithreads, it will use re-entrant library calls in preference to non-re-entrant versions. There is a bug in FreeBSD's `readdir_r` function in versions 4.5 and earlier that can cause a SEGV when reading large directories. A patch for FreeBSD libc is available (see http://www.freebsd.org/cgi/query-pr.cgi?pr=misc/30631) which has been integrated into FreeBSD 4.6.

107.1.2 $ ^X doesn't always contain a full path in FreeBSD

perl 5.8.0 sets `$^X` where possible to a full path by asking the operating system. On FreeBSD the full path of the perl interpreter is found by reading the symlink */proc/curproc/file*. There is a bug on FreeBSD, where the result of reading this symlink is can be wrong in certain circumstances (see http://www.freebsd.org/cgi/query-pr.cgi?pr=35703). In these cases perl will fall back to the old behaviour of using C's argv[0] value for `$^X`.

107.1.3 Perl will no longer be part of "base FreeBSD"

Not as bad as it sounds–what this means is that Perl will no longer be part of the **kernel build system** of FreeBSD. Perl will still very probably be part of the "default install", and in any case the latest version will be in the ports system. The first FreeBSD version this change will affect is 5.0, all 4.n versions will keep the status quo.

107.2 AUTHOR

Nicholas Clark <nick@ccl4.org>, collating wisdom supplied by Slaven Rezic and Tim Bunce.

Please report any errors, updates, or suggestions to *perlbug@perl.org*.

Chapter 108

README.hpux

Perl version 5 on Hewlett-Packard Unix (HP-UX) systems

108.1 DESCRIPTION

This document describes various features of HP's Unix operating system (HP-UX) that will affect how Perl version 5 (hereafter just Perl) is compiled and/or runs.

108.1.1 Using perl as shipped with HP-UX

As of application release September 2001, HP-UX 11.00 is shipped with perl-5.6.1 in /opt/perl. The first occurrence is on CD 5012-7954 and can be installed using

```
    swinstall -s /cdrom perl
```

assuming you have mounted that CD on /cdrom. In this version the following modules are installed:

```
    ActivePerl::DocTools-0.04   HTML::Parser-3.19   XML::DOM-1.25
    Archive::Tar-0.072          HTML::Tagset-3.03   XML::Parser-2.27
    Compress::Zlib-1.08         MIME::Base64-2.11   XML::Simple-1.05
    Convert::ASN1-0.10          Net-1.07            XML::XPath-1.09
    Digest::MD5-2.11            PPM-2.1.5           XML::XSLT-0.32
    File::CounterFile-0.12      SOAP::Lite-0.46     libwww-perl-5.51
    Font::AFM-1.18              Storable-1.011      libxml-perl-0.07
    HTML-Tree-3.11              URI-1.11            perl-ldap-0.23
```

The build is a portable hppa-1.1 multithread build that supports large files compiled with gcc-2.9-hppa-991112

If you perform a new installation, then Perl will be installed automatically.

108.1.2 Using perl from HP's porting centre

HP porting centre tries very hard to keep up with customer demand and release updates from the Open Source community. Having precompiled Perl binaries available is obvious.

The HP porting centres are limited in what systems they are allowed to port to and they usually choose the two most recent OS versions available. This means that at the moment of writing, there are only HPUX-11.00 and 11-20/22 (IA64) ports available on the porting centres.

HP has asked the porting centre to move Open Source binaries from /opt to /usr/local, so binaries produced since the start of July 2002 are located in /usr/local.

One of HP porting centres URL's is http://hpux.connect.org.uk/ The port currently available is built with GNU gcc.

108.1.3 Compiling Perl 5 on HP-UX

When compiling Perl, you must use an ANSI C compiler. The C compiler that ships with all HP-UX systems is a K&R compiler that should only be used to build new kernels.

Perl can be compiled with either HP's ANSI C compiler or with gcc. The former is recommended, as not only can it compile Perl with no difficulty, but also can take advantage of features listed later that require the use of HP compiler-specific command-line flags.

If you decide to use gcc, make sure your installation is recent and complete, and be sure to read the Perl INSTALL file for more gcc-specific details.

108.1.4 PA-RISC

HP's current Unix systems run on its own Precision Architecture (PA-RISC) chip. HP-UX used to run on the Motorola MC68000 family of chips, but any machine with this chip in it is quite obsolete and this document will not attempt to address issues for compiling Perl on the Motorola chipset.

The most recent version of PA-RISC at the time of this document's last update is 2.0.

A complete list of models at the time the OS was built is in the file /usr/sam/lib/mo/sched.models. The first column corresponds to the last part of the output of the "model" command. The second column is the PA-RISC version and the third column is the exact chip type used. (Start browsing at the bottom to prevent confusion ;-)

```
    # model
    9000/800/L1000-44
    # grep L1000-44 /usr/sam/lib/mo/sched.models
    L1000-44        2.0     PA8500
```

108.1.5 PA-RISC 1.0

The original version of PA-RISC, HP no longer sells any system with this chip.

The following systems contained PA-RISC 1.0 chips:

```
    600, 635, 645, 808, 815, 822, 825, 832, 834, 835, 840, 842, 845, 850,
    852, 855, 860, 865, 870, 890
```

108.1.6 PA-RISC 1.1

An upgrade to the PA-RISC design, it shipped for many years in many different system.

The following systems contain with PA-RISC 1.1 chips:

```
    705, 710, 712, 715, 720, 722, 725, 728, 730, 735, 742, 743, 744, 745,
    747, 750, 755, 770, 777, 778, 779, 800, 801, 803, 806, 807, 809, 811,
    813, 816, 817, 819, 821, 826, 827, 829, 831, 837, 839, 841, 847, 849,
    851, 856, 857, 859, 867, 869, 877, 887, 891, 892, 897, A180, A180C,
    B115, B120, B132L, B132L+, B160L, B180L, C100, C110, C115, C120,
    C160L, D200, D210, D220, D230, D250, D260, D310, D320, D330, D350,
    D360, D410, DX0, DX5, DXO, E25, E35, E45, E55, F10, F20, F30, G30,
    G40, G50, G60, G70, H20, H30, H40, H50, H60, H70, I30, I40, I50, I60,
    I70, J200, J210, J210XC, K100, K200, K210, K220, K230, K400, K410,
    K420, S700i, S715, S744, S760, T500, T520
```

108.1.7 PA-RISC 2.0

The most recent upgrade to the PA-RISC design, it added support for 64-bit integer data.

As of the date of this document's last update, the following systems contain PA-RISC 2.0 chips:

```
700, 780, 781, 782, 783, 785, 802, 804, 810, 820, 861, 871, 879, 889,
893, 895, 896, 898, 899, A400, A500, B1000, B2000, C130, C140, C160,
C180, C180+, C180-XP, C200+, C400+, C3000, C360, C3600, CB260, D270,
D280, D370, D380, D390, D650, J220, J2240, J280, J282, J400, J410,
J5000, J5500XM, J5600, J7000, J7600, K250, K260, K260-EG, K270, K360,
K370, K380, K450, K460, K460-EG, K460-XP, K470, K570, K580, L1000,
L2000, L3000, N4000, R380, R390, SD16000, SD32000, SD64000, T540,
T600, V2000, V2200, V2250, V2500, V2600
```

Just before HP took over Compaq, some systems were renamed. the link that contained the explanation is dead, so here's a short summary:

```
HP 9000 A-Class servers, now renamed HP Server rp2400 series.
HP 9000 L-Class servers, now renamed HP Server rp5400 series.
HP 9000 N-Class servers, now renamed HP Server rp7400.

rp2400, rp2405, rp2430, rp2450, rp2470, rp3410, rp3440, rp5400,
rp5405, rp5430, rp5450, rp5470, rp7400, rp7405, rp7410, rp7420,
rp8400, rp8420, Superdome
```

The current naming convention is:

```
aadddd
||||`+- 00 - 99 relative capacity & newness (upgrades, etc.)
|||`--- unique number for each architecture to ensure different
|||     systems do not have the same numbering across
|||     architectures
||`---- 1 - 9 identifies family and/or relative positioning
||
|`----- c = ia32 (cisc)
|       p = pa-risc
|       x = ia-64 (Itanium & Itanium 2)
|       h = housing
`------ t = tower
        r = rack optimized
        s = super scalable
        b = blade
        sa = appliance
```

108.1.8 Itanium & Itanium 2

HP also ships servers with the 128-bit Itanium processor(s). As of the date of this document's last update, the following systems contain Itanium or Itanium 2 chips (this is very likely to be out of date):

```
rx1600, rx2600, rx2600hptc, rx4610, rx4640, rx5670, rx7620, rx8620,
rx9610
```

To see all about your machine, type

```
# model
ia64 hp server rx2600
# /usr/contrib/bin/machinfo
```

108.1.9 Portability Between PA-RISC Versions

An executable compiled on a PA-RISC 2.0 platform will not execute on a PA-RISC 1.1 platform, even if they are running the same version of HP-UX. If you are building Perl on a PA-RISC 2.0 platform and want that Perl to also run on a PA-RISC 1.1, the compiler flags +DAportable and +DS32 should be used.

It is no longer possible to compile PA-RISC 1.0 executables on either the PA-RISC 1.1 or 2.0 platforms. The command-line flags are accepted, but the resulting executable will not run when transferred to a PA-RISC 1.0 system.

108.1.10 Itanium Processor Family and HP-UX

HP-UX also runs on the new Itanium processor. This requires the use of a different version of HP-UX (currently 11.23 or 11i v1.6), and with the exception of a few differences detailed below and in later sections, Perl should compile with no problems.

Although PA-RISC binaries can run on Itanium systems, you should not attempt to use a PA-RISC version of Perl on an Itanium system. This is because shared libraries created on an Itanium system cannot be loaded while running a PA-RISC executable.

108.1.11 Building Dynamic Extensions on HP-UX

HP-UX supports dynamically loadable libraries (shared libraries). Shared libraries end with the suffix .sl. On Itanium systems, they end with the suffix .so.

Shared libraries created on a platform using a particular PA-RISC version are not usable on platforms using an earlier PA-RISC version by default. However, this backwards compatibility may be enabled using the same +DAportable compiler flag (with the same PA-RISC 1.0 caveat mentioned above).

Shared libraries created on an Itanium platform cannot be loaded on a PA-RISC platform. Shared libraries created on a PA-RISC platform can only be loaded on an Itanium platform if it is a PA-RISC executable that is attempting to load the PA-RISC library. A PA-RISC shared library cannot be loaded into an Itanium executable nor vice-versa.

To create a shared library, the following steps must be performed:

```
    1. Compile source modules with +z or +Z flag to create a .o module
       which contains Position-Independent Code (PIC).  The linker will
       tell you in the next step if +Z was needed.
       (For gcc, the appropriate flag is -fpic or -fPIC.)

    2. Link the shared library using the -b flag.  If the code calls
       any functions in other system libraries (e.g., libm), it must
       be included on this line.
```

(Note that these steps are usually handled automatically by the extension’s Makefile).

If these dependent libraries are not listed at shared library creation time, you will get fatal "Unresolved symbol" errors at run time when the library is loaded.

You may create a shared library that refers to another library, which may be either an archive library or a shared library. If this second library is a shared library, this is called a "dependent library". The dependent library’s name is recorded in the main shared library, but it is not linked into the shared library. Instead, it is loaded when the main shared library is loaded. This can cause problems if you build an extension on one system and move it to another system where the libraries may not be located in the same place as on the first system.

If the referred library is an archive library, then it is treated as a simple collection of .o modules (all of which must contain PIC). These modules are then linked into the shared library.

Note that it is okay to create a library which contains a dependent library that is already linked into perl.

Some extensions, like DB_File and Compress::Zlib use/require prebuilt libraries for the perl extensions/modules to work. If these libraries are built using the default configuration, it might happen that you run into an error like "invalid loader fixup" during load phase. HP is aware of this problem. Search the HP-UX cxx-dev forums for discussions about the subject. The short answer is that **everything** (all libraries, everything) must be compiled with `+z` or `+Z` to be PIC (position independent code). (For gcc, that would be `-fpic` or `-fPIC`). In HP-UX 11.00 or newer the linker error message should tell the name of the offending object file.

A more general approach is to intervene manually, as with an example for the DB_File module, which requires SleepyCat's libdb.sl:

```
# cd .../db-3.2.9/build_unix
# vi Makefile
... add +Z to all cflags to create shared objects
CFLAGS=         -c $(CPPFLAGS) +Z -Ae +O2 +Onolimit \
                -I/usr/local/include -I/usr/include/X11R6
CXXFLAGS=       -c $(CPPFLAGS) +Z -Ae +O2 +Onolimit \
                -I/usr/local/include -I/usr/include/X11R6

# make clean
# make
# mkdir tmp
# cd tmp
# ar x ../libdb.a
# ld -b -o libdb-3.2.sl *.o
# mv libdb-3.2.sl /usr/local/lib
# rm *.o
# cd /usr/local/lib
# rm -f libdb.sl
# ln -s libdb-3.2.sl libdb.sl

# cd .../DB_File-1.76
# make distclean
# perl Makefile.PL
# make
# make test
# make install
```

It is no longer possible to link PA-RISC 1.0 shared libraries (even though the command-line flags are still present).

PA-RISC and Itanium object files are not interchangeable. Although you may be able to use ar to create an archive library of PA-RISC object files on an Itanium system, you cannot link against it using an Itanium link editor.

108.1.12 The HP ANSI C Compiler

When using this compiler to build Perl, you should make sure that the flag -Aa is added to the cpprun and cppstdin variables in the config.sh file (though see the section on 64-bit perl below). If you are using a recent version of the Perl distribution, these flags are set automatically.

108.1.13 The GNU C Compiler

When you are going to use the GNU C compiler (gcc), and you don't have gcc yet, you can either build it yourself from the sources (available from e.g. http://www.gnu.ai.mit.edu/software/gcc/releases.html) or fetch a prebuilt binary from the HP porting center. There are two places where gcc prebuilds can be fetched; the first and best (for HP-UX 11 only) is http://h21007.www2.hp.com/dspp/tech/tech_TechSoftwareDetailPage_IDX/1,1703,547,00.html the second is http://hpux.cs.utah.edu/hppd/hpux/Gnu/ where you can also find the GNU binutils package. (Browse through the list, because there are often multiple versions of the same package available).

Above mentioned distributions are depots. H.Merijn Brand has made prebuilt gcc binaries available on http://mirrors.develooper.com/hpux/ and/or http://www.cmve.net/˜merijn/ for HP-UX 10.20, HP-UX 11.00, and HP-UX

11.11 (HP-UX 11i) in both 32- and 64-bit versions. These are bzipped tar archives that also include recent GNU binutils and GNU gdb. Read the instructions on that page to rebuild gcc using itself.

On PA-RISC you need a different compiler for 32-bit applications and for 64-bit applications. On PA-RISC, 32-bit objects and 64-bit objects do not mix. period. There is no different behaviour for HP C-ANSI-C or GNU gcc. So if you require your perl binary to use 64-bit libraries, like Oracle-64bit, you MUST build a 64-bit perl.

Building a 64-bit capable gcc on PA-RISC from source is possible only when you have the HP C-ANSI C compiler or an already working 64-bit binary of gcc available. Best performance for perl is achieved with HP's native compiler.

108.1.14 Using Large Files with Perl on HP-UX

Beginning with HP-UX version 10.20, files larger than 2GB (2ˆ31 bytes) may be created and manipulated. Three separate methods of doing this are available. Of these methods, the best method for Perl is to compile using the -Duselargefiles flag to Configure. This causes Perl to be compiled using structures and functions in which these are 64 bits wide, rather than 32 bits wide. (Note that this will only work with HP's ANSI C compiler. If you want to compile Perl using gcc, you will have to get a version of the compiler that supports 64-bit operations. See above for where to find it.)

There are some drawbacks to this approach. One is that any extension which calls any file-manipulating C function will need to be recompiled (just follow the usual "perl Makefile.PL; make; make test; make install" procedure).

The list of functions that will need to recompiled is: creat, fgetpos, fopen, freopen, fsetpos, fstat, fstatvfs, fstatvfsdev, ftruncate, ftw, lockf, lseek, lstat, mmap, nftw, open, prealloc, stat, statvfs, statvfsdev, tmpfile, truncate, getrlimit, setrlimit

Another drawback is only valid for Perl versions before 5.6.0. This drawback is that the seek and tell functions (both the builtin version and POSIX module version) will not perform correctly.

It is strongly recommended that you use this flag when you run Configure. If you do not do this, but later answer the question about large files when Configure asks you, you may get a configuration that cannot be compiled, or that does not function as expected.

108.1.15 Threaded Perl on HP-UX

It is possible to compile a version of threaded Perl on any version of HP-UX before 10.30, but it is strongly suggested that you be running on HP-UX 11.00 at least.

To compile Perl with threads, add -Dusethreads to the arguments of Configure. Verify that the -D_POSIX_C_SOURCE=199506L compiler flag is automatically added to the list of flags. Also make sure that -lpthread is listed before -lc in the list of libraries to link Perl with. The hints provided for HP-UX during Configure will try very hard to get this right for you.

HP-UX versions before 10.30 require a separate installation of a POSIX threads library package. Two examples are the HP DCE package, available on "HP-UX Hardware Extensions 3.0, Install and Core OS, Release 10.20, April 1999 (B3920-13941)" or the Freely available PTH package, available though worldwide HP-UX mirrors of precompiled packages (e.g. http://hpux.tn.tudelft.nl/hppd/hpux/)

If you are going to use the HP DCE package, the library used for threading is /usr/lib/libcma.sl, but there have been multiple updates of that library over time. Perl will build with the first version, but it will not pass the test suite. Older Oracle versions might be a compelling reason not to update that library, otherwise please find a newer version in one of the following patches: PHSS_19739, PHSS_20608, or PHSS_23672

reformatted output:

```
d3:/usr/lib 106 > what libcma-*.1
libcma-00000.1:
   HP DCE/9000 1.5               Module: libcma.sl (Export)
                                 Date: Apr 29 1996 22:11:24
libcma-19739.1:
   HP DCE/9000 1.5 PHSS_19739-40 Module: libcma.sl (Export)
                                 Date: Sep  4 1999 01:59:07
libcma-20608.1:
   HP DCE/9000 1.5 PHSS_20608    Module: libcma.1 (Export)
                                 Date: Dec  8 1999 18:41:23
libcma-23672.1:
   HP DCE/9000 1.5 PHSS_23672    Module: libcma.1 (Export)
                                 Date: Apr  9 2001 10:01:06
d3:/usr/lib 107 >
```

108.1.16 64-bit Perl on HP-UX

Beginning with HP-UX 11.00, programs compiled under HP-UX can take advantage of the LP64 programming environment (LP64 means Longs and Pointers are 64 bits wide).

Work is being performed on Perl to make it 64-bit compliant on all versions of Unix. Once this is complete, scalar variables will be able to hold numbers larger than 2^32 with complete precision.

As of the date of this document, Perl is fully 64-bit compliant on HP-UX 11.00 and up for both cc- and gcc builds. If you are about to build a 64-bit perl with GNU gcc, please read the gcc section carefully.

Should a user wish to experiment with compiling Perl in the LP64 environment, use the -Duse64bitall flag to Configure. This will force Perl to be compiled in a pure LP64 environment (with the +DD64 flag for HP C-ANSI-C, with no additional options for GNU gcc 64-bit on PA-RISC, and with -mlp64 for GNU gcc on Itanium). If you want to compile Perl using gcc, you will have to get a version of the compiler that supports 64-bit operations.)

You can also use the -Duse64bitint flag to Configure. Although there are some minor differences between compiling Perl with this flag versus the -Duse64bitall flag, they should not be noticeable from a Perl user's perspective.

In both cases, it is strongly recommended that you use these flags when you run Configure. If you do not use do this, but later answer the questions about 64-bit numbers when Configure asks you, you may get a configuration that cannot be compiled, or that does not function as expected.

108.1.17 Oracle on HP-UX

Using perl to connect to Oracle databases through DBI and DBD::Oracle has caused a lot of people many headaches. Read README.hpux in the DBD::Oracle for much more information. The reason to mention it here is that Oracle requires a perl built with libcl and libpthread, the latter even when perl is build without threads. Building perl using all defaults, but still enabling to build DBD::Oracle later on can be achieved using

```
  Configure -A prepend:libswanted='cl pthread ' ...
```

Do not forget the space before the trailing quote.

Also note that this does not (yet) work with all configurations, it is known to fail with 64-bit versions of GCC.

108.1.18 GDBM and Threads on HP-UX

If you attempt to compile Perl with threads on an 11.X system and also link in the GDBM library, then Perl will immediately core dump when it starts up. The only workaround at this point is to relink the GDBM library under 11.X, then relink it into Perl.

108.1.19 NFS filesystems and utime(2) on HP-UX

If you are compiling Perl on a remotely-mounted NFS filesystem, the test io/fs.t may fail on test #18. This appears to be a bug in HP-UX and no fix is currently available.

108.1.20 perl -P and // and HP-UX

If HP-UX Perl is compiled with flags that will cause problems if the -P flag of Perl (preprocess Perl code with the C preprocessor before perl sees it) is used. The problem is that `//`, being a C++-style until-end-of-line comment, will disappear along with the remainder of the line. This means that common Perl constructs like

```
    s/foo//;
```

will turn into illegal code

```
    s/foo
```

The workaround is to use some other quoting separator than `"/"`, like for example `"!"`:

```
    s!foo!!;
```

108.1.21 HP-UX Kernel Parameters (maxdsiz) for Compiling Perl

By default, HP-UX comes configured with a maximum data segment size of 64MB. This is too small to correctly compile Perl with the maximum optimization levels. You can increase the size of the maxdsiz kernel parameter through the use of SAM.
When using the GUI version of SAM, click on the Kernel Configuration icon, then the Configurable Parameters icon. Scroll down and select the maxdsiz line. From the Actions menu, select the Modify Configurable Parameter item. Insert the new formula into the Formula/Value box. Then follow the instructions to rebuild your kernel and reboot your system.
In general, a value of 256MB (or "256*1024*1024") is sufficient for Perl to compile at maximum optimization.

108.2 nss_delete core dump from op/pwent or op/grent

You may get a bus error core dump from the op/pwent or op/grent tests. If compiled with -g you will see a stack trace much like the following:

```
  #0  0xc004216c in  () from /usr/lib/libc.2
  #1  0xc00d7550 in __nss_src_state_destr () from /usr/lib/libc.2
  #2  0xc00d7768 in __nss_src_state_destr () from /usr/lib/libc.2
  #3  0xc00d78a8 in nss_delete () from /usr/lib/libc.2
  #4  0xc01126d8 in endpwent () from /usr/lib/libc.2
  #5  0xd1950 in Perl_pp_epwent () from ./perl
  #6  0x94d3c in Perl_runops_standard () from ./perl
  #7  0x23728 in S_run_body () from ./perl
  #8  0x23428 in perl_run () from ./perl
  #9  0x2005c in main () from ./perl
```

The key here is the `nss_delete` call. One workaround for this bug seems to be to create add to the file */etc/nsswitch.conf* (at least) the following lines

```
  group: files
  passwd: files
```

Whether you are using NIS does not matter. Amazingly enough, the same bug also affects Solaris.

108.3 AUTHOR

Jeff Okamoto <okamoto@corp.hp.com> H.Merijn Brand <h.m.brand@hccnet.nl>
With much assistance regarding shared libraries from Marc Sabatella.

108.4 DATE

Version 0.7.0: 2004-06-09

Chapter 109

README.hurd

Perl version 5 on Hurd

109.1 DESCRIPTION

If you want to use Perl on the Hurd, I recommend using the Debian GNU/Hurd distribution (see http://www.debian.org/), even if an official, stable release has not yet been made. The old 'gnu-0.2' binary distribution will most certainly have additional problems.

109.1.1 Known Problems with Perl on Hurd

The Perl test suite may still report some errors on the Hurd. The 'lib/anydbm' and 'pragma/warnings' tests will almost certainly fail. Both failures are not really specific to the Hurd, as indicated by the test suite output.

The socket tests may fail if the network is not configured. You have to make '/hurd/pfinet' the translator for '/servers/socket/2', giving it the right arguments. Try '/hurd/pfinet –help' for more information.

Here are the statistics for Perl 5.005_62 on my system:

```
Failed Test  Status Wstat Total Fail  Failed  List of failed
-------------------------------------------------------------------------
lib/anydbm.t                 12    1   8.33%  12
pragma/warnings             333    1   0.30%  215

8 tests and 24 subtests skipped.
Failed 2/229 test scripts, 99.13% okay. 2/10850 subtests failed, 99.98% okay.
```

There are quite a few systems out there that do worse!

However, since I am running a very recent Hurd snapshot, in which a lot of bugs that were exposed by the Perl test suite have been fixed, you may encounter more failures. Likely candidates are: 'op/stat', 'lib/io_pipe', 'lib/io_sock', 'lib/io_udp' and 'lib/time'.

In any way, if you're seeing failures beyond those mentioned in this document, please consider upgrading to the latest Hurd before reporting the failure as a bug.

109.2 AUTHOR

Mark Kettenis <kettenis@gnu.org>

Last Updated: Fri, 29 Oct 1999 22:50:30 +0200

Chapter 110

README.irix

Perl version 5 on Irix systems

110.1 DESCRIPTION

This document describes various features of Irix that will affect how Perl version 5 (hereafter just Perl) is compiled and/or runs.

110.1.1 Building 32-bit Perl in Irix

Use

```
        sh Configure -Dcc='cc -n32'
```

to compile Perl 32-bit. Don't bother with -n32 unless you have 7.1 or later compilers (use cc -version to check).
(Building 'cc -n32' is the default.)

110.1.2 Building 64-bit Perl in Irix

Use

```
        sh Configure -Dcc='cc -64' -Duse64bitint
```

This requires require a 64-bit MIPS CPU (R8000, R10000, ...)
You can also use

```
        sh Configure -Dcc='cc -64' -Duse64bitall
```

but that makes no difference compared with the -Duse64bitint because of the `cc -64`.
You can also do

```
        sh Configure -Dcc='cc -n32' -Duse64bitint
```

to use long longs for the 64-bit integer type, in case you don't have a 64-bit CPU.
If you are using gcc, just

```
        sh Configure -Dcc=gcc -Duse64bitint
```

should be enough, the Configure should automatically probe for the correct 64-bit settings.

110.1.3 About Compiler Versions of Irix

Some Irix cc versions, e.g. 7.3.1.1m (try cc -version) have been known to have issues (coredumps) when compiling perl.c. If you've used -OPT:fast_io=ON and this happens, try removing it. If that fails, or you didn't use that, then try adjusting other optimization options (-LNO, -INLINE, -O3 to -O2, etcetera). The compiler bug has been reported to SGI. (Allen Smith <easmith@beatrice.rutgers.edu>)

110.1.4 Linker Problems in Irix

If you get complaints about so_locations then search in the file hints/irix_6.sh for "lddflags" and do the suggested adjustments. (David Billinghurst <David.Billinghurst@riotinto.com.au>)

110.1.5 Malloc in Irix

Do not try to use Perl's malloc, this will lead into very mysterious errors (especially with -Duse64bitall).

110.1.6 Building with threads in Irix

Run Configure with -Duseithreads which will configure Perl with the new Perl 5.8.0 "interpreter threads", see *threads*.

The old Perl 5.005 threads is obsolete, unmaintained, and its use is discouraged. If you really want it, run Configure with the -Dusethreads -Duse5005threads options as described in INSTALL.

For either thread model and for Irix 6.2, you have to have the following patches installed:

```
        1404 Irix 6.2 Posix 1003.1b man pages
        1645 Irix 6.2 & 6.3 POSIX header file updates
        2000 Irix 6.2 Posix 1003.1b support modules
        2254 Pthread library fixes
        2401 6.2 all platform kernel rollup
```

IMPORTANT: Without patch 2401, a kernel bug in Irix 6.2 will cause your machine to panic and crash when running threaded perl. Irix 6.3 and later are okay.

```
    Thanks to Hannu Napari <Hannu.Napari@hut.fi> for the IRIX
    pthreads patches information.
```

110.1.7 Irix 5.3

While running Configure and when building, you are likely to get quite a few of these warnings:

```
  ld:
  The shared object /usr/lib/libm.so did not resolve any symbols.
        You may want to remove it from your link line.
```

Ignore them: in IRIX 5.3 there is no way to quieten ld about this.

During compilation you will see this warning from toke.c:

```
  uopt: Warning: Perl_yylex: this procedure not optimized because it
        exceeds size threshold; to optimize this procedure, use -Olimit option
        with value >= 4252.
```

Ignore the warning.

In IRIX 5.3 and with Perl 5.8.1 (Perl 5.8.0 didn't compile in IRIX 5.3) the following failures are known.

```
Failed Test                  Stat Wstat Total Fail  Failed  List of Failed
--------------------------------------------------------------------------
../ext/List/Util/t/shuffle.t    0   139    ??   ??       %  ??
../lib/Math/Trig.t            255 65280    29   12  41.38%  24-29
../lib/sort.t                   0   138   119   72  60.50%  48-119
56 tests and 474 subtests skipped.
Failed 3/811 test scripts, 99.63% okay. 78/75813 subtests failed, 99.90% okay.
```

They are suspected to be compiler errors (at least the shuffle.t failure is known from some IRIX 6 setups) and math library errors (the Trig.t failure), but since IRIX 5 is long since end-of-lifed, further fixes for the IRIX are unlikely. If you can get gcc for 5.3, you could try that, too, since gcc in IRIX 6 is a known workaround for at least the shuffle.t and sort.t failures.

110.2 AUTHOR

Jarkko Hietaniemi <jhi@iki.fi>

Please report any errors, updates, or suggestions to *perlbug@perl.org*.

Chapter 111

README.machten

Perl version 5 on Power MachTen systems

111.1 DESCRIPTION

This document describes how to build Perl 5 on Power MachTen systems, and discusses a few wrinkles in the implementation.

111.1.1 Perl version 5.8.x and greater not supported

Power MachTen is not supported by versions of Perl later than 5.6.x. If you wish to build a version from the 5.6 track, please obtain a source distribution from the archive at http://cpan.org/src/5.0/ and follow the instructions in its README.machten file.

MachTen is no longer supported by its developers, Tenon Intersystems. A UNIX environment hosted on Mac OS Classic, MachTen has been superseded by Mac OS X and by BSD and Linux implementations for Macintosh hardware. The final version of Power MachTen, 4.1.4, lacks many features found in modern implementations of UNIX, and has a number of bugs. These shortcomings prevent recent versions of Perl from being able to use extensions on MachTen, and cause numerous test suite failures in the perl core.

In September 2003, a discussion on the MachTen mailing list determined that there was no interest in making a later version of Perl build successfully on MachTen. Consequently, support for building Perl under MachTen has been suppressed in Perl distributions published after February 2004. The hints file, *hints/machten.sh*, remains a part of the distributions for reference purposes.

111.1.2 Compiling Perl 5.6.x on MachTen

To compile perl 5.6.x under MachTen 4.1.4 (and probably earlier versions):

```
  ./Configure -de
  make
  make test
  make install
```

This builds and installs a statically-linked perl; MachTen's dynamic linking facilities are not adequate to support Perl's use of dynamically linked libraries. (See *hints/machten.sh* for more information.)

You should have at least 32 megabytes of free memory on your system before running the `make` command.

For much more information on building perl – for example, on how to change the default installation directory – see *INSTALL*.

111.1.3 Failures during `make test` on MachTen

op/lexassign.t

This test may fail when first run after building perl. It does not fail subsequently. The cause is unknown.

pragma/warnings.t

Test 257 fails due to a failure to warn about attempts to read from a filehandle which is a duplicate of stdout when stdout is attached to a pipe. The output of the test contains a block comment which discusses a different failure, not applicable to MachTen.

The root of the problem is that Machten does not assign a file type to either end of a pipe (see *stat*), resulting, among other things in Perl's `-p` test failing on file descriptors belonging to pipes. As a result, perl becomes confused, and the test for reading from a write-only file fails. I am reluctant to patch perl to get around this, as it's clearly an OS bug (about which Tenon has been informed), and limited in its effect on practical Perl programs.

111.1.4 Building external modules on MachTen

To add an external module to perl, build in the normal way, which is documented in *ExtUtils::MakeMaker*, or which can be driven automatically by the CPAN module (see *CPAN*), which is part of the standard distribution. If you want to install a module which contains XS code (C or C++ source which compiles to object code for linking with perl), you will have to replace your perl binary with a new version containing the new statically-linked object module. The build process tells you how to do this.

There is a gotcha, however, which users usually encounter immediately they respond to CPAN's invitation to `install Bundle::CPAN`. When installing a *bundle* – a group of modules which together achieve some particular purpose, the installation process for later modules in the bundle tends to assume that earlier modules have been fully installed and are available for use. This is not true on a statically-linked system for earlier modules which contain XS code. As a result the installation of the bundle fails. The work-around is not to install the bundle as a one-shot operation, but instead to see what modules it contains, and install these one-at-a-time by hand in the order given.

111.2 AUTHOR

Dominic Dunlop <domo@computer.org>

111.3 DATE

Version 1.1.0 2004-02-13

Chapter 112

README.macos

Perl under Mac OS (Classic)

112.1 SYNOPSIS

This document briefly describes perl under Mac OS (Classic). If you are running perl under Mac OS X, you don't want to be here (unless you are in the Classic environment under Mac OS X).

When we say "Mac OS" below, we mean Mac OS 7, 8, and 9, and *not* Mac OS X.

112.2 DESCRIPTION

The latest perl source itself builds on Mac OS, with some additional pieces. Support for Mac OS is now in the perl core, and MacPerl is kept in close sync with regular perl releases.

To build perl for Mac OS (as an MPW tool), you will need the addition of the *macos* subdirectory, distributed separately. It includes extra source files, config files, and make files. It also includes extra Mac-specific modules.

To build the MacPerl application, you will also need the *macperl* directory, which includes the source files for creating the application itself.

All of this is available from the development site, via HTTP (in the MacPerl Installer, which includes all the source and binaries) and anonymous CVS.

```
    http://dev.macperl.org/
```

The source is also in the main perl repository in the macperl branch (the 5.6 source is in the maint-5.6/macperl branch).

You will also need compilers and libraries, all of them freely available. These are linked to from the SourceForge site. Go that site for all things having to do with MacPerl development.

MacPerl 5.6.1 and later are supported on Mac OS 8.1 and later, for 68040 and PowerPC architectures. The MPW tool may be used on Mac OS 7.5.5 and 68030 computers.

MacPerl 5.2.0r4 is also available, on the CPAN and on SourceForge. It is based on perl 5.004, and works with Mac OS 7.5.5 and 68030 computers.

112.3 AUTHOR

perl was ported to Mac OS by Matthias Neeracher <neeracher@mac.com>. It is currently maintained by Chris Nandor <pudge@pobox.com>.

112.4 DATE

Last modified 2002.05.02.

Chapter 113

README.macosx

Perl under Mac OS X

113.1 SYNOPSIS

This document briefly describes perl under Mac OS X.

113.2 DESCRIPTION

The latest Perl (5.8.1-RC3 as of this writing) builds without changes under Mac OS X. Under the 10.3 "Panther" release, all self-tests pass, and all standard features are supported.

Earlier Mac OS X releases did not include a completely thread-safe libc, so threading is not fully supported. Also, earlier releases included a somewhat buggy libdb, so some of the DB_File tests are known to fail on those releases.

113.2.1 Installation Prefix

The default installation location for this release uses the traditional UNIX directory layout under /usr/local. This is the recommended location for most users, and will leave the Apple-supplied Perl and its modules undisturbed.

Using an installation prefix of '/usr' will result in a directory layout that mirrors that of Apple's default Perl, with core modules stored in '/System/Library/Perl/${version}', CPAN modules stored in '/Library/Perl/${version}', and the addition of '/Network/Library/Perl/${version}' to @INC for modules that are stored on a file server and used by many Macs.

113.2.2 libperl and Prebinding

Mac OS X ships with a dynamically-loaded libperl, but the default for this release is to compile a static libperl. The reason for this is pre-binding. Dynamic libraries can be pre-bound to a specific address in memory in order to decrease load time. To do this, one needs to be aware of the location and size of all previously-loaded libraries. Apple collects this information as part of their overall OS build process, and thus has easy access to it when building Perl, but ordinary users would need to go to a great deal of effort to obtain the information needed for pre-binding.

You can override the default and build a shared libperl if you wish (Configure ... -Duseshrlib), but the load time will be significantly greater than either the static library, or Apple's pre-bound dynamic library.

113.2.3 Updating Panther

As of this writing, the latest Perl release that has been tested and approved for inclusion in the 10.3 "Panther" release of Mac OS X is 5.8.1 RC3. It is currently unknown whether the final 5.8.1 release will be made in time to be tested and included with Panther.

If the final release of Perl 5.8.1 is not made in time to be included with Panther, it is recommended that you wait for an official Apple update to the OS, rather than attempting to update it yourself. In most cases, if you need a newer Perl, it is preferable to install it in some other location, such as /usr/local or /opt, rather than overwriting the system Perl. The default location (no -Dprefix=... specified when running Configure) is /usr/local.

If you find that you do need to update the system Perl, there is one potential issue. If you upgrade using the default static libperl, you will find that the dynamic libperl supplied by Apple will not be deleted. If both libraries are present when an application that links against libperl is built, ld will link against the dynamic library by default. So, if you need to replace Apple's dynamic libperl with a static libperl, you need to be sure to delete the older dynamic library after you've installed the update.

Note that this is only an issue when updating from an older build of the same Perl version. If you're updating from (for example) 5.8.1 to 5.8.2, this issue won't affect you.

113.2.4 Known problems

If you have installed extra libraries such as GDBM through Fink (in other words, you have libraries under */sw/lib*), or libdlcompat to */usr/local/lib*, you may need to be extra careful when running Configure to not to confuse Configure and Perl about which libraries to use. Being confused will show up for example as "dyld" errors about symbol problems, for example during "make test". The safest bet is to run Configure as

```
    Configure ... -Uloclibpth -Dlibpth=/usr/lib
```

to make Configure look only into the system libraries. If you have some extra library directories that you really want to use (such as newer Berkeley DB libraries in pre-Panther systems), add those to the libpth:

```
    Configure ... -Uloclibpth -Dlibpth='/usr/lib /opt/lib'
```

The default of building Perl statically may cause problems with complex applications like Tk: in that case consider building shared Perl

```
    Configure ... -Duseshrplib
```

but remember that there's a startup cost to pay in that case (see above "libperl and Prebinding").

113.2.5 MacPerl

Quite a bit has been written about MacPerl, the Perl distribution for "Classic MacOS" - that is, versions 9 and earlier of MacOS. Because it runs in environment that's very different from that of UNIX, many things are done differently in MacPerl. Modules are installed using a different procedure, Perl itself is built differently, path names are different, etc.

From the perspective of a Perl programmer, Mac OS X is more like a traditional UNIX than Classic MacOS. If you find documentation that refers to a special procedure that's needed for MacOS that's drastically different from the instructions provided for UNIX, the MacOS instructions are quite often intended for MacPerl on Classic MacOS. In that case, the correct procedure on Mac OS X is usually to follow the UNIX instructions, rather than the MacPerl instructions.

113.2.6 Carbon

MacPerl ships with a number of modules that are used to access the classic MacOS toolbox. Many of these modules have been updated to use Mac OS X's newer "Carbon" toolbox, and are available from CPAN in the "Mac::Carbon" module.

113.2.7 Cocoa

There are two ways to use Cocoa from Perl. Apple's PerlObjCBridge module, included with Mac OS X, can be used by standalone scripts to access Foundation (i.e. non-GUI) classes and objects.

An alternative is CamelBones, a framework that allows access to both Foundation and AppKit classes and objects, so that full GUI applications can be built in Perl. CamelBones can be found on SourceForge, at http://www.sourceforge.net/projects/camelbones/.

113.3 Starting From Scratch

Unfortunately it is not that difficult somehow manage to break one's Mac OS X Perl rather severely. If all else fails and you want to really, **REALLY**, start from scratch and remove even your Apple Perl installation (which has become corrupted somehow), the following instructions should do it. **Please think twice before following these instructions: they are much like conducting brain surgery to yourself. Without anesthesia.** We will **not** come to fix your system if you do this.

First, get rid of the libperl.dylib:

```
    # cd /System/Library/Perl/darwin/CORE
    # rm libperl.dylib
```

Then delete every .bundle file found anywhere in the folders:

```
    /System/Library/Perl
    /Library/Perl
```

You can find them for example by

```
    # find /System/Library/Perl /Library/Perl -name '*.bundle' -print
```

After this you can either copy Perl from your operating system CDs (you will need at least the /System/Library/Perl and /usr/bin/perl), or rebuild Perl from the source code with `Configure -Dprefix=/usr -Dusershrplib` NOTE: the `-Dprefix=/usr` to replace the system Perl works much better with Perl 5.8.1 and later, in Perl 5.8.0 the settings were not quite right.

113.4 AUTHOR

This README was written by Sherm Pendley <sherm@dot-app.org>. The "Starting From Scratch" recipe was contributed by John Montbriand <montbriand@apple.com>.

113.5 DATE

Last modified 2003-09-08.

Chapter 114

README.mint

Perl version 5 on Atari MiNT

114.1 DESCRIPTION

There is a binary version of perl available from the FreeMiNT project http://freemint.de/ You may wish to use this instead of trying to compile yourself.

The following advice is from perl 5.004_02 and is probably rather out of date.

If you want to build perl yourself on MiNT (or maybe on an Atari without MiNT) you may want to accept some advice from somebody who already did it...

There was a perl port for Atari ST done by ++jrb bammi@cadence.com. This port tried very hard to build on non-MiNT-systems. For the sake of efficiency I've left this way. Yet, I haven't removed bammi's patches but left them intact. Unfortunately some of the files that bammi contributed to the perl distribution seem to have vanished?

So, how can you distinguish my patches from bammi's patches? All of bammi's stuff is embedded in "#ifdef atarist" preprocessor macros. My MiNT port uses "#ifdef __MINT__" instead (and unconditionally undefines "atarist". If you want to continue on bammi's port, all you have to do is to swap the "-D" and "-U" switches for "__MINT__" and "atarist" in the variable ccflags.

However, I think that my version will still run on non-MiNT-systems provided that the user has a Eunuchs-like environment (i.e. the standard envariables like $PATH, $HOME, ... are set, there is a POSIX compliant shell in /bin/sh, and...)

114.2 Known problems with Perl on MiNT

The problems you may encounter when building perl on your machine are most probably due to deficiencies in MiNT resp. the Atari platform in general.

First of all, if you have less than 8 MB of RAM you shouldn't even try to build Perl yourself. Better grab a binary pre-compiled version somewhere. Even if you have more memory you should take some care. Try to run in a fresh environment (without memory fragmented too much) with as few daemons, accessories, xcontrol modules etc. as possible. If you run some AES you should consider to start a console based environment instead.

A problem has been reported with sed. Sed is used to create some configuration files based on the answers you have given to the Configure script. Unfortunately the Perl Configure script shows sed on MiNT its limits. I have sed 2.05 with a stacksize of 64k and I have encountered no problems. If sed crashes during your configuration process you should first try to augment sed's stacksize:

```
        fixstk 64k /usr/bin/sed
```

(or similar). If it still doesn't help you may have a look which other versions of sed are installed on your system. If you have a KGMD 1.0 installation you will find three in /usr/bin. Have a look there.

Perl has some "mammut" C files. If gcc reports "internal compiler error: program cc1 got fatal signal 10" this is very likely due to a stack overflow in program cc1. Find cc1 and fix its stack. I have made good experiences with

```
        fixstk 2 cc1
```

This doesn't establish a stack of 2 Bytes only as you might think. It really reserves one half of the available memory for cc1's stack. A setting of 1 would reserve the entire memory for cc1, 3 would reserve three fourths. You will have to find out the value that suits to your system yourself.

To find out the location of the program 'cc1' simply type 'gcc –print-prog-name cc1' at your shell prompt.

Now run make (maybe "make -k"). If you get a fatal signal 10 increase cc1's stacksize, if you run out of memory you should either decrease the stacksize or follow some more hints:

Perl's building process is very handy on machines with a lot of virtual memory but may result in a disaster if you are short of memory. If gcc fails to compile many source files you should reduce the optimization. Grep for "optimize" in the file config.sh and change the flags.

If only several huge files cause problems (actually it is not a matter of the file size resp. the amount of code but depends on the size of the individual functions) it is useful to bypass the make program and compile these files directly from the command line. For example if you got something like the following from make:

```
        CCCMD = gcc -DPERL_CORE ....
        ...
        ...: virtual memory exhausted
```

you should hack into the shell:

```
        gcc -DPERL_CORE ... toke.c
```

Please note that you have to add the name of the source file (here toke.c) at the end.

If none of this helps, you're helpless. Wait for a binary release. If you have succeeded you may encounter another problem at the linking process. If gcc complains that it can't find some libraries within the perl distribution you probably have an old linker. If it complains for example about "file not found for xxx.olb" you should cd into the directory in question and

```
        ln -s libxxx.a xxx.olb
```

This will fix the problem.

This version (5.00402) of perl has passed most of the tests on my system:

```
 Failed Test  Status Wstat Total Fail  Failed  List of failed
 ------------------------------------------------------------------------------
 io/pipe.t                   10    2  20.00%  7, 9
 io/tell.t                   13    1   7.69%  12
 lib/complex.t              762   13   1.71%  84-85, 248-251, 257, 272-273,
                                              371, 380, 419-420
 lib/io_pipe.t               10    1  10.00%  9
 lib/io_tell.t               13    1   7.69%  12
 op/magic.t                  30    2   6.67%  29-30
 Failed 6/152 test scripts, 96.05% okay. 20/4359 subtests failed, 99.54% okay.
```

Pipes always cause problems with MiNT, it's actually a surprise that most of the tests did work. I've got no idea why the "tell" test failed, this shouldn't mean too big a problem however.

Most of the failures of lib/complex seem to be harmless, actually errors far right to the decimal point... Two failures seem to be serious: The sign of the results is reversed. I would say that this is due to minor bugs in the portable math lib that I compiled perl with.

I haven't bothered very much to find the reason for the failures with op/magic.t and op/stat.t. Maybe you'll find it out.

###

Another possible problem may arise from the implementation of the "pwd" command. It happened to add a carriage return and newline to its output no matter what the setting of $UNIXMODE is. This is quite annoying since many library modules for perl take the output of pwd, chop off the trailing newline character and then expect to see a valid path in that. But the carriage return (last but second character!) isn't chopped off. You can either try to patch all library modules (at the price of performance for the extra transformation) or you can use my version of pwd that doesn't suffer from this deficiency.

The fixed implementation is in the mint subdirectory. Running "Configure" will attempt to build and install it if necessary (hints/mint.sh will do this work) but you can build and install it explicitly by:

```
    cd mint
    make install
```

This is the fastest solution.

Just in case you want to go the hard way: perl won't even build with a broken pwd! You will have to fix the library modules (ext/POSIX/POSIX.pm, lib/Cwd.pm, lib/pwd.pl) at last after building miniperl.

A major nuisance of current MiNTLib versions is the implementation of system() which is far from being POSIX compliant. A real system() should fork and then exec /bin/sh with its argument as a command line to the shell. The MiNTLib system() however doesn't expect that every user has a POSIX shell in /bin/sh. It tries to work around the problem by forking and exec'ing the first token in its argument string. To get a little bit of compliance to POSIX system() it tries to handle at least redirection ("<" or ">") on its own behalf.

This isn't a good idea since many programs expect that they can pass a command line to system() that exploits all features of a POSIX shell. If you use the MiNTLib version of system() with perl the Perl function system() will suffer from the same deficiencies.

You will find a fixed version of system() in the mint subdirectory. You can easily insert this version into your system libc:

```
    cd mint
    make system.o
    ar r /usr/lib/libc.a
    ranlib /usr/lib/libc.a
```

If you are suspicious you should either back up your libc before or extract the original system.o from your libc with "ar x /usr/lib/libc.a system.o". You can then backup the system.o module somewhere before you succeed.

Anything missing? Yep, I've almost forgotten... No file in this distribution without a fine saying. Take this one:

```
    "From a thief you should learn: (1) to work at night;
    (2) if one cannot gain what one wants in one night to
    try again the next night; (3) to love one's coworkers
    just as thieves love each other; (4) to be willing to
    risk one's life even for a little thing; (5) not to
    attach too much value to things even though one has
    risked one's life for them - just as a thief will resell
    a stolen article for a fraction of its real value;
    (6) to withstand all kinds of beatings and tortures
    but to remain what you are; and (7) to believe your
    work is worthwhile and not be willing to change it."

                    -- Rabbi Dov Baer, Maggid of Mezeritch
```

OK, this was my motto while working on Perl for MiNT, especially rule (1)...

Have fun with Perl!

114.3 AUTHOR

Guido Flohr

```
mailto:guido@FreeMiNT.de
```

Chapter 115

README.mpeix

Perl/iX for HP e3000 MPE

115.1 SYNOPSIS

```
http://www.bixby.org/mark/perlix.html
http://jazz.external.hp.com/src/hp_freeware/perl/
Perl language for MPE
Last updated July 29, 2003 @ 2100 UTC
```

115.2 NOTE

This is a podified version of the above-mentioned web page, podified by Jarkko Hietaniemi 2001-Jan-01.

115.3 Binary distribution from HP

The simplest way to obtain Perl for the MPE/iX is to go either of these URLs and follow the instructions within.

http://jazz.external.hp.com/src/hp_freeware/perl/ http://www.bixby.org/mark/perlix.html

Use which ever one is more recent.

115.4 What's New in Perl for MPE/iX

June 1, 2000

- Rebuilt to be compatible with mod_perl. If you plan on using mod_perl, you MUST download and install this version of Perl/iX!
- uselargefiles="undef": not available in MPE for POSIX files yet.
- Now bundled with various add-on packages:
  - libnet (as seen on CPAN)
  - libwww-perl (LWP) which lets Perl programs behave like web browsers:

```
1. #!/PERL/PUB/perl
2. use LWP::Simple;
3. $doc = get('http://www.bixby.org/mark/perlix.html');  # reads the
   web page into variable $doc
```

(http://www.bixby.org/mark/perlix.html)

- mod_perl (just the perl portion; the actual DSO will be released soon with Apache/iX 1.3.12 from bixby.org). This module allows you to write high performance persistent Perl CGI scripts and all sorts of cool things. (http://perl.apache.org/)

  and much much more hiding under /PERL/PUB/.cpan/
- The CPAN module now works for automatic downloading and installing of add-on packages:

```
1. export FTP_PASSIVE=1
2. perl -MCPAN -e shell
3. Ignore any terminal I/O related complaints!
```

  (http://theoryx5.uwinnipeg.ca/CPAN/data/perl/CPAN.html)

May 20, 2000

- Updated to version 5.6.0. Builds straight out of the box on MPE/iX.
- Perl's getpwnam() function which had regressed to being unimplemented on MPE is now implemented once again.

September 17, 1999

- Migrated from cccd.edu to bixby.org.

115.5 Welcome to Perl/iX

This is the official home page for the HP e3000 MPE/iX (http://www.hp.com/go/e3000) port of the Perl scripting language (http://www.perl.com/) which gives you all of the power of C, awk, sed, and sh in a single language. Check here for the latest news, implemented functionality, known bugs, to-do list, etc. Status reports about major milestones will also be posted to the HP3000-L mailing list (http://www.lsoft.com/scripts/wl.exe?SL1=HP3000-L&H=RAVEN.UTC.EDU) and its associated gatewayed newsgroup comp.sys.hp.mpe.

I'm doing this port because I can't live without Perl on the Unix machines that I administer, and I want to have the same power available to me on MPE.

Please send your comments, questions, and bug reports directly to me, Mark Bixby (http://www.bixby.org/mark/). Or just post them to HP3000-L.

The platform I'm using to do this port is an HP 3000 957RX running MPE/iX 6.0 and using the GNU gcc C compiler (http://jazz.external.hp.com/src/gnu/gnuframe.html).

The combined porting wisdom from all of my ports can be found in my MPE/iX Porting Guide (http://www.bixby.org/mark/porting.html).

IMPORTANT NOTICE: Yes, I do work for the HP CSY R&D lab, but ALL of the software you download from bixby.org is my personal freeware that is NOT supported by HP.

115.6 System Requirements for Perl/iX

- MPE/iX 5.5 or later. This version of Perl/iX does NOT run on MPE/iX 5.0 or earlier, nor does it run on "classic" MPE/V machines.
- If you wish to recompile Perl, you must install both GNUCORE and GNUGCC from jazz (http://jazz.external.hp.com/src/gnu/gnuframe.html).
- Perl/iX will be happier on MPE/iX 5.5 if you install the MPEKX40B extended POSIX filename characters patch, but this is optional.
- Patch LBCJXT6A is required on MPE/iX 5.5 machines in order to prevent Perl/iX from dying with an unresolved external reference to _getenv_libc.
- If you will be compiling Perl/iX yourself, you will also need Syslog/iX (http://www.bixby.org/mark/syslogix.html) and the /BIND/PUB/include and /BIND/PUB/lib portions of BIND/iX (http://www.bixby.org/mark/bindix.html).

115.7 How to Obtain Perl/iX

1. Download Perl using either FTP.ARPA.SYS or some other client
2. Extract the installation script
3. Edit the installation script
4. Run the installation script
5. Convert your *.a system archive libraries to *.sl shared libraries

Download Perl using FTP.ARPA.SYS from your HP 3000 (the preferred method).....

```
:HELLO MANAGER.SYS
:XEQ FTP.ARPA.SYS
open ftp.bixby.org
anonymous
your@email.address
bytestream
cd /pub/mpe
get perl-5.6.0-mpe.tar.Z /tmp/perl.tar.Z;disc=2147483647
exit
```

.....Or download using some other generic web or ftp client (the alternate method)
Download the following files (make sure that you use "binary mode" or whatever client feature that is 8-bit clean):

- Perl from

  ```
  http://www.bixby.org/ftp/pub/mpe/perl-5.6.0-mpe.tar.Z
  ```

  or

  ```
  ftp://ftp.bixby.org/pub/mpe/perl-5.6.0-mpe.tar.Z
  ```

- Upload those files to your HP 3000 in an 8-bit clean bytestream manner to:

  ```
  /tmp/perl.tar.Z
  ```

- Then extract the installation script (after both download methods)

  ```
  :CHDIR /tmp
  :XEQ TAR.HPBIN.SYS 'xvfopz /tmp/perl.tar.Z INSTALL'
  ```

- Edit the installation script

  Examine the accounting structure creation commands and modify if necessary (adding additional capabilities, choosing a non-system volume set, etc).

  ```
  :XEQ VI.HPBIN.SYS /tmp/INSTALL
  ```

- Run the installation script.

  The accounting structure will be created and then all files will be extracted from the archive.

  ```
  :XEQ SH.HPBIN.SYS /tmp/INSTALL
  ```

- Convert your *.a system archive libraries to *.sl shared libraries

  You only have to do this ONCE on your MPE/iX 5.5 machine in order to convert /lib/lib*.a and /usr/lib/lib*.a libraries to their *.sl equivalents. This step should not be necessary on MPE/iX 6.0 or later machines because the 6.0 or later update process does it for you.

  ```
  :XEQ SH.HPBIN.SYS /PERL/PUB/LIBSHP3K
  ```

115.8 Perl/iX Distribution Contents Highlights

README

The file you're reading now.

INSTALL

Perl/iX Installation script.

LIBSHP3K

Script to convert *.a system archive libraries to *.sl shared libraries.

PERL

Perl NMPRG executable. A version-numbered backup copy also exists. You might wish to "ln -s /PERL/PUB/PERL /usr/local/bin/perl".

.cpan/

Much add-on source code downloaded with the CPAN module.

lib/

Perl libraries, both core and add-on.

man/

Perl man page documentation.

public_html/feedback.cgi

Sample feedback CGI form written in Perl.

src/perl-5.6.0-mpe

Source code.

115.9 How to Compile Perl/iX

1. cd src/perl-5.6.0-mpe
2. Read the INSTALL file for the official instructions
3. ./Configure -d
4. make
5. ./mpeix/relink
6. make test (expect approximately 15 out of 11306 subtests to fail, mostly due to MPE not supporting hard links, UDP socket problems, and handling exit() return codes improperly)
7. make install
8. Optionally create symbolic links that point to the Perl executable, i.e. ln -s /PERL/PUB/PERL /usr/local/bin/perl

The summary test results from "cd t; ./perl -I../lib harness":

```
Failed Test  Status Wstat Total Fail  Failed  List of failed
-------------------------------------------------------------------------
io/fs.t                     29    8  27.59%  2-5, 7-9, 11
io/openpid.t                10    1  10.00%  7
lib/io_sock.t               14    1   7.14%  13
lib/io_udp.t                 7    2  28.57%  3, 5
```

```
  lib/posix.t                  27    1   3.70%  12
  op/lex_assign.t             187    1   0.53%  13
  op/stat.t                    58    1   1.72%  3
  15 tests and 94 subtests skipped.
  Failed 7/236 test scripts, 97.03% okay. 15/11306 subtests failed, 99.87% okay.
```

115.10 Getting Started with Perl/iX

Create your Perl script files with "#!/PERL/PUB/perl" (or an equivalent symbolic link) as the first line. Use the chmod command to make sure that your script has execute permission. Run your script!

Be sure to take a look at the CPAN module list (http://www.cpan.org/CPAN.html). A wide variety of free Perl software is available. You can automatically download these packages by using the CPAN module (http://theoryx5.uwinnipeg.ca/CPAN/data/perl/CPAN.html).

115.11 MPE/iX Implementation Considerations

There some minor functionality issues to be aware of when comparing Perl for Unix (Perl/UX) to Perl/iX:

- MPE gcc/ld doesn't properly support linking NMPRG executables against NMXL dynamic libraries, so you must manually run mpeix/relink after each re-build of Perl.
- Perl/iX File::Copy will use MPE's /bin/cp command to copy files by name in order to preserve file attributes like file code.
- MPE (and thus Perl/iX) lacks support for setgrent(), endgrent(), setpwent(), endpwent().
- MPE (and thus Perl/iX) lacks support for hard links.
- MPE requires GETPRIVMODE() in order to bind() to ports less than 1024. Perl/iX will call GETPRIVMODE() automatically on your behalf if you attempt to bind() to these low-numbered ports. Note that the Perl/iX executable and the PERL account do not normally have CAP=PM, so if you will be bind()-ing to these privileged ports, you will manually need to add PM capability as appropriate.
- MPE requires that you bind() to an IP address of zero. Perl/iX automatically replaces the IP address that you pass to bind() with a zero.
- MPE requires GETPRIVMODE() in order to setuid(). There are too many calls to setuid() within Perl/iX, so I have not attempted an automatic GETPRIVMODE() solution similar to bind().

115.12 Known Perl/iX Bugs Under Investigation

None.

115.13 Perl/iX To-Do List

- Make setuid()/setgid() support work.
- Make sure that fcntl() against a socket descriptor is redirected to sfcntl().
- Add support for Berkeley DB once I've finished porting Berkeley DB.
- Write an MPE XS extension library containing miscellaneous important MPE functions like GETPRIVMODE(), GETUSERMODE(), and sfcntl().

115.14 Perl/iX Change History

May 6, 1999

- Patch LBCJXT6A is required on MPE/iX 5.5 machines in order to prevent Perl/iX from dying with an unresolved external reference to _getenv_libc.

April 7, 1999

- Updated to version 5.005_03.
- The official source distribution once again compiles "straight out of the box" for MPE.
- The current incarnation of the 5.5 POSIX filename extended characters patch is now MPEKX40B.
- The LIBSHP3K *.a -> *.sl library conversion script is now included as /PERL/PUB/LIBSHP3K.

November 20, 1998

- Updated to version 5.005_02.
- Fixed a DynaLoader bug that was unable to load symbols from relative path name libraries.
- Fixed a .xs compilation bug where the mpeixish.sh include file wasn't being installed into the proper directory.
- All bugfixes will be submitted back to the official Perl developers.
- The current incarnation of the POSIX filename extended characters patch is now MPEKXJ3A.

August 14, 1998

- The previous POSIX filename extended characters patch MPEKX44C has been superseded by MPEKXB5A.

August 7, 1998

- The previous POSIX filename extended characters patch MPEKX76A has been superseded by MPEKX44C.

July 28, 1998

- Updated to version 5.005_01.

July 23, 1998

- Updated to version 5.005 (production release). The public freeware sources are now 100% MPE-ready "straight out of the box".

July 17, 1998

- Updated to version 5.005b1 (public beta release). The public freeware sources are now 99.9% MPE-ready. By installing and testing this beta on your own HP3000, you will be helping to insure that the final release of 5.005 will be 100% MPE-ready and 100% bug free.
- My MPE binary release is now extracted using my standard INSTALL script.

July 15, 1998

- Changed startperl to #!/PERL/PUB/perl so that Perl will recognize scripts more easily and efficiently.

July 8, 1998

- Updated to version 5.004_70 (internal developer release) which is now MPE-ready. The next public freeware release of Perl should compile "straight out of the box" on MPE. Note that this version of Perl/iX was strictly internal to me and never publicly released. Note that [21]BIND/iX is now required (well, the include files and libbind.a) if you wish to compile Perl/iX.

November 6, 1997

- Updated to version 5.004_04. No changes in MPE-specific functionality.

October 16, 1997

- Added Demos section to the Perl/iX home page so you can see some sample Perl applications running on my 3000.

October 3, 1997

- Added System Requirements section to the Perl/iX home page just so the prerequisites stand out more. Various other home page tweaks.

October 2, 1997

- Initial public release.

September 1997

- Porting begins.

115.15 AUTHOR

Mark Bixby, http://www.bixby.org/mark/

Chapter 116

perlnetware

Perl for NetWare

116.1 DESCRIPTION

This file gives instructions for building Perl 5.7 and above, and also Perl modules for NetWare. Before you start, you may want to read the README file found in the top level directory into which the Perl source code distribution was extracted. Make sure you read and understand the terms under which the software is being distributed.

116.2 BUILD

This section describes the steps to be performed to build a Perl NLM and other associated NLMs.

116.2.1 Tools & SDK

The build requires CodeWarrior compiler and linker. In addition, the "NetWare SDK", "NLM & NetWare Libraries for C" and "NetWare Server Protocol Libraries for C", all available at http://developer.novell.com/ndk/, are also required. Microsoft Visual C++ version 4.2 or later is also required.

116.2.2 Setup

The build process is dependent on the location of the NetWare SDK. Once the Tools & SDK are installed, the build environment has to be setup. The following batch files setup the environment.

SetNWBld.bat

The Execution of this file takes 2 parameters as input. The first being the NetWare SDK path, second being the path for CodeWarrior Compiler & tools. Execution of this file sets these paths and also sets the build type to Release by default.

Buildtype.bat

This is used to set the build type to debug or release. Change the build type only after executing SetNWBld.bat

*****

Example: 1. Typing "buildtype d on" at the command prompt causes the buildtype to be set to Debug type with D2 flag set. 2. Typing "buildtype d off" or "buildtype d" at the command prompt causes the buildtype to be set to Debug type with D1 flag set. 2. Typing "buildtype r" at the command prompt sets it to Release Build type.

116.2.3 Make

The make process runs only under WinNT shell. The NetWare makefile is located under the NetWare folder. This makes use of miniperl.exe to run some of the Perl scripts. To create miniperl.exe, first set the required paths for Visual c++ compilier (specify vcvars32 location) at the command prompt. Then run nmake from win32 folder through WinNT command prompt. The build process can be stopped after miniperl.exe is created. Then run nmake from NetWare folder through WinNT command prompt.

Currently the following two build types are tested on NetWare:

- USE_MULTI, USE_ITHREADS & USE_IMP_SYS defined
- USE_MULTI & USE_IMP_SYS defined and USE_ITHREADS not defined

116.2.4 Interpreter

Once miniperl.exe creation is over, run nmake from the NetWare folder. This will build the Perl interpreter for NetWare as *perl.nlm*. This is copied under the *Release* folder if you are doing a release build, else will be copied under *Debug* folder for debug builds.

116.2.5 Extensions

The make process also creates the Perl extensions as <*Extension*.nlm>

116.3 INSTALL

To install NetWare Perl onto a NetWare server, first map the Sys volume of a NetWare server to *i:*. This is because the makefile by default sets the drive letter to *i:*. Type *nmake nwinstall* from NetWare folder on a WinNT command prompt. This will copy the binaries and module files onto the NetWare server under *sys:\Perl* folder. The Perl interpreter, *perl.nlm*, is copied under *sys:\perl\system* folder. Copy this to *sys:\system* folder.

Example: At the command prompt Type "nmake nwinstall". This will install NetWare Perl on the NetWare Server. Similiarly if you type "nmake install", This will cause the binaries to be installed on the local machine. (Typically under the c:\perl folder)

116.4 BUILD NEW EXTENSIONS

To build extensions other than standard extensions, NetWare Perl has to be installed on Windows along with Windows Perl. The Perl for Windows can be either downloaded from the CPAN site and built using the sources, or the binaries can be directly downloaded from the ActiveState site. Installation can be done by invoking *nmake install* from the NetWare folder on a WinNT command prompt after building NetWare Perl by following steps given above. This will copy all the *.pm files and other required files. Documentation files are not copied. Thus one must first install Windows Perl, Then install NetWare Perl.

Once this is done, do the following to build any extension:

- Change to the extension directory where its source files are present.
- Run the following command at the command prompt:

  ```
  perl -II<path to NetWare lib dir> -II<path to lib> Makefile.pl
  ```

  Example:

  ```
  perl -Ic:/perl/5.6.1/lib/NetWare-x86-multi-thread -Ic:\perl\5.6.1\lib MakeFile.pl
  ```

or

```
perl -Ic:/perl/5.8.0/lib/NetWare-x86-multi-thread -Ic:\perl\5.8.0\lib MakeFile.pl
```

- nmake
- nmake install

  Install will copy the files into the Windows machine where NetWare Perl is installed and these files may have to be copied to the NetWare server manually. Alternatively, pass *INSTALLSITELIB=i:\perl\lib* as an input to makefile.pl above. Here *i:* is the mapped drive to the sys: volume of the server where Perl on NetWare is installed. Now typing *nmake install*, will copy the files onto the NetWare server.

  Example: You can execute the following on the command prompt.

```
perl -Ic:/perl/5.6.1/lib/NetWare-x86-multi-thread -Ic:\perl\5.6.1\lib MakeFile.pl
INSTALLSITELIB=i:\perl\lib
```

or

```
perl -Ic:/perl/5.8.0/lib/NetWare-x86-multi-thread -Ic:\perl\5.8.0\lib MakeFile.pl
INSTALLSITELIB=i:\perl\lib
```

- Note: Some modules downloaded from CPAN may require NetWare related API in order to build on NetWare. Other modules may however build smoothly with or without minor changes depending on the type of module.

116.5 ACKNOWLEDGEMENTS

The makefile for Win32 is used as a reference to create the makefile for NetWare. Also, the make process for NetWare port uses miniperl.exe to run scripts during the make and installation process.

116.6 AUTHORS

Anantha Kesari H Y (hyanantha@novell.com) Aditya C (caditya@novell.com)

116.7 DATE

- Created - 18 Jan 2001
- Modified - 25 June 2001
- Modified - 13 July 2001
- Modified - 28 May 2002

Chapter 117

perlos2

Perl under OS/2, DOS, Win0.3*, Win0.95 and WinNT.

117.1 SYNOPSIS

One can read this document in the following formats:

```
        man perlos2
        view perl perlos2
        explorer perlos2.html
        info perlos2
```

to list some (not all may be available simultaneously), or it may be read *as is*: either as *README.os2*, or *pod/perlos2.pod*.

To read the *.INF* version of documentation (**very** recommended) outside of OS/2, one needs an IBM's reader (may be available on IBM ftp sites (?) (URL anyone?)) or shipped with PC DOS 7.0 and IBM's Visual Age C++ 3.5.

A copy of a Win* viewer is contained in the "Just add OS/2 Warp" package

```
  ftp://ftp.software.ibm.com/ps/products/os2/tools/jaow/jaow.zip
```

in *?:\JUST_ADD\view.exe*. This gives one an access to EMX's *.INF* docs as well (text form is available in */emx/doc* in EMX's distribution). There is also a different viewer named xview.

Note that if you have *lynx.exe* or *netscape.exe* installed, you can follow WWW links from this document in *.INF* format. If you have EMX docs installed correctly, you can follow library links (you need to have `view emxbook` working by setting `EMXBOOK` environment variable as it is described in EMX docs).

117.2 DESCRIPTION

117.2.1 Target

The target is to make OS/2 one of the best supported platform for using/building/developing Perl and *Perl applications*, as well as make Perl the best language to use under OS/2. The secondary target is to try to make this work under DOS and Win* as well (but not **too** hard).

The current state is quite close to this target. Known limitations:

- Some *nix programs use fork() a lot; with the mostly useful flavors of perl for OS/2 (there are several built simultaneously) this is supported; but some flavors do not support this (e.g., when Perl is called from inside REXX). Using fork() after *use*ing dynamically loading extensions would not work with *very* old versions of EMX.

- You need a separate perl executable *perl__.exe* (see perl__.exe) if you want to use PM code in your application (as Perl/Tk or OpenGL Perl modules do) without having a text-mode window present.

  While using the standard *perl.exe* from a text-mode window is possible too, I have seen cases when this causes degradation of the system stability. Using *perl__.exe* avoids such a degradation.

- There is no simple way to access WPS objects. The only way I know is via `OS2::REXX` and `SOM` extensions (see *OS2::REXX*, *Som*). However, we do not have access to convenience methods of Object-REXX. (Is it possible at all? I know of no Object-REXX API.) The `SOM` extension (currently in alpha-text) may eventually remove this shortcoming; however, due to the fact that DII is not supported by the `SOM` module, using `SOM` is not as convenient as one would like it.

Please keep this list up-to-date by informing me about other items.

117.2.2 Other OSes

Since OS/2 port of perl uses a remarkable EMX environment, it can run (and build extensions, and - possibly - be built itself) under any environment which can run EMX. The current list is DOS, DOS-inside-OS/2, Win0.3*, Win0.95 and WinNT. Out of many perl flavors, only one works, see §117.11.4.

Note that not all features of Perl are available under these environments. This depends on the features the *extender* - most probably RSX - decided to implement.

Cf. *Prerequisites*.

117.2.3 Prerequisites

EMX

EMX runtime is required (may be substituted by RSX). Note that it is possible to make *perl_.exe* to run under DOS without any external support by binding *emx.exe*/*rsx.exe* to it, see *emxbind*. Note that under DOS for best results one should use RSX runtime, which has much more functions working (like `fork`, `popen` and so on). In fact RSX is required if there is no VCPI present. Note the RSX requires DPMI. Many implementations of DPMI are known to be very buggy, beware!

Only the latest runtime is supported, currently `0.9d fix 03`. Perl may run under earlier versions of EMX, but this is not tested.

One can get different parts of EMX from, say

```
http://www.leo.org/pub/comp/os/os2/leo/gnu/emx+gcc/
http://powerusersbbs.com/pub/os2/dev/   [EMX+GCC Development]
http://hobbes.nmsu.edu/pub/os2/dev/emx/v0.9d/
```

The runtime component should have the name *emxrt.zip*.

NOTE. When using *emx.exe*/*rsx.exe*, it is enough to have them on your path. One does not need to specify them explicitly (though this

```
emx perl_.exe -de 0
```

will work as well.)

RSX

To run Perl on DPMI platforms one needs RSX runtime. This is needed under DOS-inside-OS/2, Win0.3*, Win0.95 and WinNT (see §117.2.2). RSX would not work with VCPI only, as EMX would, it requires DMPI.

Having RSX and the latest *sh.exe* one gets a fully functional ***nix**-ish environment under DOS, say, `fork`, “ and pipe-`open` work. In fact, MakeMaker works (for static build), so one can have Perl development environment under DOS.

One can get RSX from, say

```
ftp://ftp.cdrom.com/pub/os2/emx09c/contrib
ftp://ftp.uni-bielefeld.de/pub/systems/msdos/misc
ftp://ftp.leo.org/pub/comp/os/os2/leo/devtools/emx+gcc/contrib
```

Contact the author on `rainer@mathematik.uni-bielefeld.de`.

The latest *sh.exe* with DOS hooks is available in

```
http://www.ilyaz.org/software/os2/
```

as *sh_dos.zip* or under similar names starting with `sh`, `pdksh` etc.

HPFS

Perl does not care about file systems, but the perl library contains many files with long names, so to install it intact one needs a file system which supports long file names.

Note that if you do not plan to build the perl itself, it may be possible to fool EMX to truncate file names. This is not supported, read EMX docs to see how to do it.

pdksh

To start external programs with complicated command lines (like with pipes in between, and/or quoting of arguments), Perl uses an external shell. With EMX port such shell should be named *sh.exe*, and located either in the wired-in-during-compile locations (usually *F:/bin*), or in configurable location (see §117.12.4).

For best results use EMX pdksh. The standard binary (5.2.14 or later) runs under DOS (with *RSX*) as well, see

```
http://www.ilyaz.org/software/os2/
```

117.2.4 Starting Perl programs under OS/2 (and DOS and...)

Start your Perl program *foo.pl* with arguments `arg1 arg2 arg3` the same way as on any other platform, by

```
perl foo.pl arg1 arg2 arg3
```

If you want to specify perl options `-my_opts` to the perl itself (as opposed to your program), use

```
perl -my_opts foo.pl arg1 arg2 arg3
```

Alternately, if you use OS/2-ish shell, like CMD or 4os2, put the following at the start of your perl script:

```
extproc perl -S -my_opts
```

rename your program to *foo.cmd*, and start it by typing

```
foo arg1 arg2 arg3
```

Note that because of stupid OS/2 limitations the full path of the perl script is not available when you use `extproc`, thus you are forced to use `-S` perl switch, and your script should be on the `PATH`. As a plus side, if you know a full path to your script, you may still start it with

```
perl ../../blah/foo.cmd arg1 arg2 arg3
```

(note that the argument `-my_opts` is taken care of by the `extproc` line in your script, see `extproc` on the first line).
To understand what the above *magic* does, read perl docs about `-S` switch - see *perlrun*, and cmdref about `extproc`:

```
view perl perlrun
man perlrun
view cmdref extproc
help extproc
```

or whatever method you prefer.
There are also endless possibilities to use *executable extensions* of 4os2, *associations* of WPS and so on... However, if you use *nixish shell (like *sh.exe* supplied in the binary distribution), you need to follow the syntax specified in Switches in *perlrun*.
Note that **-S** switch supports scripts with additional extensions *.cmd*, *.btm*, *.bat*, *.pl* as well.

117.2.5 Starting OS/2 (and DOS) programs under Perl

This is what system() (see system in *perlfunc*), " (see I/O Operators in *perlop*), and *open pipe* (see open in *perlfunc*) are for. (Avoid exec() (see exec in *perlfunc*) unless you know what you do).

Note however that to use some of these operators you need to have a sh-syntax shell installed (see §**??**, §117.3), and perl should be able to find it (see §117.12.4).

The cases when the shell is used are:

1. One-argument system() (see system in *perlfunc*), exec() (see exec in *perlfunc*) with redirection or shell meta-characters;
2. Pipe-open (see open in *perlfunc*) with the command which contains redirection or shell meta-characters;
3. Backticks " (see I/O Operators in *perlop*) with the command which contains redirection or shell meta-characters;
4. If the executable called by system()/exec()/pipe-open()/" is a script with the "magic" `#!` line or `extproc` line which specifies shell;
5. If the executable called by system()/exec()/pipe-open()/" is a script without "magic" line, and `$ENV{EXECSHELL}` is set to shell;
6. If the executable called by system()/exec()/pipe-open()/" is not found (is not this remark obsolete?);
7. For globbing (see glob in *perlfunc*, I/O Operators in *perlop*) (obsolete? Perl uses builtin globbing nowadays...).

For the sake of speed for a common case, in the above algorithms backslashes in the command name are not considered as shell metacharacters.

Perl starts scripts which begin with cookies `extproc` or `#!` directly, without an intervention of shell. Perl uses the same algorithm to find the executable as *pdksh*: if the path on `#!` line does not work, and contains `/`, then the directory part of the executable is ignored, and the executable is searched in . and on `PATH`. To find arguments for these scripts Perl uses a different algorithm than *pdksh*: up to 3 arguments are recognized, and trailing whitespace is stripped.

If a script does not contain such a cooky, then to avoid calling *sh.exe*, Perl uses the same algorithm as *pdksh*: if `$ENV{EXECSHELL}` is set, the script is given as the first argument to this command, if not set, then `$ENV{COMSPEC} /c` is used (or a hardwired guess if `$ENV{COMSPEC}` is not set).

When starting scripts directly, Perl uses exactly the same algorithm as for the search of script given by **-S** command-line option: it will look in the current directory, then on components of `$ENV{PATH}` using the following order of appended extensions: no extension, *.cmd*, *.btm*, *.bat*, *.pl*.

Note that Perl will start to look for scripts only if OS/2 cannot start the specified application, thus `system 'blah'` will not look for a script if there is an executable file *blah.exe anywhere* on `PATH`. In other words, `PATH` is essentially searched twice: once by the OS for an executable, then by Perl for scripts.

Note also that executable files on OS/2 can have an arbitrary extension, but *.exe* will be automatically appended if no dot is present in the name. The workaround is as simple as that: since *blah.* and *blah* denote the same file (at list on FAT and HPFS file systems), to start an executable residing in file *n:/bin/blah* (no extension) give an argument `n:/bin/blah.` (dot appended) to system().

Perl will start PM programs from VIO (=text-mode) Perl process in a separate PM session; the opposite is not true: when you start a non-PM program from a PM Perl process, Perl would not run it in a separate session. If a separate session is desired, either ensure that shell will be used, as in `system 'cmd /c myprog'`, or start it using optional arguments to system() documented in `OS2::Process` module. This is considered to be a feature.

117.3 Frequently asked questions

117.3.1 "It does not work"

Perl binary distributions come with a *testperl.cmd* script which tries to detect common problems with misconfigured installations. There is a pretty large chance it will discover which step of the installation you managed to goof. `;-)`

117.3.2 I cannot run external programs

- Did you run your programs with `-w` switch? See Starting OS/2 (and DOS) programs under Perl.
- Do you try to run *internal* shell commands, like `` `copy a b` `` (internal for *cmd.exe*), or `` `glob a*b` `` (internal for ksh)? You need to specify your shell explicitly, like `` `cmd /c copy a b` ``, since Perl cannot deduce which commands are internal to your shell.

117.3.3 I cannot embed perl into my program, or use *perl.dll* from my program.

Is your program EMX-compiled with `-Zmt -Zcrtdll`?

Well, nowadays Perl DLL should be usable from a differently compiled program too... If you can run Perl code from REXX scripts (see *OS2::REXX*), then there are some other aspect of interaction which are overlooked by the current hackish code to support differently-compiled principal programs.

If everything else fails, you need to build a stand-alone DLL for perl. Contact me, I did it once. Sockets would not work, as a lot of other stuff.

Did you use *ExtUtils::Embed*?

Some time ago I had reports it does not work. Nowadays it is checked in the Perl test suite, so grep *./t* subdirectory of the build tree (as well as **.t* files in the *./lib* subdirectory) to find how it should be done "correctly".

117.3.4 ‘ and pipe-open do not work under DOS.

This may a variant of just §117.3.2, or a deeper problem. Basically: you *need* RSX (see §117.6.2) for these commands to work, and you may need a port of *sh.exe* which understands command arguments. One of such ports is listed in §117.6.2 under RSX. Do not forget to set variable §`117.12.4` as well.

DPMI is required for RSX.

117.3.5 Cannot start `find.exe "pattern" file`

The whole idea of the "standard C API to start applications" is that the forms `foo` and `"foo"` of program arguments are completely interchangable. *find* breaks this paradigm;

```
find "pattern" file
find pattern file
```

are not equivalent; *find* cannot be started directly using the above API. One needs a way to surround the doublequotes in some other quoting construction, necessarily having an extra non-Unixish shell in between.

Use one of

```
system 'cmd', '/c', 'find "pattern" file';
`cmd /c 'find "pattern" file'`
```

This would start *find.exe* via *cmd.exe* via `sh.exe` via `perl.exe`, but this is a price to pay if you want to use non-conforming program.

117.4 INSTALLATION

117.4.1 Automatic binary installation

The most convenient way of installing a binary distribution of perl is via perl installer *install.exe*. Just follow the instructions, and 99% of the installation blues would go away.

Note however, that you need to have *unzip.exe* on your path, and EMX environment *running*. The latter means that if you just installed EMX, and made all the needed changes to *Config.sys*, you may need to reboot in between. Check EMX runtime by running

```
        emxrev
```

Binary installer also creates a folder on your desktop with some useful objects. If you need to change some aspects of the work of the binary installer, feel free to edit the file *Perl.pkg*. This may be useful e.g., if you need to run the installer many times and do not want to make many interactive changes in the GUI.

Things not taken care of by automatic binary installation:

`PERL_BADLANG`

may be needed if you change your codepage *after* perl installation, and the new value is not supported by EMX. See §117.12.2.

`PERL_BADFREE`

see §117.12.3.

Config.pm

This file resides somewhere deep in the location you installed your perl library, find it out by

```
  perl -MConfig -le "print $INC{'Config.pm'}"
```

While most important values in this file *are* updated by the binary installer, some of them may need to be hand-edited. I know no such data, please keep me informed if you find one. Moreover, manual changes to the installed version may need to be accompanied by an edit of this file.

NOTE. Because of a typo the binary installer of 5.00305 would install a variable `PERL_SHPATH` into *Config.sys*. Please remove this variable and put PERL_SH_DIR instead.

117.4.2 Manual binary installation

As of version 5.00305, OS/2 perl binary distribution comes split into 11 components. Unfortunately, to enable configurable binary installation, the file paths in the zip files are not absolute, but relative to some directory.

Note that the extraction with the stored paths is still necessary (default with unzip, specify `-d` to pkunzip). However, you need to know where to extract the files. You need also to manually change entries in *Config.sys* to reflect where did you put the files. Note that if you have some primitive unzipper (like `pkunzip`), you may get a lot of warnings/errors during unzipping. Upgrade to `(w)unzip`.

Below is the sample of what to do to reproduce the configuration on my machine. In *VIEW.EXE* you can press `Ctrl-Insert` now, and cut-and-paste from the resulting file - created in the directory you started *VIEW.EXE* from.

For each component, we mention environment variables related to each installation directory. Either choose directories to match your values of the variables, or create/append-to variables to take into account the directories.

Perl VIO and PM executables (dynamically linked)

```
  unzip perl_exc.zip *.exe *.ico -d f:/emx.add/bin
  unzip perl_exc.zip *.dll -d f:/emx.add/dll
```

(have the directories with `*.exe` on PATH, and `*.dll` on LIBPATH);

Perl_ VIO executable (statically linked)

```
unzip perl_aou.zip -d f:/emx.add/bin
```

(have the directory on PATH);

Executables for Perl utilities

```
unzip perl_utl.zip -d f:/emx.add/bin
```

(have the directory on PATH);

Main Perl library

```
unzip perl_mlb.zip -d f:/perllib/lib
```

If this directory is exactly the same as the prefix which was compiled into *perl.exe*, you do not need to change anything. However, for perl to find the library if you use a different path, you need to `set PERLLIB_PREFIX` in *Config.sys*, see §117.12.1.

Additional Perl modules

```
unzip perl_ste.zip -d f:/perllib/lib/site_perl/5.8.5/
```

Same remark as above applies. Additionally, if this directory is not one of directories on @INC (and @INC is influenced by `PERLLIB_PREFIX`), you need to put this directory and subdirectory *./os2* in `PERLLIB` or `PERL5LIB` variable. Do not use `PERL5LIB` unless you have it set already. See ENVIRONMENT in *perl*.

[Check whether this extraction directory is still applicable with the new directory structure layout!]

Tools to compile Perl modules

```
unzip perl_blb.zip -d f:/perllib/lib
```

Same remark as for *perl_ste.zip*.

Manpages for Perl and utilities

```
unzip perl_man.zip -d f:/perllib/man
```

This directory should better be on `MANPATH`. You need to have a working *man* to access these files.

Manpages for Perl modules

```
unzip perl_mam.zip -d f:/perllib/man
```

This directory should better be on `MANPATH`. You need to have a working man to access these files.

Source for Perl documentation

```
unzip perl_pod.zip -d f:/perllib/lib
```

This is used by the `perldoc` program (see *perldoc*), and may be used to generate HTML documentation usable by WWW browsers, and documentation in zillions of other formats: `info`, `LaTeX`, `Acrobat`, `FrameMaker` and so on. [Use programs such as *pod2latex* etc.]

Perl manual in *.INF* format

```
unzip perl_inf.zip -d d:/os2/book
```

This directory should better be on BOOKSHELF.

Pdksh

```
unzip perl_sh.zip -d f:/bin
```

This is used by perl to run external commands which explicitly require shell, like the commands using *redirection* and *shell metacharacters*. It is also used instead of explicit */bin/sh*.

Set PERL_SH_DIR (see §117.12.4) if you move *sh.exe* from the above location.

Note. It may be possible to use some other sh-compatible shell (untested).

After you installed the components you needed and updated the *Config.sys* correspondingly, you need to hand-edit *Config.pm*. This file resides somewhere deep in the location you installed your perl library, find it out by

```
perl -MConfig -le "print $INC{'Config.pm'}"
```

You need to correct all the entries which look like file paths (they currently start with f:/).

117.4.3 Warning

The automatic and manual perl installation leave precompiled paths inside perl executables. While these paths are overwriteable (see §117.12.1, §117.12.4), some people may prefer binary editing of paths inside the executables/DLLs.

117.5 Accessing documentation

Depending on how you built/installed perl you may have (otherwise identical) Perl documentation in the following formats:

117.5.1 OS/2 .*INF* file

Most probably the most convenient form. Under OS/2 view it as

```
view perl
view perl perlfunc
view perl less
view perl ExtUtils::MakeMaker
```

(currently the last two may hit a wrong location, but this may improve soon). Under Win* see §117.1.

If you want to build the docs yourself, and have *OS/2 toolkit*, run

```
pod2ipf > perl.ipf
```

in */perllib/lib/pod* directory, then

```
ipfc /inf perl.ipf
```

(Expect a lot of errors during the both steps.) Now move it on your BOOKSHELF path.

117.5.2 Plain text

If you have perl documentation in the source form, perl utilities installed, and GNU groff installed, you may use

```
        perldoc perlfunc
        perldoc less
        perldoc ExtUtils::MakeMaker
```

to access the perl documentation in the text form (note that you may get better results using perl manpages).
Alternately, try running pod2text on *.pod* files.

117.5.3 Manpages

If you have *man* installed on your system, and you installed perl manpages, use something like this:

```
        man perlfunc
        man 3 less
        man ExtUtils.MakeMaker
```

to access documentation for different components of Perl. Start with

```
        man perl
```

Note that dot (.) is used as a package separator for documentation for packages, and as usual, sometimes you need to give the section - `3` above - to avoid shadowing by the *less(1) manpage*.
Make sure that the directory **above** the directory with manpages is on our `MANPATH`, like this

```
  set MANPATH=c:/man;f:/perllib/man
```

for Perl manpages in `f:/perllib/man/man1/` etc.

117.5.4 HTML

If you have some WWW browser available, installed the Perl documentation in the source form, and Perl utilities, you can build HTML docs. Cd to directory with *.pod* files, and do like this

```
        cd f:/perllib/lib/pod
        pod2html
```

After this you can direct your browser the file *perl.html* in this directory, and go ahead with reading docs, like this:

```
        explore file:///f:/perllib/lib/pod/perl.html
```

Alternatively you may be able to get these docs prebuilt from CPAN.

117.5.5 GNU `info` files

Users of Emacs would appreciate it very much, especially with `CPerl` mode loaded. You need to get latest `pod2texi` from `CPAN`, or, alternately, the prebuilt info pages.

117.5.6 *PDF* files

for `Acrobat` are available on CPAN (may be for slightly older version of perl).

117.5.7 LaTeX docs

can be constructed using `pod2latex`.

117.6 BUILD

Here we discuss how to build Perl under OS/2. There is an alternative (but maybe older) view on http://www.shadow.net/˜troc/os2perl.html.

117.6.1 The short story

Assume that you are a seasoned porter, so are sure that all the necessary tools are already present on your system, and you know how to get the Perl source distribution. Untar it, change to the extract directory, and

```
  gnupatch -p0 < os2\diff.configure
  sh Configure -des -D prefix=f:/perllib
  make
  make test
  make install
  make aout_test
  make aout_install
```

This puts the executables in f:/perllib/bin. Manually move them to the `PATH`, manually move the built *perl*.dll* to `LIBPATH` (here for Perl DLL * is a not-very-meaningful hex checksum), and run

```
  make installcmd INSTALLCMDDIR=d:/ir/on/path
```

Assuming that the `man`-files were put on an appropriate location, this completes the installation of minimal Perl system. (The binary distribution contains also a lot of additional modules, and the documentation in INF format.)

What follows is a detailed guide through these steps.

117.6.2 Prerequisites

You need to have the latest EMX development environment, the full GNU tool suite (gawk renamed to awk, and GNU *find.exe* earlier on path than the OS/2 *find.exe*, same with *sort.exe*, to check use

```
  find --version
  sort --version
```

). You need the latest version of *pdksh* installed as *sh.exe*.

Check that you have **BSD** libraries and headers installed, and - optionally - Berkeley DB headers and libraries, and crypt.

Possible locations to get the files:

```
  ftp://hobbes.nmsu.edu/os2/unix/
  ftp://ftp.cdrom.com/pub/os2/unix/
  ftp://ftp.cdrom.com/pub/os2/dev32/
  ftp://ftp.cdrom.com/pub/os2/emx09c/
```

It is reported that the following archives contain enough utils to build perl: *gnufutil.zip*, *gnusutil.zip*, *gnututil.zip*, *gnused.zip*, *gnupatch.zip*, *gnuawk.zip*, *gnumake.zip*, *gnugrep.zip*, *bsddev.zip* and *ksh527rt.zip* (or a later version). Note that all these utilities are known to be available from LEO:

```
  ftp://ftp.leo.org/pub/comp/os/os2/leo/gnu
```

Note also that the *db.lib* and *db.a* from the EMX distribution are not suitable for multi-threaded compile (even single-threaded flavor of Perl uses multi-threaded C RTL, for compatibility with XFree86-OS/2). Get a corrected one from

```
http://www.ilyaz.org/software/os2/db_mt.zip
```

If you have *exactly the same version of Perl* installed already, make sure that no copies or perl are currently running. Later steps of the build may fail since an older version of *perl.dll* loaded into memory may be found. Running `make test` becomes meaningless, since the test are checking a previous build of perl (this situation is detected and reported by *lib/os2_base.t* test). Do not forget to unset `PERL_EMXLOAD_SEC` in environment.

Also make sure that you have */tmp* directory on the current drive, and . directory in your `LIBPATH`. One may try to correct the latter condition by

```
set BEGINLIBPATH .\.
```

if you use something like *CMD.EXE* or latest versions of *4os2.exe*. (Setting BEGINLIBPATH to just `.` is ignored by the OS/2 kernel.)

Make sure your gcc is good for `-Zomf` linking: run `omflibs` script in */emx/lib* directory.

Check that you have link386 installed. It comes standard with OS/2, but may be not installed due to customization. If typing

```
link386
```

shows you do not have it, do *Selective install*, and choose `Link object modules` in *Optional system utilities/More*. If you get into link386 prompts, press `Ctrl-C` to exit.

117.6.3 Getting perl source

You need to fetch the latest perl source (including developers releases). With some probability it is located in

```
http://www.cpan.org/src/5.0
http://www.cpan.org/src/5.0/unsupported
```

If not, you may need to dig in the indices to find it in the directory of the current maintainer.

Quick cycle of developers release may break the OS/2 build time to time, looking into

```
http://www.cpan.org/ports/os2/
```

may indicate the latest release which was publicly released by the maintainer. Note that the release may include some additional patches to apply to the current source of perl.

Extract it like this

```
tar vzxf perl5.00409.tar.gz
```

You may see a message about errors while extracting *Configure*. This is because there is a conflict with a similarly-named file *configure*.

Change to the directory of extraction.

117.6.4 Application of the patches

You need to apply the patches in *./os2/diff.** like this:

```
  gnupatch -p0 < os2\diff.configure
```

You may also need to apply the patches supplied with the binary distribution of perl. It also makes sense to look on the perl5-porters mailing list for the latest OS/2-related patches (see http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/). Such patches usually contain strings `/os2/` and `patch`, so it makes sense looking for these strings.

117.6.5 Hand-editing

You may look into the file *./hints/os2.sh* and correct anything wrong you find there. I do not expect it is needed anywhere.

117.6.6 Making

```
  sh Configure -des -D prefix=f:/perllib
```

`prefix` means: where to install the resulting perl library. Giving correct prefix you may avoid the need to specify `PERLLIB_PREFIX`, see §117.12.1.

Ignore the message about missing `ln`, and about `-c` option to tr. The latter is most probably already fixed, if you see it and can trace where the latter spurious warning comes from, please inform me.

Now

```
  make
```

At some moment the built may die, reporting a *version mismatch* or *unable to run* perl. This means that you do not have . in your LIBPATH, so *perl.exe* cannot find the needed *perl67B2.dll* (treat these hex digits as line noise). After this is fixed the build should finish without a lot of fuss.

117.6.7 Testing

Now run

```
  make test
```

All tests should succeed (with some of them skipped). If you have the same version of Perl installed, it is crucial that you have `.` early in your LIBPATH (or in BEGINLIBPATH), otherwise your tests will most probably test the wrong version of Perl.

Some tests may generate extra messages similar to

A lot of bad `free`

in database tests related to Berkeley DB. *This should be fixed already.* If it persists, you may disable this warnings, see §117.12.3.

Process terminated by SIGTERM/SIGINT

This is a standard message issued by OS/2 applications. *nix applications die in silence. It is considered to be a feature. One can easily disable this by appropriate sighandlers.

However the test engine bleeds these message to screen in unexpected moments. Two messages of this kind *should* be present during testing.

To get finer test reports, call

```
  perl t/harness
```

The report with *io/pipe.t* failing may look like this:

```
  Failed Test  Status Wstat Total Fail  Failed  List of failed
  ------------------------------------------------------------
  io/pipe.t                    12    1   8.33%  9
  7 tests skipped, plus 56 subtests skipped.
  Failed 1/195 test scripts, 99.49% okay. 1/6542 subtests failed, 99.98% okay.
```

The reasons for most important skipped tests are:

op/fs.t

1. Checks `atime` and `mtime` of `stat()` - unfortunately, HPFS provides only 2sec time granularity (for compatibility with FAT?).
2. Checks `truncate()` on a filehandle just opened for write - I do not know why this should or should not work.

op/stat.t

Checks `stat()`. Tests:

1. Checks `atime` and `mtime` of `stat()` - unfortunately, HPFS provides only 2sec time granularity (for compatibility with FAT?).

117.6.8 Installing the built perl

If you haven't yet moved `perl*.dll` onto LIBPATH, do it now.
Run

```
  make install
```

It would put the generated files into needed locations. Manually put *perl.exe*, *perl__.exe* and *perl___.exe* to a location on your PATH, *perl.dll* to a location on your LIBPATH.
Run

```
  make installcmd INSTALLCMDDIR=d:/ir/on/path
```

to convert perl utilities to *.cmd* files and put them on PATH. You need to put *.EXE*-utilities on path manually. They are installed in `$prefix/bin`, here `$prefix` is what you gave to *Configure*, see *Making*.
If you use `man`, either move the installed **/man/* directories to your `MANPATH`, or modify `MANPATH` to match the location. (One could have avoided this by providing a correct `manpath` option to *./Configure*, or editing *./config.sh* between configuring and making steps.)

117.6.9 `a.out`-style build

Proceed as above, but make *perl_.exe* (see §117.11.4) by

```
  make perl_
```

test and install by

```
  make aout_test
  make aout_install
```

Manually put *perl_.exe* to a location on your PATH.
Note. The build process for `perl_` *does not know* about all the dependencies, so you should make sure that anything is up-to-date, say, by doing

```
  make perl_dll
```

first.

117.7 Building a binary distribution

[This section provides a short overview only...]

Building should proceed differently depending on whether the version of perl you install is already present and used on your system, or is a new version not yet used. The description below assumes that the version is new, so installing its DLLs and *.pm* files will not disrupt the operation of your system even if some intermediate steps are not yet fully working.

The other cases require a little bit more convoluted procedures. Below I suppose that the current version of Perl is `5.8.2`, so the executables are named accordingly.

1. Fully build and test the Perl distribution. Make sure that no tests are failing with `test` and `aout_test` targets; fix the bugs in Perl and the Perl test suite detected by these tests. Make sure that `all_test` make target runs as clean as possible. Check that `os2/perlrexx.cmd` runs fine.

2. Fully install Perl, including `installcmd` target. Copy the generated DLLs to `LIBPATH`; copy the numbered Perl executables (as in *perl5.8.2.exe*) to `PATH`; copy `perl_.exe` to `PATH` as `perl_5.8.2.exe`. Think whether you need backward-compatibility DLLs. In most cases you do not need to install them yet; but sometime this may simplify the following steps.

3. Make sure that `CPAN.pm` can download files from CPAN. If not, you may need to manually install `Net::FTP`.

4. Install the bundle `Bundle::OS2_default`

   ```
   perl5.8.2 -MCPAN -e "install Bundle::OS2_default" < nul |& tee 00cpan_i_1
   ```

   This may take a couple of hours on 1GHz processor (when run the first time). And this should not be necessarily a smooth procedure. Some modules may not specify required dependencies, so one may need to repeat this procedure several times until the results stabilize.

   ```
   perl5.8.2 -MCPAN -e "install Bundle::OS2_default" < nul |& tee 00cpan_i_2
   perl5.8.2 -MCPAN -e "install Bundle::OS2_default" < nul |& tee 00cpan_i_3
   ```

   Even after they stabilize, some tests may fail.

   Fix as many discovered bugs as possible. Document all the bugs which are not fixed, and all the failures with unknown reasons. Inspect the produced logs *00cpan_i_1* to find suspiciously skipped tests, and other fishy events.

   Keep in mind that *installation* of some modules may fail too: for example, the DLLs to update may be already loaded by *CPAN.pm*. Inspect the `install` logs (in the example above *00cpan_i_1* etc) for errors, and install things manually, as in

   ```
   cd $CPANHOME/.cpan/build/Digest-MD5-2.31
   make install
   ```

   Some distributions may fail some tests, but you may want to install them anyway (as above, or via `force install` command of `CPAN.pm` shell-mode).

   Since this procedure may take quite a long time to complete, it makes sense to "freeze" your CPAN configuration by disabling periodic updates of the local copy of CPAN index: set `index_expire` to some big value (I use 365), then save the settings

   ```
   CPAN> o conf index_expire 365
   CPAN> o conf commit
   ```

   Reset back to the default value `1` when you are finished.

5. When satisfied with the results, rerun the `installcmd` target. Now you can copy `perl5.8.2.exe` to `perl.exe`, and install the other OMF-build executables: `perl__.exe` etc. They are ready to be used.

6. Change to the `./pod` directory of the build tree, download the Perl logo *CamelGrayBig.BMP*, and run

   ```
   ( perl2ipf > perl.ipf ) |& tee 00ipf
   ipfc /INF perl.ipf |& tee 00inf
   ```

   This produces the Perl docs online book `perl.INF`. Install in on `BOOKSHELF` path.

7. Now is the time to build statically linked executable *perl_.exe* which includes newly-installed via `Bundle::OS2_default` modules. Doing testing via `CPAN.pm` is going to be painfully slow, since it statically links a new executable per XS extension.

   Here is a possible workaround: create a toplevel *Makefile.PL* in *$ CPANHOME/.cpan/build/* with contents being (compare with Making executables with a custom collection of statically loaded extensions)

   ```
   use ExtUtils::MakeMaker;
   WriteMakefile NAME => 'dummy';
   ```

   execute this as

   ```
   perl_5.8.2.exe Makefile.PL <nul |& tee 00aout_c1
   make -k all test <nul |& 00aout_t1
   ```

   Again, this procedure should not be absolutely smooth. Some `Makefile.PL`'s in subdirectories may be buggy, and would not run as "child" scripts. The interdependency of modules can strike you; however, since non-XS modules are already installed, the prerequisites of most modules have a very good chance to be present.

   If you discover some glitches, move directories of problematic modules to a different location; if these modules are non-XS modules, you may just ignore them - they are already installed; the remaining, XS, modules you need to install manually one by one.

   After each such removal you need to rerun the `Makefile.PL`/`make` process; usually this procedure converges soon. (But be sure to convert all the necessary external C libraries from *.lib* format to *.a* format: run one of

   ```
   emxaout foo.lib
   emximp -o foo.a foo.lib
   ```

   whichever is appropriate.) Also, make sure that the DLLs for external libraries are usable with with executables compiled without `-Zmtd` options.

   When you are sure that only a few subdirectories lead to failures, you may want to add `-j4` option to `make` to speed up skipping subdirectories with already finished build.

   When you are satisfied with the results of tests, install the build C libraries for extensions:

   ```
   make install |& tee 00aout_i
   ```

   Now you can rename the file *./perl.exe* generated during the last phase to *perl_5.8.2.exe*; place it on `PATH`; if there is an inter-dependency between some XS modules, you may need to repeat the `test`/`install` loop with this new executable and some excluded modules - until the procedure converges.

   Now you have all the necessary *.a* libraries for these Perl modules in the places where Perl builder can find it. Use the perl builder: change to an empty directory, create a "dummy" *Makefile.PL* again, and run

   ```
   perl_5.8.2.exe Makefile.PL |& tee 00c
   make perl                  |& tee 00p
   ```

   This should create an executable *./perl.exe* with all the statically loaded extensions built in. Compare the generated *perlmain.c* files to make sure that during the iterations the number of loaded extensions only increases. Rename *./perl.exe* to *perl_5.8.2.exe* on `PATH`.

   When it converges, you got a functional variant of *perl_5.8.2.exe*; copy it to `perl_.exe`. You are done with generation of the local Perl installation.

8. Make sure that the installed modules are actually installed in the location of the new Perl, and are not inherited from entries of @INC given for inheritance from the older versions of Perl: set `PERLLIB_582_PREFIX` to redirect the new version of Perl to a new location, and copy the installed files to this new location. Redo the tests to make sure that the versions of modules inherited from older versions of Perl are not needed.

   Actually, the log output of *pod2ipf* during the step 6 gives a very detailed info about which modules are loaded from which place; so you may use it as an additional verification tool.

   Check that some temporary files did not make into the perl install tree. Run something like this

   ```
   pfind . -f "!(/\.(pm|pl|ix|al|h|a|lib|txt|pod|imp|bs|dll|ld|bs|inc|xbm|yml|cgi|uu|e2x|skip|packl
   ```

   in the install tree (both top one and *sitelib* one).

   Compress all the DLLs with *lxlite*. The tiny *.exe* can be compressed with `/c:max` (the bug only appears when there is a fixup in the last 6 bytes of a page (?); since the tiny executables are much smaller than a page, the bug will not hit). Do not compress `perl_.exe` - it would not work under DOS.

9. Now you can generate the binary distribution. This is done by running the test of the CPAN distribution `OS2::SoftInstaller`. Tune up the file *test.pl* to suit the layout of current version of Perl first. Do not forget to pack the necessary external DLLs accordingly. Include the description of the bugs and test suite failures you could not fix. Include the small-stack versions of Perl executables from Perl build directory.

   Include *perl5.def* so that people can relink the perl DLL preserving the binary compatibility, or can create compatibility DLLs. Include the diff files (`diff -pu old new`) of fixes you did so that people can rebuild your version. Include *perl5.map* so that one can use remote debugging.

10. Share what you did with the other people. Relax. Enjoy fruits of your work.

11. Brace yourself for thanks, bug reports, hate mail and spam coming as result of the previous step. No good deed should remain unpunished!

117.8 Building custom *.EXE* files

The Perl executables can be easily rebuilt at any moment. Moreover, one can use the *embedding* interface (see *perlembed*) to make very customized executables.

117.8.1 Making executables with a custom collection of statically loaded extensions

It is a little bit easier to do so while *decreasing* the list of statically loaded extensions. We discuss this case only here.

1. Change to an empty directory, and create a placeholder <Makefile.PL>:

   ```
   use ExtUtils::MakeMaker;
   WriteMakefile NAME => 'dummy';
   ```

2. Run it with the flavor of Perl (*perl.exe* or *perl_.exe*) you want to rebuild.

   ```
   perl_ Makefile.PL
   ```

3. Ask it to create new Perl executable:

   ```
   make perl
   ```

   (you may need to manually add `PERLTYPE=-DPERL_CORE` to this commandline on some versions of Perl; the symptom is that the command-line globbing does not work from OS/2 shells with the newly-compiled executable; check with

```
    .\perl.exe -wle "print for @ARGV" *
```

).

4. The previous step created *perlmain.c* which contains a list of newXS() calls near the end. Removing unnecessary calls, and rerunning

   ```
   make perl
   ```

   will produce a customized executable.

117.8.2 Making executables with a custom search-paths

The default perl executable is flexible enough to support most usages. However, one may want something yet more flexible; for example, one may want to find Perl DLL relatively to the location of the EXE file; or one may want to ignore the environment when setting the Perl-library search patch, etc.

If you fill comfortable with *embedding* interface (see *perlembed*), such things are easy to do repeating the steps outlined in Making executables with a custom collection of statically loaded extensions, and doing more comprehensive edits to main() of *perlmain.c*. The people with little desire to understand Perl can just rename main(), and do necessary modification in a custom main() which calls the renamed function in appropriate time.

However, there is a third way: perl DLL exports the main() function and several callbacks to customize the search path. Below is a complete example of a "Perl loader" which

1. Looks for Perl DLL in the directory `$exedir/../dll`;
2. Prepends the above directory to `BEGINLIBPATH`;
3. Fails if the Perl DLL found via `BEGINLIBPATH` is different from what was loaded on step 1; e.g., another process could have loaded it from `LIBPATH` or from a different value of `BEGINLIBPATH`. In these cases one needs to modify the setting of the system so that this other process either does not run, or loads the DLL from `BEGINLIBPATH` with `LIBPATHSTRICT=T` (available with kernels after September 2000).
4. Loads Perl library from `$exedir/../dll/lib/`.
5. Uses Bourne shell from `$exedir/../dll/sh/ksh.exe`.

For best results compile the C file below with the same options as the Perl DLL. However, a lot of functionality will work even if the executable is not an EMX applications, e.g., if compiled with

```
  gcc -Wall -DDOSISH -DOS2=1 -O2 -s -Zomf -Zsys perl-starter.c -DPERL_DLL_BASENAME=\"perl312F\" -Zstack
```

Here is the sample C file:

```
  #define INCL_DOS
  #define INCL_NOPM
  /* These are needed for compile if os2.h includes os2tk.h, not os2emx.h */
  #define INCL_DOSPROCESS
  #include <os2.h>

  #include "EXTERN.h"
  #define PERL_IN_MINIPERLMAIN_C
  #include "perl.h"

  static char *me;
  HMODULE handle;
```

```
static void
die_with(char *msg1, char *msg2, char *msg3, char *msg4)
{
   ULONG c;
   char *s = " error: ";

   DosWrite(2, me, strlen(me), &c);
   DosWrite(2, s, strlen(s), &c);
   DosWrite(2, msg1, strlen(msg1), &c);
   DosWrite(2, msg2, strlen(msg2), &c);
   DosWrite(2, msg3, strlen(msg3), &c);
   DosWrite(2, msg4, strlen(msg4), &c);
   DosWrite(2, "\r\n", 2, &c);
   exit(255);
}

typedef ULONG (*fill_extLibpath_t)(int type, char *pre, char *post, int replace, char *msg);
typedef int (*main_t)(int type, char *argv[], char *env[]);
typedef int (*handler_t)(void* data, int which);

#ifndef PERL_DLL_BASENAME
#  define PERL_DLL_BASENAME "perl"
#endif

static HMODULE
load_perl_dll(char *basename)
{
    char buf[300], fail[260];
    STRLEN l, dirl;
    fill_extLibpath_t f;
    ULONG rc_fullname;
    HMODULE handle, handle1;

    if (_execname(buf, sizeof(buf) - 13) != 0)
        die_with("Can't find full path: ", strerror(errno), "", "");
    /* XXXX Fill 'me' with new value */
    l = strlen(buf);
    while (l && buf[l-1] != '/' && buf[l-1] != '\\')
        l--;
    dirl = l - 1;
    strcpy(buf + l, basename);
    l += strlen(basename);
    strcpy(buf + l, ".dll");
    if ( (rc_fullname = DosLoadModule(fail, sizeof fail, buf, &handle)) != 0
         && DosLoadModule(fail, sizeof fail, basename, &handle) != 0 )
        die_with("Can't load DLL ", buf, "", "");
    if (rc_fullname)
        return handle;                /* was loaded with short name; all is fine */
    if (DosQueryProcAddr(handle, 0, "fill_extLibpath", (PFN*)&f))
        die_with(buf, ": DLL exports no symbol ", "fill_extLibpath", "");
    buf[dirl] = 0;
    if (f(0 /*BEGINLIBPATH*/, buf /* prepend */, NULL /* append */,
          0 /* keep old value */, me))
        die_with(me, ": prepending BEGINLIBPATH", "", "");
    if (DosLoadModule(fail, sizeof fail, basename, &handle1) != 0)
        die_with(me, ": finding perl DLL again via BEGINLIBPATH", "", "");
    buf[dirl] = '\\';
    if (handle1 != handle) {
        if (DosQueryModuleName(handle1, sizeof(fail), fail))
            strcpy(fail, "???");
        die_with(buf, ":\n\tperl DLL via BEGINLIBPATH is different: \n\t",
                 fail,
                 "\n\tYou may need to manipulate global BEGINLIBPATH and LIBPATHSTRICT"
                 "\n\tso that the other copy is loaded via BEGINLIBPATH.");
    }
    return handle;
}

int
main(int argc, char **argv, char **env)
{
    main_t f;
    handler_t h;

    me = argv[0];
    /**/
    handle = load_perl_dll(PERL_DLL_BASENAME);

    if (DosQueryProcAddr(handle, 0, "Perl_OS2_handler_install", (PFN*)&h))
        die_with(PERL_DLL_BASENAME, ": DLL exports no symbol ", "Perl_OS2_handler_install", "");
    if ( !h((void *)"~installprefix", Perlos2_handler_perllib_from)
         || !h((void *)"~dll", Perlos2_handler_perllib_to)
         || !h((void *)"~dll/sh/ksh.exe", Perlos2_handler_perl_sh) )
        die_with(PERL_DLL_BASENAME, ": Can't install @INC manglers", "", "");

    if (DosQueryProcAddr(handle, 0, "dll_perlmain", (PFN*)&f))
        die_with(PERL_DLL_BASENAME, ": DLL exports no symbol ", "dll_perlmain", "");
    return f(argc, argv, env);
}
```

117.9 Build FAQ

117.9.1 Some / became \ in pdksh.

You have a very old pdksh. See *Prerequisites*.

117.9.2 'errno' - unresolved external

You do not have MT-safe *db.lib*. See *Prerequisites*.

117.9.3 Problems with tr or sed

reported with very old version of tr and sed.

117.9.4 Some problem (forget which ;-)

You have an older version of *perl.dll* on your LIBPATH, which broke the build of extensions.

117.9.5 Library ... not found

You did not run `omflibs`. See *Prerequisites*.

117.9.6 Segfault in make

You use an old version of GNU make. See *Prerequisites*.

117.9.7 op/sprintf test failure

This can result from a bug in emx sprintf which was fixed in 0.9d fix 03.

117.10 Specific (mis)features of OS/2 port

117.10.1 `setpriority, getpriority`

Note that these functions are compatible with *nix, not with the older ports of '94 - 95. The priorities are absolute, go from 32 to -95, lower is quicker. 0 is the default priority.

WARNING. Calling `getpriority` on a non-existing process could lock the system before Warp3 fixpak22. Starting with Warp3, Perl will use a workaround: it aborts getpriority() if the process is not present. This is not possible on older versions `2.*`, and has a race condition anyway.

117.10.2 `system()`

Multi-argument form of `system()` allows an additional numeric argument. The meaning of this argument is described in *OS2::Process*.

When finding a program to run, Perl first asks the OS to look for executables on `PATH` (OS/2 adds extension *.exe* if no extension is present). If not found, it looks for a script with possible extensions added in this order: no extension, *.cmd*, *.btm*, *.bat*, *.pl*. If found, Perl checks the start of the file for magic strings `"#!"` and `"extproc "`. If found, Perl uses the rest of the first line as the beginning of the command line to run this script. The only mangling done to the first line is extraction of arguments (currently up to 3), and ignoring of the path-part of the "interpreter" name if it can't be found using the full path.

E.g., `system 'foo', 'bar', 'baz'` may lead Perl to finding *C:/emx/bin/foo.cmd* with the first line being

```
 extproc /bin/bash    -x   -c
```

If */bin/bash.exe* is not found, then Perl looks for an executable *bash.exe* on `PATH`. If found in *C:/emx.add/bin/bash.exe*, then the above system() is translated to

```
  system qw(C:/emx.add/bin/bash.exe -x -c C:/emx/bin/foo.cmd bar baz)
```

One additional translation is performed: instead of */bin/sh* Perl uses the hardwired-or-customized shell (see §`117.12.4`).

The above search for "interpreter" is recursive: if *bash* executable is not found, but *bash.btm* is found, Perl will investigate its first line etc. The only hardwired limit on the recursion depth is implicit: there is a limit 4 on the number of additional arguments inserted before the actual arguments given to system(). In particular, if no additional arguments are specified on the "magic" first lines, then the limit on the depth is 4.

If Perl finds that the found executable is of PM type when the current session is not, it will start the new process in a separate session of necessary type. Call via `OS2::Process` to disable this magic.

WARNING. Due to the described logic, you need to explicitly specify *.com* extension if needed. Moreover, if the executable *perl5.6.1* is requested, Perl will not look for *perl5.6.1.exe*. [This may change in the future.]

117.10.3 `extproc` on the first line

If the first chars of a Perl script are `"extproc "`, this line is treated as `#!`-line, thus all the switches on this line are processed (twice if script was started via cmd.exe). See DESCRIPTION in *perlrun*.

117.10.4 Additional modules:

OS2::Process, *OS2::DLL*, *OS2::REXX*, *OS2::PrfDB*, *OS2::ExtAttr*. These modules provide access to additional numeric argument for `system` and to the information about the running process, to DLLs having functions with REXX signature and to the REXX runtime, to OS/2 databases in the *.INI* format, and to Extended Attributes.

Two additional extensions by Andreas Kaiser, `OS2::UPM`, and `OS2::FTP`, are included into `ILYAZ` directory, mirrored on CPAN. Other OS/2-related extensions are available too.

117.10.5 Prebuilt methods:

`File::Copy::syscopy`

used by `File::Copy::copy`, see *File::Copy*.

`DynaLoader::mod2fname`

used by `DynaLoader` for DLL name mangling.

`Cwd::current_drive()`

Self explanatory.

`Cwd::sys_chdir(name)`

leaves drive as it is.

`Cwd::change_drive(name)`

chanes the "current" drive.

`Cwd::sys_is_absolute(name)`

means has drive letter and is_rooted.

`Cwd::sys_is_rooted(name)`

means has leading `[/\\]` (maybe after a drive-letter:).

`Cwd::sys_is_relative(name)`

means changes with current dir.

`Cwd::sys_cwd(name)`

Interface to cwd from EMX. Used by `Cwd::cwd`.

`Cwd::sys_abspath(name, dir)`

Really really odious function to implement. Returns absolute name of file which would have `name` if CWD were `dir`. `Dir` defaults to the current dir.

`Cwd::extLibpath([type])`

Get current value of extended library search path. If `type` is present and positive, works with `END_LIBPATH`, if negative, works with `LIBPATHSTRICT`, otherwise with `BEGIN_LIBPATH`.

`Cwd::extLibpath_set( path [, type ] )`

Set current value of extended library search path. If `type` is present and positive, works with <END_LIBPATH>, if negative, works with `LIBPATHSTRICT`, otherwise with `BEGIN_LIBPATH`.

`OS2::Error(do_harderror,do_exception)`

Returns `undef` if it was not called yet, otherwise bit 1 is set if on the previous call do_harderror was enabled, bit 2 is set if on previous call do_exception was enabled.

This function enables/disables error popups associated with hardware errors (Disk not ready etc.) and software exceptions.

I know of no way to find out the state of popups *before* the first call to this function.

`OS2::Errors2Drive(drive)`

Returns `undef` if it was not called yet, otherwise return false if errors were not requested to be written to a hard drive, or the drive letter if this was requested.

This function may redirect error popups associated with hardware errors (Disk not ready etc.) and software exceptions to the file POPUPLOG.OS2 at the root directory of the specified drive. Overrides OS2::Error() specified by individual programs. Given argument undef will disable redirection.

Has global effect, persists after the application exits.

I know of no way to find out the state of redirection of popups to the disk *before* the first call to this function.

OS2::SysInfo()

Returns a hash with system information. The keys of the hash are

```
        MAX_PATH_LENGTH, MAX_TEXT_SESSIONS, MAX_PM_SESSIONS,
        MAX_VDM_SESSIONS, BOOT_DRIVE, DYN_PRI_VARIATION,
        MAX_WAIT, MIN_SLICE, MAX_SLICE, PAGE_SIZE,
        VERSION_MAJOR, VERSION_MINOR, VERSION_REVISION,
        MS_COUNT, TIME_LOW, TIME_HIGH, TOTPHYSMEM, TOTRESMEM,
        TOTAVAILMEM, MAXPRMEM, MAXSHMEM, TIMER_INTERVAL,
        MAX_COMP_LENGTH, FOREGROUND_FS_SESSION,
        FOREGROUND_PROCESS
```

OS2::BootDrive()

Returns a letter without colon.

`OS2::MorphPM(serve)`, `OS2::UnMorphPM(serve)`

Transforms the current application into a PM application and back. The argument true means that a real message loop is going to be served. OS2::MorphPM() returns the PM message queue handle as an integer.

See §117.10.10 for additional details.

`OS2::Serve_Messages(force)`

Fake on-demand retrieval of outstanding PM messages. If `force` is false, will not dispatch messages if a real message loop is known to be present. Returns number of messages retrieved.

Dies with "QUITing..." if WM_QUIT message is obtained.

`OS2::Process_Messages(force [, cnt])`

Retrieval of PM messages until window creation/destruction. If `force` is false, will not dispatch messages if a real message loop is known to be present.

Returns change in number of windows. If `cnt` is given, it is incremented by the number of messages retrieved.

Dies with "QUITing..." if WM_QUIT message is obtained.

`OS2::_control87(new,mask)`

the same as _control87(3) of EMX. Takes integers as arguments, returns the previous coprocessor control word as an integer. Only bits in `new` which are present in `mask` are changed in the control word.

OS2::get_control87()

gets the coprocessor control word as an integer.

`OS2::set_control87_em(new=MCW_EM,mask=MCW_EM)`

The variant of OS2::_control87() with default values good for handling exception mask: if no `mask`, uses exception mask part of `new` only. If no `new`, disables all the floating point exceptions.

See §117.10.7 for details.

`OS2::DLLname([how [, \&xsub]])`

Gives the information about the Perl DLL or the DLL containing the C function bound to by `&xsub`. The meaning of `how` is: default (2): full name; 0: handle; 1: module name.

(Note that some of these may be moved to different libraries - eventually).

117.10.6 Prebuilt variables:

$ OS2::emx_rev

numeric value is the same as _emx_rev of EMX, a string value the same as _emx_vprt (similar to `0.9c`).

$ OS2::emx_env

same as _emx_env of EMX, a number similar to 0x8001.

$ OS2::os_ver

a number `OS_MAJOR + 0.001 * OS_MINOR`.

$ OS2::is_aout

true if the Perl library was compiled in AOUT format.

$ OS2::can_fork

true if the current executable is an AOUT EMX executable, so Perl can fork. Do not use this, use the portable check for $Config::Config{dfork}.

$ OS2::nsyserror

This variable (default is 1) controls whether to enforce the contents of $^E to start with `SYS0003`-like id. If set to 0, then the string value of $^E is what is available from the OS/2 message file. (Some messages in this file have an `SYS0003`-like id prepended, some not.)

117.10.7 Misfeatures

- Since *flock*(3) is present in EMX, but is not functional, it is emulated by perl. To disable the emulations, set environment variable `USE_PERL_FLOCK=0`.
- Here is the list of things which may be "broken" on EMX (from EMX docs):
  - The functions *recvmsg*(3), *sendmsg*(3), and *socketpair*(3) are not implemented.
  - sock_init(3) is not required and not implemented.
  - *flock*(3) is not yet implemented (dummy function). (Perl has a workaround.)
  - *kill*(3): Special treatment of PID=0, PID=1 and PID=-1 is not implemented.
  - *waitpid*(3):

```
WUNTRACED
        Not implemented.
waitpid() is not implemented for negative values of PID.
```

  Note that `kill -9` does not work with the current version of EMX.
- See §117.13.1.

- Unix-domain sockets on OS/2 live in a pseudo-file-system `/sockets/...`. To avoid a failure to create a socket with a name of a different form, `"/socket/"` is prepended to the socket name (unless it starts with this already).

  This may lead to problems later in case the socket is accessed via the "usual" file-system calls using the "initial" name.

- Apparently, IBM used a compiler (for some period of time around '95?) which changes FP mask right and left. This is not *that* bad for IBM's programs, but the same compiler was used for DLLs which are used with general-purpose applications. When these DLLs are used, the state of floating-point flags in the application is not predictable.

  What is much worse, some DLLs change the floating point flags when in _DLLInitTerm() (e.g., *TCP32IP*). This means that even if you do not *call* any function in the DLL, just the act of loading this DLL will reset your flags. What is worse, the same compiler was used to compile some HOOK DLLs. Given that HOOK dlls are executed in the context of *all* the applications in the system, this means a complete unpredictablity of floating point flags on systems using such HOOK DLLs. E.g., *GAMESRVR.DLL* of **DIVE** origin changes the floating point flags on each write to the TTY of a VIO (windowed text-mode) applications.

  Some other (not completely debugged) situations when FP flags change include some video drivers (?), and some operations related to creation of the windows. People who code **OpenGL** may have more experience on this.

  Perl is generally used in the situation when all the floating-point exceptions are ignored, as is the default under EMX. If they are not ignored, some benign Perl programs would get a `SIGFPE` and would die a horrible death.

  To circumvent this, Perl uses two hacks. They help against *one* type of damage only: FP flags changed when loading a DLL.

  One of the hacks is to disable floating point exceptions on Perl startup (as is the default with EMX). This helps only with compile-time-linked DLLs changing the flags before main() had a chance to be called.

  The other hack is to restore FP flags after a call to dlopen(). This helps against similar damage done by DLLs _DLLInitTerm() at runtime. Currently no way to switch these hacks off is provided.

117.10.8 Modifications

Perl modifies some standard C library calls in the following ways:

`popen`

`my_popen` uses *sh.exe* if shell is required, cf. §117.12.4.

`tmpnam`

is created using `TMP` or `TEMP` environment variable, via `tempnam`.

`tmpfile`

If the current directory is not writable, file is created using modified `tmpnam`, so there may be a race condition.

`ctermid`

a dummy implementation.

`stat`

`os2_stat` special-cases */dev/tty* and */dev/con*.

`mkdir`, `rmdir`

these EMX functions do not work if the path contains a trailing `/`. Perl contains a workaround for this.

`flock`

Since *flock*(3) is present in EMX, but is not functional, it is emulated by perl. To disable the emulations, set environment variable `USE_PERL_FLOCK=0`.

117.10.9 Identifying DLLs

All the DLLs built with the current versions of Perl have ID strings identifying the name of the extension, its version, and the version of Perl required for this DLL. Run `bldlevel DLL-name` to find this info.

117.10.10 Centralized management of resources

Since to call certain OS/2 API one needs to have a correctly initialized `Win` subsystem, OS/2-specific extensions may require getting `HAB`s and `HMQ`s. If an extension would do it on its own, another extension could fail to initialize.

Perl provides a centralized management of these resources:

`HAB`

To get the HAB, the extension should call `hab = perl_hab_GET()` in C. After this call is performed, `hab` may be accessed as `Perl_hab`. There is no need to release the HAB after it is used.

If by some reasons *perl.h* cannot be included, use

```
extern int Perl_hab_GET(void);
```

instead.

`HMQ`

There are two cases:

- the extension needs an `HMQ` only because some API will not work otherwise. Use `serve = 0` below.
- the extension needs an `HMQ` since it wants to engage in a PM event loop. Use `serve = 1` below.

To get an `HMQ`, the extension should call `hmq = perl_hmq_GET(serve)` in C. After this call is performed, `hmq` may be accessed as `Perl_hmq`.

To signal to Perl that HMQ is not needed any more, call `perl_hmq_UNSET(serve)`. Perl process will automatically morph/unmorph itself into/from a PM process if HMQ is needed/not-needed. Perl will automatically enable/disable `WM_QUIT` message during shutdown if the message queue is served/not-served.

NOTE. If during a shutdown there is a message queue which did not disable WM_QUIT, and which did not process the received WM_QUIT message, the shutdown will be automatically cancelled. Do not call `perl_hmq_GET(1)` unless you are going to process messages on an orderly basis.

*** Treating errors reported by OS/2 API**

There are two principal conventions (it is useful to call them `Dos*` and `Win*` - though this part of the function signature is not always determined by the name of the API) of reporting the error conditions of OS/2 API. Most of `Dos*` APIs report the error code as the result of the call (so 0 means success, and there are many types of errors). Most of `Win*` API report success/fail via the result being `TRUE`/`FALSE`; to find the reason for the failure one should call WinGetLastError() API.

Some `Win*` entry points also overload a "meaningful" return value with the error indicator; having a 0 return value indicates an error. Yet some other `Win*` entry points overload things even more, and 0 return value may mean a successful call returning a valid value 0, as well as an error condition; in the case of a 0 return value one should call WinGetLastError() API to distinguish a successful call from a failing one.

By convention, all the calls to OS/2 API should indicate their failures by resetting $ˆE. All the Perl-accessible functions which call OS/2 API may be broken into two classes: some die()s when an API error is encountered, the other report the error via a false return value (of course, this does not concern Perl-accessible functions which *expect* a failure of the OS/2 API call, having some workarounds coded).

Obviously, in the situation of the last type of the signature of an OS/2 API, it is must more convenient for the users if the failure is indicated by die()ing: one does not need to check $ˆE to know that something went wrong. If, however, this solution is not desirable by some reason, the code in question should reset $ˆE to 0 before making this OS/2 API call, so that the caller of this Perl-accessible function has a chance to distinguish a success-but-0-return value from a failure. (One may return undef as an alternative way of reporting an error.)

The macros to simplify this type of error propagation are

`CheckOSError(expr)`
Returns true on error, sets $ˆE. Expects expr() be a call of `Dos*`-style API.

`CheckWinError(expr)`
Returns true on error, sets $ˆE. Expects expr() be a call of `Win*`-style API.

`SaveWinError(expr)`
Returns `expr`, sets $ˆE from WinGetLastError() if `expr` is false.

`SaveCroakWinError(expr,die,name1,name2)`
Returns `expr`, sets $ˆE from WinGetLastError() if `expr` is false, and die()s if `die` and $ˆE are true. The message to die is the concatenated strings `name1` and `name2`, separated by `": "` from the contents of $ˆE.

`WinError_2_Perl_rc`
Sets `Perl_rc` to the return value of WinGetLastError().

`FillWinError`
Sets `Perl_rc` to the return value of WinGetLastError(), and sets $ˆE to the corresponding value.

`FillOSError(rc)`
Sets `Perl_rc` to `rc`, and sets $ˆE to the corresponding value.

*** Loading DLLs and ordinals in DLLs**

Some DLLs are only present in some versions of OS/2, or in some configurations of OS/2. Some exported entry points are present only in DLLs shipped with some versions of OS/2. If these DLLs and entry points were linked directly for a Perl executable/DLL or from a Perl extensions, this binary would work only with the specified versions/setups. Even if these entry points were not needed, the *load* of the executable (or DLL) would fail.

For example, many newer useful APIs are not present in OS/2 v2; many PM-related APIs require DLLs not available on floppy-boot setup.

To make these calls fail *only when the calls are executed*, one should call these API via a dynamic linking API. There is a subsystem in Perl to simplify such type of calls. A large number of entry points available for such linking is provided (see `entries_ordinals` - and also `PMWIN_entries` - in *os2ish.h*). These ordinals can be accessed via the APIs:

```
CallORD(), DeclFuncByORD(), DeclVoidFuncByORD(),
DeclOSFuncByORD(), DeclWinFuncByORD(), AssignFuncPByORD(),
DeclWinFuncByORD_CACHE(), DeclWinFuncByORD_CACHE_survive(),
DeclWinFuncByORD_CACHE_resetError_survive(),
DeclWinFunc_CACHE(), DeclWinFunc_CACHE_resetError(),
DeclWinFunc_CACHE_survive(), DeclWinFunc_CACHE_resetError_survive()
```

See the header files and the C code in the supplied OS/2-related modules for the details on usage of these functions.

Some of these functions also combine dynaloading semantic with the error-propagation semantic discussed above.

117.11 Perl flavors

Because of idiosyncrasies of OS/2 one cannot have all the eggs in the same basket (though EMX environment tries hard to overcome this limitations, so the situation may somehow improve). There are 4 executables for Perl provided by the distribution:

117.11.1 *perl.exe*

The main workhorse. This is a chimera executable: it is compiled as an `a.out`-style executable, but is linked with `omf`-style dynamic library *perl.dll*, and with dynamic CRT DLL. This executable is a VIO application.

It can load perl dynamic extensions, and it can fork().

Note. Keep in mind that fork() is needed to open a pipe to yourself.

117.11.2 *perl_.exe*

This is a statically linked `a.out`-style executable. It cannot load dynamic Perl extensions. The executable supplied in binary distributions has a lot of extensions prebuilt, thus the above restriction is important only if you use custom-built extensions. This executable is a VIO application.

This is the only executable with does not require OS/2. The friends locked into `M$` world would appreciate the fact that this executable runs under DOS, Win0.3*, Win0.95 and WinNT with an appropriate extender. See §117.2.2.

117.11.3 *perl__.exe*

This is the same executable as *perl___.exe*, but it is a PM application.

Note. Usually (unless explicitly redirected during the startup) STDIN, STDERR, and STDOUT of a PM application are redirected to *nul*. However, it is possible to *see* them if you start `perl__.exe` from a PM program which emulates a console window, like *Shell mode* of Emacs or EPM. Thus it *is possible* to use Perl debugger (see *perldebug*) to debug your PM application (but beware of the message loop lockups - this will not work if you have a message queue to serve, unless you hook the serving into the getc() function of the debugger).

Another way to see the output of a PM program is to run it as

```
pm_prog args 2>&1 | cat -
```

with a shell *different* from *cmd.exe*, so that it does not create a link between a VIO session and the session of `pm_porg`. (Such a link closes the VIO window.) E.g., this works with *sh.exe* - or with Perl!

```
open P, 'pm_prog args 2>&1 |' or die;
print while <P>;
```

The flavor *perl__.exe* is required if you want to start your program without a VIO window present, but not `detached` (run `help detach` for more info). Very useful for extensions which use PM, like `Perl/Tk` or `OpenGL`.

Note also that the differences between PM and VIO executables are only in the *default* behaviour. One can start *any* executable in *any* kind of session by using the arguments `/fs`, `/pm` or `/win` switches of the command `start` (of *CMD.EXE* or a similar shell). Alternatively, one can use the numeric first argument of the `system` Perl function (see `OS2::Process`).

117.11.4 *perl___.exe*

This is an `omf`-style executable which is dynamically linked to *perl.dll* and CRT DLL. I know no advantages of this executable over `perl.exe`, but it cannot fork() at all. Well, one advantage is that the build process is not so convoluted as with `perl.exe`.

It is a VIO application.

117.11.5 Why strange names?

Since Perl processes the `#!`-line (cf. DESCRIPTION in *perlrun*, Switches in *perlrun*, Not a perl script in *perldiag*, No Perl script found in input in *perldiag*), it should know when a program *is a Perl*. There is some naming convention which allows Perl to distinguish correct lines from wrong ones. The above names are almost the only names allowed by this convention which do not contain digits (which have absolutely different semantics).

117.11.6 Why dynamic linking?

Well, having several executables dynamically linked to the same huge library has its advantages, but this would not substantiate the additional work to make it compile. The reason is the complicated-to-developers but very quick and convenient-to-users "hard" dynamic linking used by OS/2.

There are two distinctive features of the dyna-linking model of OS/2: first, all the references to external functions are resolved at the compile time; second, there is no runtime fixup of the DLLs after they are loaded into memory. The first feature is an enormous advantage over other models: it avoids conflicts when several DLLs used by an application export entries with the same name. In such cases "other" models of dyna-linking just choose between these two entry points using some random criterion - with predictable disasters as results. But it is the second feature which requires the build of *perl.dll*.

The address tables of DLLs are patched only once, when they are loaded. The addresses of the entry points into DLLs are guaranteed to be the same for all the programs which use the same DLL. This removes the runtime fixup - once DLL is loaded, its code is read-only.

While this allows some (significant?) performance advantages, this makes life much harder for developers, since the above scheme makes it impossible for a DLL to be "linked" to a symbol in the *.EXE* file. Indeed, this would need a DLL to have different relocations tables for the (different) executables which use this DLL.

However, a dynamically loaded Perl extension is forced to use some symbols from the perl executable, e.g., to know how to find the arguments to the functions: the arguments live on the perl internal evaluation stack. The solution is to put the main code of the interpreter into a DLL, and make the *.EXE* file which just loads this DLL into memory and supplies command-arguments. The extension DLL cannot link to symbols in *.EXE*, but it has no problem linking to symbols in the *.DLL*.

This *greatly* increases the load time for the application (as well as complexity of the compilation). Since interpreter is in a DLL, the C RTL is basically forced to reside in a DLL as well (otherwise extensions would not be able to use CRT). There are some advantages if you use different flavors of perl, such as running *perl.exe* and *perl__.exe* simultaneously: they share the memory of *perl.dll*.

NOTE. There is one additional effect which makes DLLs more wasteful: DLLs are loaded in the shared memory region, which is a scarse resource given the 512M barrier of the "standard" OS/2 virtual memory. The code of *.EXE* files is also shared by all the processes which use the particular *.EXE*, but they are "shared in the private address space of the process"; this is possible because the address at which different sections of the *.EXE* file are loaded is decided at compile-time, thus all the processes have these sections loaded at same addresses, and no fixup of internal links inside the *.EXE* is needed.

Since DLLs may be loaded at run time, to have the same mechanism for DLLs one needs to have the address range of *any of the loaded* DLLs in the system to be available *in all the processes* which did not load a particular DLL yet. This is why the DLLs are mapped to the shared memory region.

117.11.7 Why chimera build?

Current EMX environment does not allow DLLs compiled using Unixish `a.out` format to export symbols for data (or at least some types of data). This forces `omf`-style compile of *perl.dll*.

Current EMX environment does not allow *.EXE* files compiled in `omf` format to fork(). fork() is needed for exactly three Perl operations:

- explicit fork() in the script,
- `open FH, "|-"`
- `open FH, "-|"`, in other words, opening pipes to itself.

While these operations are not questions of life and death, they are needed for a lot of useful scripts. This forces `a.out`-style compile of *perl.exe*.

117.12 ENVIRONMENT

Here we list environment variables with are either OS/2- and DOS- and Win*-specific, or are more important under OS/2 than under other OSes.

117.12.1 PERLLIB_PREFIX

Specific for EMX port. Should have the form

```
path1;path2
```

or

```
path1 path2
```

If the beginning of some prebuilt path matches *path1*, it is substituted with *path2*.

Should be used if the perl library is moved from the default location in preference to `PERL(5)LIB`, since this would not leave wrong entries in @INC. For example, if the compiled version of perl looks for @INC in *f:/perllib/lib*, and you want to install the library in *h:/opt/gnu*, do

```
set PERLLIB_PREFIX=f:/perllib/lib;h:/opt/gnu
```

This will cause Perl with the prebuilt @INC of

```
f:/perllib/lib/5.00553/os2
f:/perllib/lib/5.00553
f:/perllib/lib/site_perl/5.00553/os2
f:/perllib/lib/site_perl/5.00553
.
```

to use the following @INC:

```
h:/opt/gnu/5.00553/os2
h:/opt/gnu/5.00553
h:/opt/gnu/site_perl/5.00553/os2
h:/opt/gnu/site_perl/5.00553
.
```

117.12.2 PERL_BADLANG

If 0, perl ignores setlocale() failing. May be useful with some strange *locale*s.

117.12.3 PERL_BADFREE

If 0, perl would not warn of in case of unwarranted free(). With older perls this might be useful in conjunction with the module DB_File, which was buggy when dynamically linked and OMF-built.

Should not be set with newer Perls, since this may hide some *real* problems.

117.12.4 PERL_SH_DIR

Specific for EMX port. Gives the directory part of the location for *sh.exe*.

117.12.5 `USE_PERL_FLOCK`

Specific for EMX port. Since *flock*(3) is present in EMX, but is not functional, it is emulated by perl. To disable the emulations, set environment variable `USE_PERL_FLOCK=0`.

117.12.6 `TMP` or `TEMP`

Specific for EMX port. Used as storage place for temporary files.

117.13 Evolution

Here we list major changes which could make you by surprise.

117.13.1 Text-mode filehandles

Starting from version 5.8, Perl uses a builtin translation layer for text-mode files. This replaces the efficient well-tested EMX layer by some code which should be best characterized as a "quick hack".

In addition to possible bugs and an inability to follow changes to the translation policy with off/on switches of TERMIO translation, this introduces a serious incompatible change: before sysread() on text-mode filehandles would go through the translation layer, now it would not.

117.13.2 Priorities

`setpriority` and `getpriority` are not compatible with earlier ports by Andreas Kaiser. See "`setpriority, getpriority`".

117.13.3 DLL name mangling: pre 5.6.2

With the release 5.003_01 the dynamically loadable libraries should be rebuilt when a different version of Perl is compiled. In particular, DLLs (including *perl.dll*) are now created with the names which contain a checksum, thus allowing workaround for OS/2 scheme of caching DLLs.

It may be possible to code a simple workaround which would

- find the old DLLs looking through the old @INC;
- mangle the names according to the scheme of new perl and copy the DLLs to these names;
- edit the internal `LX` tables of DLL to reflect the change of the name (probably not needed for Perl extension DLLs, since the internally coded names are not used for "specific" DLLs, they used only for "global" DLLs).
- edit the internal `IMPORT` tables and change the name of the "old" *perl????.dll* to the "new" *perl????.dll*.

117.13.4 DLL name mangling: 5.6.2 and beyond

In fact mangling of *extension* DLLs was done due to misunderstanding of the OS/2 dynaloading model. OS/2 (effectively) maintains two different tables of loaded DLL:

Global DLLs

those loaded by the base name from `LIBPATH`; including those associated at link time;

specific DLLs

loaded by the full name.

When resolving a request for a global DLL, the table of already-loaded specific DLLs is (effectively) ignored; moreover, specific DLLs are *always* loaded from the prescribed path.

There is/was a minor twist which makes this scheme fragile: what to do with DLLs loaded from

`BEGINLIBPATH` and `ENDLIBPATH`

(which depend on the process)

. from `LIBPATH`

which *effectively* depends on the process (although `LIBPATH` is the same for all the processes).

Unless `LIBPATHSTRICT` is set to `T` (and the kernel is after 2000/09/01), such DLLs are considered to be global. When loading a global DLL it is first looked in the table of already-loaded global DLLs. Because of this the fact that one executable loaded a DLL from `BEGINLIBPATH` and `ENDLIBPATH`, or . from `LIBPATH` may affect *which* DLL is loaded when *another* executable requests a DLL with the same name. *This* is the reason for version-specific mangling of the DLL name for perl DLL.

Since the Perl extension DLLs are always loaded with the full path, there is no need to mangle their names in a version-specific ways: their directory already reflects the corresponding version of perl, and @INC takes into account binary compatibility with older version. Starting from `5.6.2` the name mangling scheme is fixed to be the same as for Perl 5.005_53 (same as in a popular binary release). Thus new Perls will be able to *resolve the names* of old extension DLLs if @INC allows finding their directories.

However, this still does not guarantee that these DLL may be loaded. The reason is the mangling of the name of the *Perl DLL*. And since the extension DLLs link with the Perl DLL, extension DLLs for older versions would load an older Perl DLL, and would most probably segfault (since the data in this DLL is not properly initialized).

There is a partial workaround (which can be made complete with newer OS/2 kernels): create a forwarder DLL with the same name as the DLL of the older version of Perl, which forwards the entry points to the newer Perl's DLL. Make this DLL accessible on (say) the `BEGINLIBPATH` of the new Perl executable. When the new executable accesses old Perl's extension DLLs, they would request the old Perl's DLL by name, get the forwarder instead, so effectively will link with the currently running (new) Perl DLL.

This may break in two ways:

- Old perl executable is started when a new executable is running has loaded an extension compiled for the old executable (ouph!). In this case the old executable will get a forwarder DLL instead of the old perl DLL, so would link with the new perl DLL. While not directly fatal, it will behave the same as new executable. This beats the whole purpose of explicitly starting an old executable.
- A new executable loads an extension compiled for the old executable when an old perl executable is running. In this case the extension will not pick up the forwarder - with fatal results.

With support for `LIBPATHSTRICT` this may be circumvented - unless one of DLLs is started from . from `LIBPATH` (I do not know whether `LIBPATHSTRICT` affects this case).

REMARK. Unless newer kernels allow . in `BEGINLIBPATH` (older do not), this mess cannot be completely cleaned. (It turns out that as of the beginning of 2002, . is not allowed, but .\. is - and it has the same effect.)

REMARK. `LIBPATHSTRICT`, `BEGINLIBPATH` and `ENDLIBPATH` are not environment variables, although *cmd.exe* emulates them on `SET ...` lines. From Perl they may be accessed by *Cwd::extLibpath* and Cwd::extLibpath_set.

117.13.5 DLL forwarder generation

Assume that the old DLL is named *perlE0AC.dll* (as is one for 5.005_53), and the new version is 5.6.1. Create a file *perl5shim.def-leader* with

```
LIBRARY 'perlE0AC' INITINSTANCE TERMINSTANCE
DESCRIPTION '@#perl5-porters@perl.org:5.006001#@ Perl module for 5.00553 -> Perl 5.6.1 forwarder'
CODE LOADONCALL
DATA LOADONCALL NONSHARED MULTIPLE
EXPORTS
```

modifying the versions/names as needed. Run

```
perl -wnle "next if 0../EXPORTS/; print qq(  \"$1\") if /\"(\w+)\"/" perl5.def >lst
```

in the Perl build directory (to make the DLL smaller replace perl5.def with the definition file for the older version of Perl if present).

```
cat perl5shim.def-leader lst >perl5shim.def
gcc -Zomf -Zdll -o perlE0AC.dll perl5shim.def -s -llibperl
```

(ignore multiple `warning L4085`).

117.13.6 Threading

As of release 5.003_01 perl is linked to multithreaded C RTL DLL. If perl itself is not compiled multithread-enabled, so will not be perl's malloc(). However, extensions may use multiple thread on their own risk.

This was needed to compile `Perl/Tk` for XFree86-OS/2 out-of-the-box, and link with DLLs for other useful libraries, which typically are compiled with `-Zmt -Zcrtdll`.

117.13.7 Calls to external programs

Due to a popular demand the perl external program calling has been changed wrt Andreas Kaiser's port. *If* perl needs to call an external program *via shell*, the *f:/bin/sh.exe* will be called, or whatever is the override, see §117.12.4.

Thus means that you need to get some copy of a *sh.exe* as well (I use one from pdksh). The path *F:/bin* above is set up automatically during the build to a correct value on the builder machine, but is overridable at runtime,

Reasons: a consensus on `perl5-porters` was that perl should use one non-overridable shell per platform. The obvious choices for OS/2 are *cmd.exe* and *sh.exe*. Having perl build itself would be impossible with *cmd.exe* as a shell, thus I picked up `sh.exe`. This assures almost 100% compatibility with the scripts coming from *nix. As an added benefit this works as well under DOS if you use DOS-enabled port of pdksh (see §117.6.2).

Disadvantages: currently *sh.exe* of pdksh calls external programs via fork()/exec(), and there is *no* functioning exec() on OS/2. exec() is emulated by EMX by an asynchronous call while the caller waits for child completion (to pretend that the `pid` did not change). This means that 1 *extra* copy of *sh.exe* is made active via fork()/exec(), which may lead to some resources taken from the system (even if we do not count extra work needed for fork()ing).

Note that this a lesser issue now when we do not spawn *sh.exe* unless needed (metachars found).

One can always start *cmd.exe* explicitly via

```
  system 'cmd', '/c', 'mycmd', 'arg1', 'arg2', ...
```

If you need to use *cmd.exe*, and do not want to hand-edit thousands of your scripts, the long-term solution proposed on p5-p is to have a directive

```
  use OS2::Cmd;
```

which will override system(), exec(), ", and `open(,'...|')`. With current perl you may override only system(), readpipe() - the explicit version of ", and maybe exec(). The code will substitute the one-argument call to system() by `CORE::system('cmd.exe', '/c', shift)`.

If you have some working code for `OS2::Cmd`, please send it to me, I will include it into distribution. I have no need for such a module, so cannot test it.

For the details of the current situation with calling external programs, see Starting OS/2 (and DOS) programs under Perl. Set us mention a couple of features:

- External scripts may be called by their basename. Perl will try the same extensions as when processing **-S** command-line switch.
- External scripts starting with `#!` or `extproc` will be executed directly, without calling the shell, by calling the program specified on the rest of the first line.

117.13.8 Memory allocation

Perl uses its own malloc() under OS/2 - interpreters are usually malloc-bound for speed, but perl is not, since its malloc is lightning-fast. Perl-memory-usage-tuned benchmarks show that Perl's malloc is 5 times quicker than EMX one. I do not have convincing data about memory footprint, but a (pretty random) benchmark showed that Perl's one is 5% better.

Combination of perl's malloc() and rigid DLL name resolution creates a special problem with library functions which expect their return value to be free()d by system's free(). To facilitate extensions which need to call such functions, system memory-allocation functions are still available with the prefix `emx_` added. (Currently only DLL perl has this, it should propagate to *perl_.exe* shortly.)

117.13.9 Threads

One can build perl with thread support enabled by providing `-D usethreads` option to *Configure*. Currently OS/2 support of threads is very preliminary.

Most notable problems:

`COND_WAIT`

may have a race condition (but probably does not due to edge-triggered nature of OS/2 Event semaphores). (Needs a reimplementation (in terms of chaining waiting threads, with the linked list stored in per-thread structure?)?)

os2.c

has a couple of static variables used in OS/2-specific functions. (Need to be moved to per-thread structure, or serialized?)

Note that these problems should not discourage experimenting, since they have a low probability of affecting small programs.

117.14 BUGS

This description is not updated often (since 5.6.1?), see *./os2/Changes* (*perlos2delta*) for more info.

117.15 AUTHOR

Ilya Zakharevich, cpan@ilyaz.org

117.16 SEE ALSO

perl(1).

Chapter 118

README.os390

Building and installing Perl for OS/390 and z/OS

118.1 SYNOPSIS

This document will help you Configure, build, test and install Perl on OS/390 (aka z/OS) Unix System Services.

118.2 DESCRIPTION

This is a fully ported Perl for OS/390 Version 2 Release 3, 5, 6, 7, 8, and 9. It may work on other versions or releases, but those are the ones we've tested it on.
You may need to carry out some system configuration tasks before running the Configure script for Perl.

118.2.1 Tools

The z/OS Unix Tools and Toys list may prove helpful and contains links to ports of much of the software helpful for building Perl. http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1toy.html

118.2.2 Unpacking Perl distribution on OS/390

If using ftp remember to transfer the distribution in binary format.
Gunzip/gzip for OS/390 is discussed at:

```
  http://www-1.ibm.com/servers/eserver/zseries/zos/unix/faq/bpxqp1.html
```

to extract an ASCII tar archive on OS/390, try this:

```
   pax -o to=IBM-1047,from=ISO8859-1 -r < latest.tar
```

or

```
   zcat latest.tar.Z | pax -o to=IBM-1047,from=ISO8859-1 -r
```

If you get lots of errors of the form

```
  tar: FSUM7171 ...: cannot set uid/gid: EDC5139I Operation not permitted.
```

you didn't read the above and tried to use tar instead of pax, you'll first have to remove the (now corrupt) perl directory

```
   rm -rf perl-...
```

and then use pax.

118.2.3 Setup and utilities for Perl on OS/390

Be sure that your yacc installation is in place including any necessary parser template files. If you have not already done so then be sure to:

```
  cp /samples/yyparse.c /etc
```

This may also be a good time to ensure that your /etc/protocol file and either your /etc/resolv.conf or /etc/hosts files are in place. The IBM document that described such USS system setup issues was SC28-1890-07 "OS/390 UNIX System Services Planning", in particular Chapter 6 on customizing the OE shell.

GNU make for OS/390, which is recommended for the build of perl (as well as building CPAN modules and extensions), is available from the Tools.

Some people have reported encountering "Out of memory!" errors while trying to build Perl using GNU make binaries. If you encounter such trouble then try to download the source code kit and build GNU make from source to eliminate any such trouble. You might also find GNU make (as well as Perl and Apache) in the red-piece/book "Open Source Software for OS/390 UNIX", SG24-5944-00 from IBM.

If instead of the recommended GNU make you would like to use the system supplied make program then be sure to install the default rules file properly via the shell command:

```
    cp /samples/startup.mk /etc
```

and be sure to also set the environment variable _C89_CCMODE=1 (exporting _C89_CCMODE=1 is also a good idea for users of GNU make).

You might also want to have GNU groff for OS/390 installed before running the ‘make install‘ step for Perl.

There is a syntax error in the /usr/include/sys/socket.h header file that IBM supplies with USS V2R7, V2R8, and possibly V2R9. The problem with the header file is that near the definition of the SO_REUSEPORT constant there is a spurious extra ’/’ character outside of a comment like so:

```
 #define SO_REUSEPORT    0x0200    /* allow local address & port
                                      reuse */                    /
```

You could edit that header yourself to remove that last ’/’, or you might note that Language Environment (LE) APAR PQ39997 describes the problem and PTF’s UQ46272 and UQ46271 are the (R8 at least) fixes and apply them. If left unattended that syntax error will turn up as an inability for Perl to build its "Socket" extension.

For successful testing you may need to turn on the sticky bit for your world readable /tmp directory if you have not already done so (see man chmod).

118.2.4 Configure Perl on OS/390

Once you’ve unpacked the distribution, run "sh Configure" (see INSTALL for a full discussion of the Configure options). There is a "hints" file for os390 that specifies the correct values for most things. Some things to watch out for include:

- A message of the form:

  ```
   (I see you are using the Korn shell.  Some ksh’s blow up on Configure,
   mainly on older exotic systems.  If yours does, try the Bourne shell instead.)
  ```

  is nothing to worry about at all.

- Some of the parser default template files in /samples are needed in /etc. In particular be sure that you at least copy /samples/yyparse.c to /etc before running Perl’s Configure. This step ensures successful extraction of EBCDIC versions of parser files such as perly.c, perly.h, and x2p/a2p.c. This has to be done before running Configure the first time. If you failed to do so then the easiest way to re-Configure Perl is to delete your misconfigured build root and re-extract the source from the tar ball. Then you must ensure that /etc/yyparse.c is properly in place before attempting to re-run Configure.

- This port will support dynamic loading, but it is not selected by default. If you would like to experiment with dynamic loading then be sure to specify -Dusedl in the arguments to the Configure script. See the comments in hints/os390.sh for more information on dynamic loading. If you build with dynamic loading then you will need to add the $archlibexp/CORE directory to your LIBPATH environment variable in order for perl to work. See the config.sh file for the value of $archlibexp. If in trying to use Perl you see an error message similar to:

  ```
  CEE3501S The module libperl.dll was not found.
          From entry point __dllstaticinit at compile unit offset +00000194 at
  ```

  then your LIBPATH does not have the location of libperl.x and either libperl.dll or libperl.so in it. Add that directory to your LIBPATH and proceed.
- Do not turn on the compiler optimization flag "-O". There is a bug in either the optimizer or perl that causes perl to not work correctly when the optimizer is on.
- Some of the configuration files in /etc used by the networking APIs are either missing or have the wrong names. In particular, make sure that there's either an /etc/resolv.conf or an /etc/hosts, so that gethostbyname() works, and make sure that the file /etc/proto has been renamed to /etc/protocol (NOT /etc/protocols, as used by other Unix systems). You may have to look for things like HOSTNAME and DOMAINORIGIN in the "//'SYS1.TCPPARMS(TCPDATA)'" PDS member in order to properly set up your /etc networking files.

118.2.5 Build, Test, Install Perl on OS/390

Simply put:

```
    sh Configure
    make
    make test
```

if everything looks ok (see the next section for test/IVP diagnosis) then:

```
    make install
```

this last step may or may not require UID=0 privileges depending on how you answered the questions that Configure asked and whether or not you have write access to the directories you specified.

118.2.6 Build Anomalies with Perl on OS/390

"Out of memory!" messages during the build of Perl are most often fixed by re building the GNU make utility for OS/390 from a source code kit.

Another memory limiting item to check is your MAXASSIZE parameter in your 'SYS1.PARMLIB(BPXPRMxx)' data set (note too that as of V2R8 address space limits can be set on a per user ID basis in the USS segment of a RACF profile). People have reported successful builds of Perl with MAXASSIZE parameters as small as 503316480 (and it may be possible to build Perl with a MAXASSIZE smaller than that).

Within USS your /etc/profile or $HOME/.profile may limit your ulimit settings. Check that the following command returns reasonable values:

```
    ulimit -a
```

To conserve memory you should have your compiler modules loaded into the Link Pack Area (LPA/ELPA) rather than in a link list or step lib.

If the c89 compiler complains of syntax errors during the build of the Socket extension then be sure to fix the syntax error in the system header /usr/include/sys/socket.h.

118.2.7 Testing Anomalies with Perl on OS/390

The ‘make test‘ step runs a Perl Verification Procedure, usually before installation. You might encounter STDERR messages even during a successful run of ‘make test‘. Here is a guide to some of the more commonly seen anomalies:

- A message of the form:

  ```
  comp/cpp.............ERROR CBC3191 ./.301989890.c:1     The character $ is not a
   valid C source character.
  FSUM3065 The COMPILE step ended with return code 12.
  FSUM3017 Could not compile .301989890.c. Correct the errors and try again.
  ok
  ```

  indicates that the t/comp/cpp.t test of Perl’s -P command line switch has passed but that the particular invocation of c89 -E in the cpp script does not suppress the C compiler check of source code validity.

- A message of the form:

  ```
  io/openpid...........CEE5210S The signal SIGHUP was received.
  CEE5210S The signal SIGHUP was received.
  CEE5210S The signal SIGHUP was received.
  ok
  ```

  indicates that the t/io/openpid.t test of Perl has passed but done so with extraneous messages on stderr from CEE.

- A message of the form:

  ```
  lib/ftmp-security....File::Temp::_gettemp: Parent directory (/tmp/) is not safe
  (sticky bit not set when world writable?) at lib/ftmp-security.t line 100
  File::Temp::_gettemp: Parent directory (/tmp/) is not safe (sticky bit not
  set when world writable?) at lib/ftmp-security.t line 100
  ok
  ```

  indicates a problem with the permissions on your /tmp directory within the HFS. To correct that problem issue the command:

  ```
      chmod a+t /tmp
  ```

  from an account with write access to the directory entry for /tmp.

- Out of Memory!

  Recent perl test suite is quite memory hunrgy. In addition to the comments above on memory limitations it is also worth checking for _CEE_RUNOPTS in your environment. Perl now has (in miniperlmain.c) a C #pragma to set CEE run options, but the environment variable wins.

  The C code asks for:

  ```
  #pragma runopts(HEAP(2M,500K,ANYWHERE,KEEP,8K,4K) STACK(,,ANY,) ALL31(ON))
  ```

  The important parts of that are the second argument (the increment) to HEAP, and allowing the stack to be "Above the (16M) line". If the heap increment is too small then when perl (for example loading unicode/Name.pl) tries to create a "big" (400K+) string it cannot fit in a single segment and you get "Out of Memory!" - even if there is still plenty of memory available.

  A related issue is use with perl’s malloc. Perl’s malloc uses `sbrk()` to get memory, and `sbrk()` is limited to the first allocation so in this case something like:

  ```
   HEAP(8M,500K,ANYWHERE,KEEP,8K,4K)
  ```

  is needed to get through the test suite.

118.2.8 Installation Anomalies with Perl on OS/390

The installman script will try to run on OS/390. There will be fewer errors if you have a roff utility installed. You can obtain GNU groff from the Redbook SG24-5944-00 ftp site.

118.2.9 Usage Hints for Perl on OS/390

When using perl on OS/390 please keep in mind that the EBCDIC and ASCII character sets are different. See perlebcdic.pod for more on such character set issues. Perl builtin functions that may behave differently under EBCDIC are also mentioned in the perlport.pod document.

Open Edition (UNIX System Services) from V2R8 onward does support #!/path/to/perl script invocation. There is a PTF available from IBM for V2R7 that will allow shell/kernel support for #!. USS releases prior to V2R7 did not support the #! means of script invocation. If you are running V2R6 or earlier then see:

```
    head `whence perldoc`
```

for an example of how to use the "eval exec" trick to ask the shell to have Perl run your scripts on those older releases of Unix System Services.

If you are having trouble with square brackets then consider switching your rlogin or telnet client. Try to avoid older 3270 emulators and ISHELL for working with Perl on USS.

118.2.10 Floating Point Anomalies with Perl on OS/390

There appears to be a bug in the floating point implementation on S/390 systems such that calling int() on the product of a number and a small magnitude number is not the same as calling int() on the quotient of that number and a large magnitude number. For example, in the following Perl code:

```
    my $x = 100000.0;
    my $y = int($x * 1e-5) * 1e5; # '0'
    my $z = int($x / 1e+5) * 1e5;  # '100000'
    print "\$y is $y and \$z is $z\n"; # $y is 0 and $z is 100000
```

Although one would expect the quantities $y and $z to be the same and equal to 100000 they will differ and instead will be 0 and 100000 respectively.

The problem can be further examined in a roughly equivalent C program:

```
    #include <stdio.h>
    #include <math.h>
    main()
    {
    double r1,r2;
    double x = 100000.0;
    double y = 0.0;
    double z = 0.0;
    x = 100000.0 * 1e-5;
    r1 = modf (x,&y);
    x = 100000.0 / 1e+5;
    r2 = modf (x,&z);
    printf("y is %e and z is %e\n",y*1e5,z*1e5);
    /* y is 0.000000e+00 and z is 1.000000e+05 (with c89) */
    }
```

118.2.11 Modules and Extensions for Perl on OS/390

Pure pure (that is non xs) modules may be installed via the usual:

```
perl Makefile.PL
make
make test
make install
```

If you built perl with dynamic loading capability then that would also be the way to build xs based extensions. However, if you built perl with the default static linking you can still build xs based extensions for OS/390 but you will need to follow the instructions in ExtUtils::MakeMaker for building statically linked perl binaries. In the simplest configurations building a static perl + xs extension boils down to:

```
perl Makefile.PL
make
make perl
make test
make install
make -f Makefile.aperl inst_perl MAP_TARGET=perl
```

In most cases people have reported better results with GNU make rather than the system's /bin/make program, whether for plain modules or for xs based extensions.

If the make process encounters trouble with either compilation or linking then try setting the _C89_CCMODE to 1. Assuming sh is your login shell then run:

```
export _C89_CCMODE=1
```

If tcsh is your login shell then use the setenv command.

118.3 AUTHORS

David Fiander and Peter Prymmer with thanks to Dennis Longnecker and William Raffloer for valuable reports, LPAR and PTF feedback. Thanks to Mike MacIsaac and Egon Terwedow for SG24-5944-00. Thanks to Ignasi Roca for pointing out the floating point problems. Thanks to John Goodyear for dynamic loading help.

118.4 SEE ALSO

INSTALL, *perlport*, *perlebcdic*, *ExtUtils::MakeMaker*.

```
http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1toy.html

http://www.redbooks.ibm.com/abstracts/sg245944.html

http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1ty1.html#opensrc

http://www.xray.mpe.mpg.de/mailing-lists/perl-mvs/

http://publibz.boulder.ibm.com:80/cgi-bin/bookmgr_OS390/BOOKS/ceea3030/

http://publibz.boulder.ibm.com:80/cgi-bin/bookmgr_OS390/BOOKS/CBCUG030/
```

118.4.1 Mailing list for Perl on OS/390

If you are interested in the VM/ESA, z/OS (formerly known as OS/390) and POSIX-BC (BS2000) ports of Perl then see the perl-mvs mailing list. To subscribe, send an empty message to perl-mvs-subscribe@perl.org.

See also:

```
    http://lists.perl.org/showlist.cgi?name=perl-mvs
```

There are web archives of the mailing list at:

```
    http://www.xray.mpe.mpg.de/mailing-lists/perl-mvs/
    http://archive.develooper.com/perl-mvs@perl.org/
```

118.5 HISTORY

This document was originally written by David Fiander for the 5.005 release of Perl.

This document was podified for the 5.005_03 release of Perl 11 March 1999.

Updated 28 November 2001 for broken URLs.

Updated 12 November 2000 for the 5.7.1 release of Perl.

Updated 15 January 2001 for the 5.7.1 release of Perl.

Updated 24 January 2001 to mention dynamic loading.

Updated 12 March 2001 to mention //'SYS1.TCPPARMS(TCPDATA)'.

Chapter 119

README.os400

Perl version 5 on OS/400

119.1 DESCRIPTION

This document describes various features of IBM's OS/400 operating system that will affect how Perl version 5 (hereafter just Perl) is compiled and/or runs.

By far the easiest way to build Perl for OS/400 is to use the PASE (Portable Application Solutions Environment), for more information see http://www.iseries.ibm.com/developer/factory/pase/index.html This environment allows one to use AIX APIs while programming, and it provides a runtime that allows AIX binaries to execute directly on the PowerPC iSeries.

119.1.1 Compiling Perl for OS/400 PASE

The recommended way to build Perl for the OS/400 PASE is to build the Perl 5 source code (release 5.8.1 or later) under AIX.

The trick is to give a special parameter to the Configure shell script when running it on AIX:

```
  sh Configure -DPASE ...
```

The default installation directory of Perl under PASE is /QOpenSys/perl. This can be modified if needed with Configure parameter -Dprefix=/some/dir.

Starting from OS/400 V5R2 the IBM Visual Age compiler is supported on OS/400 PASE, so it is possible to build Perl natively on OS/400. The easier way, however, is to compile in AIX, as just described.

If you don't want to install the compiled Perl in AIX into /QOpenSys (for packaging it before copying it to PASE), you can use a Configure parameter: -Dinstallprefix=/tmp/QOpenSys/perl. This will cause the "make install" to install everything into that directory, while the installed files still think they are (will be) in /QOpenSys/perl.

If building natively on PASE, please do the build under the /QOpenSys directory, since Perl is happier when built on a case sensitive filesystem.

119.1.2 Installing Perl in OS/400 PASE

If you are compiling on AIX, simply do a "make install" on the AIX box. Once the install finishes, tar up the /QOpenSys/perl directory. Transfer the tarball to the OS/400 using FTP with the following commands:

```
  > binary
  > site namefmt 1
  > put perl.tar /QOpenSys
```

Once you have it on, simply bring up a PASE shell and extract the tarball.

If you are compiling in PASE, then "make install" is the only thing you will need to do.

The default path for perl binary is /QOpenSys/perl/bin/perl. You'll want to symlink /QOpenSys/usr/bin/perl to this file so you don't have to modify your path.

119.1.3 Using Perl in OS/400 PASE

Perl in PASE may be used in the same manner as you would use Perl on AIX.

Scripts starting with #!/usr/bin/perl should work if you have /QOpenSys/usr/bin/perl symlinked to your perl binary. This will not work if you've done a setuid/setgid or have environment variable PASE_EXEC_QOPENSYS="N". If you have V5R1, you'll need to get the latest PTFs to have this feature. Scripts starting with #!/QOpenSys/perl/bin/perl should always work.

119.1.4 Known Problems

When compiling in PASE, there is no "oslevel" command. Therefore, you may want to create a script called "oslevel" that echoes the level of AIX that your version of PASE runtime supports. If you're unsure, consult your documentation or use "4.3.3.0".

If you have test cases that fail, check for the existence of spool files. The test case may be trying to use a syscall that is not implemented in PASE. To avoid the SIGILL, try setting the PASE_SYSCALL_NOSIGILL environment variable or have a handler for the SIGILL. If you can compile programs for PASE, run the config script and edit config.sh when it gives you the option. If you want to remove fchdir(), which isn't implement in V5R1, simply change the line that says:

d_fchdir='define'

to

d_fchdir='undef'

and then compile Perl. The places where fchdir() is used have alternatives for systems that do not have fchdir() available.

119.1.5 Perl on ILE

There exists a port of Perl to the ILE environment. This port, however, is based quite an old release of Perl, Perl 5.00502 (August 1998). (As of July 2002 the latest release of Perl is 5.8.0, and even 5.6.1 has been out since April 2001.) If you need to run Perl on ILE, though, you may need this older port: http://www.cpan.org/ports/#os400 Note that any Perl release later than 5.00502 has not been ported to ILE.

If you need to use Perl in the ILE environment, you may want to consider using Qp2RunPase() to call the PASE version of Perl.

119.2 AUTHORS

Jarkko Hietaniemi <jhi@iki.fi> Bryan Logan <bryanlog@us.ibm.com> David Larson <larson1@us.ibm.com>

Chapter 120

perlplan9

Plan 9-specific documentation for Perl

120.1 DESCRIPTION

These are a few notes describing features peculiar to Plan 9 Perl. As such, it is not intended to be a replacement for the rest of the Perl 5 documentation (which is both copious and excellent). If you have any questions to which you can't find answers in these man pages, contact Luther Huffman at lutherh@stratcom.com and we'll try to answer them.

120.1.1 Invoking Perl

Perl is invoked from the command line as described in *perl*. Most perl scripts, however, do have a first line such as "#!/usr/local/bin/perl". This is known as a shebang (shell-bang) statement and tells the OS shell where to find the perl interpreter. In Plan 9 Perl this statement should be "#!/bin/perl" if you wish to be able to directly invoke the script by its name. Alternatively, you may invoke perl with the command "Perl" instead of "perl". This will produce Acme-friendly error messages of the form "filename:18".
Some scripts, usually identified with a *.PL extension, are self-configuring and are able to correctly create their own shebang path from config information located in Plan 9 Perl. These you won't need to be worried about.

120.1.2 What's in Plan 9 Perl

Although Plan 9 Perl currently only provides static loading, it is built with a number of useful extensions. These include Opcode, FileHandle, Fcntl, and POSIX. Expect to see others (and DynaLoading!) in the future.

120.1.3 What's not in Plan 9 Perl

As mentioned previously, dynamic loading isn't currently available nor is MakeMaker. Both are high-priority items.

120.1.4 Perl5 Functions not currently supported in Plan 9 Perl

Some, such as `chown` and `umask` aren't provided because the concept does not exist within Plan 9. Others, such as some of the socket-related functions, simply haven't been written yet. Many in the latter category may be supported in the future.
The functions not currently implemented include:

```
    chown, chroot, dbmclose, dbmopen, getsockopt,
    setsockopt, recvmsg, sendmsg, getnetbyname,
    getnetbyaddr, getnetent, getprotoent, getservent,
    sethostent, setnetent, setprotoent, setservent,
    endservent, endnetent, endprotoent, umask
```

There may be several other functions that have undefined behavior so this list shouldn't be considered complete.

120.1.5 Signals in Plan 9 Perl

For compatibility with perl scripts written for the Unix environment, Plan 9 Perl uses the POSIX signal emulation provided in Plan 9's ANSI POSIX Environment (APE). Signal stacking isn't supported. The signals provided are:

```
    SIGHUP, SIGINT, SIGQUIT, SIGILL, SIGABRT,
    SIGFPE, SIGKILL, SIGSEGV, SIGPIPE, SIGPIPE, SIGALRM,
    SIGTERM, SIGUSR1, SIGUSR2, SIGCHLD, SIGCONT,
    SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU
```

120.2 COMPILING AND INSTALLING PERL ON PLAN 9

WELCOME to Plan 9 Perl, brave soul!

```
   This is a preliminary alpha version of Plan 9 Perl. Still to be
implemented are MakeMaker and DynaLoader. Many perl commands are
missing or currently behave in an inscrutable manner. These gaps will,
with perseverance and a modicum of luck, be remedied in the near
future.To install this software:
```

1. Create the source directories and libraries for perl by running the plan9/setup.rc command (i.e., located in the plan9 subdirectory). Note: the setup routine assumes that you haven't dearchived these files into /sys/src/cmd/perl. After running setup.rc you may delete the copy of the source you originally detarred, as source code has now been installed in /sys/src/cmd/perl. If you plan on installing perl binaries for all architectures, run "setup.rc -a".
2. After making sure that you have adequate privileges to build system software, from /sys/src/cmd/perl/5.00301 (adjust version appropriately) run:

```
        mk install
```

If you wish to install perl versions for all architectures (68020, mips, sparc and 386) run:

```
        mk installall
```

3. Wait. The build process will take a *long* time because perl bootstraps itself. A 75MHz Pentium, 16MB RAM machine takes roughly 30 minutes to build the distribution from scratch.

120.2.1 Installing Perl Documentation on Plan 9

This perl distribution comes with a tremendous amount of documentation. To add these to the built-in manuals that come with Plan 9, from /sys/src/cmd/perl/5.00301 (adjust version appropriately) run:

```
        mk man
```

To begin your reading, start with:

```
        man perl
```

This is a good introduction and will direct you towards other man pages that may interest you.

(Note: "mk man" may produce some extraneous noise. Fear not.)

120.3 BUGS

"As many as there are grains of sand on all the beaches of the world . . ." - Carl Sagan

120.4 Revision date

This document was revised 09-October-1996 for Perl 5.003_7.

120.5 AUTHOR

Direct questions, comments, and the unlikely bug report (ahem) direct comments toward:
Luther Huffman, lutherh@stratcom.com, Strategic Computer Solutions, Inc.

Chapter 121

README.qnx

Perl version 5 on QNX

121.1 DESCRIPTION

As of perl5.7.2 all tests pass under:

```
  QNX 4.24G
  Watcom 10.6 with Beta/970211.wcc.update.tar.F
  socket3r.lib Nov21 1996.
```

As of perl5.8.1 there is at least one test still failing.
Some tests may complain under known circumstances.
See below and hints/qnx.sh for more information.
Under QNX 6.2.0 there are still a few tests which fail. See below and hints/qnx.sh for more information.

121.1.1 Required Software for Compiling Perl on QNX4

As with many unix ports, this one depends on a few "standard" unix utilities which are not necessarily standard for QNX4.

/bin/sh
: This is used heavily by Configure and then by perl itself. QNX4's version is fine, but Configure will choke on the 16-bit version, so if you are running QNX 4.22, link /bin/sh to /bin32/ksh

ar
: This is the standard unix library builder. We use wlib. With Watcom 10.6, when wlib is linked as "ar", it behaves like ar and all is fine. Under 9.5, a cover is required. One is included in ../qnx

nm
: This is used (optionally) by configure to list the contents of libraries. I will generate a cover function on the fly in the UU directory.

cpp
: Configure and perl need a way to invoke a C preprocessor. I have created a simple cover for cc which does the right thing. Without this, Configure will create its own wrapper which works, but it doesn't handle some of the command line arguments that perl will throw at it.

make
: You really need GNU make to compile this. GNU make ships by default with QNX 4.23, but you can get it from quics for earlier versions.

121.1.2 Outstanding Issues with Perl on QNX4

There is no support for dynamically linked libraries in QNX4.

If you wish to compile with the Socket extension, you need to have the TCP/IP toolkit, and you need to make sure that -lsocket locates the correct copy of socket3r.lib. Beware that the Watcom compiler ships with a stub version of socket3r.lib which has very little functionality. Also beware the order in which wlink searches directories for libraries. You may have /usr/lib/socket3r.lib pointing to the correct library, but wlink may pick up /usr/watcom/10.6/usr/lib/socket3r.lib instead. Make sure they both point to the correct library, that is, /usr/tcptk/current/usr/lib/socket3r.lib.

The following tests may report errors under QNX4:

ext/Cwd/Cwd.t will complain if ‘pwd‘ and cwd don’t give the same results. cwd calls ‘fullpath -t‘, so if you cd ‘fullpath -t‘ before running the test, it will pass.

lib/File/Find/taint.t will complain if ’.’ is in your PATH. The PATH test is triggered because cwd calls ‘fullpath -t‘.

ext/IO/lib/IO/t/io_sock.t: Subtests 14 and 22 are skipped due to the fact that the functionality to read back the non-blocking status of a socket is not implemented in QNX’s TCP/IP. This has been reported to QNX and it may work with later versions of TCP/IP.

t/io/tell.t: Subtest 27 is failing. We are still investigating.

121.1.3 QNX auxiliary files

The files in the "qnx" directory are:

qnx/ar

A script that emulates the standard unix archive (aka library) utility. Under Watcom 10.6, ar is linked to wlib and provides the expected interface. With Watcom 9.5, a cover function is required. This one is fairly crude but has proved adequate for compiling perl.

qnx/cpp

A script that provides C preprocessing functionality. Configure can generate a similar cover, but it doesn’t handle all the command-line options that perl throws at it. This might be reasonably placed in /usr/local/bin.

121.1.4 Outstanding issues with perl under QNX6

The following tests are still failing for Perl 5.8.1 under QNX 6.2.0:

```
  op/sprintf.........................FAILED at test 91
  lib/Benchmark......................FAILED at test 26
```

This is due to a bug in the C library’s printf routine. printf("’%e’", 0.) produces ’0.000000e+0’, but ANSI requires ’0.000000e+00’. QNX has acknowledged the bug.

121.2 AUTHOR

Norton T. Allen (allen@huarp.harvard.edu)

Chapter 122

README.solaris

Perl version 5 on Solaris systems

122.1 DESCRIPTION

This document describes various features of Sun's Solaris operating system that will affect how Perl version 5 (hereafter just perl) is compiled and/or runs. Some issues relating to the older SunOS 4.x are also discussed, though they may be out of date.

For the most part, everything should just work.

Starting with Solaris 8, perl5.00503 (or higher) is supplied with the operating system, so you might not even need to build a newer version of perl at all. The Sun-supplied version is installed in /usr/perl5 with /usr/bin/perl pointing to /usr/perl5/bin/perl. Do not disturb that installation unless you really know what you are doing. If you remove the perl supplied with the OS, you will render some bits of your system inoperable. If you wish to install a newer version of perl, install it under a different prefix from /usr/perl5. Common prefixes to use are /usr/local and /opt/perl.

You may wish to put your version of perl in the PATH of all users by changing the link /usr/bin/perl. This is OK, as all perl scripts shipped with Solaris use an explicit path. Solaris ships with a range of Solaris-specific modules. If you choose to install your own version of perl you will find the source of many of these modules is available on CPAN under the Sun::Solaris:: namespace.

Solaris may include two versions of perl, e.g. Solaris 9 includes both 5.005_03 and 5.6.1. This is to provide stability across Solaris releases, in cases where a later perl version has incompatibilities with the version included in the preceeding Solaris release. The default perl version will always be the most recent, and in general the old version will only be retained for one Solaris release. Note also that the default perl will NOT be configured to search for modules in the older version, again due to compatibility/stability concerns. As a consequence if you upgrade Solaris, you will have to rebuild/reinstall any additional CPAN modules that you installed for the previous Solaris version. See the CPAN manpage under 'autobundle' for a quick way of doing this.

As an interim measure, you may either change the #! line of your scripts to specifically refer to the old perl version, e.g. on Solaris 9 use #!/usr/perl5/5.00503/bin/perl to use the perl version that was the default for Solaris 8, or if you have a large number of scripts it may be more convenient to make the old version of perl the default on your system. You can do this by changing the appropriate symlinks under /usr/perl5 as follows (example for Solaris 9):

```
# cd /usr/perl5
# rm bin man pod
# ln -s ./5.00503/bin
# ln -s ./5.00503/man
# ln -s ./5.00503/lib/pod
# rm /usr/bin/perl
# ln -s ../perl5/5.00503/bin/perl /usr/bin/perl
```

In both cases this should only be considered to be a temporary measure - you should upgrade to the later version of perl as soon as is practicable.

Note also that the perl command-line utilities (e.g. perldoc) and any that are added by modules that you install will be under /usr/perl5/bin, so that directory should be added to your PATH.

122.1.1 Solaris Version Numbers.

For consistency with common usage, perl's Configure script performs some minor manipulations on the operating system name and version number as reported by uname. Here's a partial translation table:

```
         Sun:                      perl's Configure:
 uname    uname -r   Name          osname     osvers
 SunOS    4.1.3     Solaris 1.1    sunos      4.1.3
 SunOS    5.6       Solaris 2.6    solaris    2.6
 SunOS    5.8       Solaris 8      solaris    2.8
 SunOS    5.9       Solaris 9      solaris    2.9
 SunOS    5.10      Solaris 10     solaris    2.10
```

The complete table can be found in the Sun Managers' FAQ ftp://ftp.cs.toronto.edu/pub/jdd/sunmanagers/faq under "9.1) Which Sun models run which versions of SunOS?".

122.2 RESOURCES

There are many, many sources for Solaris information. A few of the important ones for perl:

Solaris FAQ

The Solaris FAQ is available at http://www.science.uva.nl/pub/solaris/solaris2.html.

The Sun Managers' FAQ is available at ftp://ftp.cs.toronto.edu/pub/jdd/sunmanagers/faq

Precompiled Binaries

Precompiled binaries, links to many sites, and much, much more are available at http://www.sunfreeware.com/ and http://www.blastwave.org/.

Solaris Documentation

All Solaris documentation is available on-line at http://docs.sun.com/.

122.3 SETTING UP

122.3.1 File Extraction Problems on Solaris.

Be sure to use a tar program compiled under Solaris (not SunOS 4.x) to extract the perl-5.x.x.tar.gz file. Do not use GNU tar compiled for SunOS4 on Solaris. (GNU tar compiled for Solaris should be fine.) When you run SunOS4 binaries on Solaris, the run-time system magically alters pathnames matching m#lib/locale# so that when tar tries to create lib/locale.pm, a file named lib/oldlocale.pm gets created instead. If you found this advice too late and used a SunOS4-compiled tar anyway, you must find the incorrectly renamed file and move it back to lib/locale.pm.

122.3.2 Compiler and Related Tools on Solaris.

You must use an ANSI C compiler to build perl. Perl can be compiled with either Sun's add-on C compiler or with gcc. The C compiler that shipped with SunOS4 will not do.

Include /usr/ccs/bin/ in your PATH.

Several tools needed to build perl are located in /usr/ccs/bin/: ar, as, ld, and make. Make sure that /usr/ccs/bin/ is in your PATH.

You need to make sure the following packages are installed (this info is extracted from the Solaris FAQ):

for tools (sccs, lex, yacc, make, nm, truss, ld, as): SUNWbtool, SUNWsprot, SUNWtoo

for libraries & headers: SUNWhea, SUNWarc, SUNWlibm, SUNWlibms, SUNWdfbh, SUNWcg6h, SUNWxwinc, SUNWolinc

for 64 bit development: SUNWarcx, SUNWbtoox, SUNWdplx, SUNWscpux, SUNWsprox, SUNWtoox, SUNWlmsx, SUNWlmx, SUNWlibCx

If you are in doubt which package contains a file you are missing, try to find an installation that has that file. Then do a

```
 $ grep /my/missing/file /var/sadm/install/contents
```

This will display a line like this:

/usr/include/sys/errno.h f none 0644 root bin 7471 37605 956241356 SUNWhea

The last item listed (SUNWhea in this example) is the package you need.

Avoid /usr/ucb/cc.

You don't need to have /usr/ucb/ in your PATH to build perl. If you want /usr/ucb/ in your PATH anyway, make sure that /usr/ucb/ is NOT in your PATH before the directory containing the right C compiler.

Sun's C Compiler

If you use Sun's C compiler, make sure the correct directory (usually /opt/SUNWspro/bin/) is in your PATH (before /usr/ucb/).

GCC

If you use gcc, make sure your installation is recent and complete. perl versions since 5.6.0 build fine with gcc > 2.8.1 on Solaris >= 2.6.

You must Configure perl with

```
 $ sh Configure -Dcc=gcc
```

If you don't, you may experience strange build errors.

If you have updated your Solaris version, you may also have to update your gcc. For example, if you are running Solaris 2.6 and your gcc is installed under /usr/local, check in /usr/local/lib/gcc-lib and make sure you have the appropriate directory, sparc-sun-solaris2.6/ or i386-pc-solaris2.6/. If gcc's directory is for a different version of Solaris than you are running, then you will need to rebuild gcc for your new version of Solaris.

You can get a precompiled version of gcc from http://www.sunfreeware.com/ or http://www.blastwave.org/. Make sure you pick up the package for your Solaris release.

If you wish to use gcc to build add-on modules for use with the perl shipped with Solaris, you should use the Solaris::PerlGcc module which is available from CPAN. The perl shipped with Solaris is configured and built with the Sun compilers, and the compiler configuration information stored in Config.pm is therefore only relevant to the Sun compilers. The Solaris:PerlGcc module contains a replacement Config.pm that is correct for gcc - see the module for details.

GNU as and GNU ld

The following information applies to gcc version 2. Volunteers to update it as appropropriate for gcc version 3 would be appreciated.

The versions of as and ld supplied with Solaris work fine for building perl. There is normally no need to install the GNU versions to compile perl.

If you decide to ignore this advice and use the GNU versions anyway, then be sure that they are relatively recent. Versions newer than 2.7 are apparently new enough. Older versions may have trouble with dynamic loading.

If you wish to use GNU ld, then you need to pass it the -Wl,-E flag. The hints/solaris_2.sh file tries to do this automatically by setting the following Configure variables:

```
ccdlflags="$ccdlflags -Wl,-E"
lddlflags="$lddlflags -Wl,-E -G"
```

However, over the years, changes in gcc, GNU ld, and Solaris ld have made it difficult to automatically detect which ld ultimately gets called. You may have to manually edit config.sh and add the -Wl,-E flags yourself, or else run Configure interactively and add the flags at the appropriate prompts.

If your gcc is configured to use GNU as and ld but you want to use the Solaris ones instead to build perl, then you'll need to add -B/usr/ccs/bin/ to the gcc command line. One convenient way to do that is with

```
$ sh Configure -Dcc='gcc -B/usr/ccs/bin/'
```

Note that the trailing slash is required. This will result in some harmless warnings as Configure is run:

```
gcc: file path prefix `/usr/ccs/bin/' never used
```

These messages may safely be ignored. (Note that for a SunOS4 system, you must use -B/bin/ instead.)

Alternatively, you can use the GCC_EXEC_PREFIX environment variable to ensure that Sun's as and ld are used. Consult your gcc documentation for further information on the -B option and the GCC_EXEC_PREFIX variable.

Sun and GNU make

The make under /usr/ccs/bin works fine for building perl. If you have the Sun C compilers, you will also have a parallel version of make (dmake). This works fine to build perl, but can sometimes cause problems when running 'make test' due to underspecified dependencies between the different test harness files. The same problem can also affect the building of some add-on modules, so in those cases either specify '-m serial' on the dmake command line, or use /usr/ccs/bin/make instead. If you wish to use GNU make, be sure that the set-group-id bit is not set. If it is, then arrange your PATH so that /usr/ccs/bin/make is before GNU make or else have the system administrator disable the set-group-id bit on GNU make.

Avoid libucb.

Solaris provides some BSD-compatibility functions in /usr/ucblib/libucb.a. Perl will not build and run correctly if linked against -lucb since it contains routines that are incompatible with the standard Solaris libc. Normally this is not a problem since the solaris hints file prevents Configure from even looking in /usr/ucblib for libraries, and also explicitly omits -lucb.

122.3.3 Environment for Compiling perl on Solaris

PATH

Make sure your PATH includes the compiler (/opt/SUNWspro/bin/ if you're using Sun's compiler) as well as /usr/ccs/bin/ to pick up the other development tools (such as make, ar, as, and ld). Make sure your path either doesn't include /usr/ucb or that it includes it after the compiler and compiler tools and other standard Solaris directories. You definitely don't want /usr/ucb/cc.

LD_LIBRARY_PATH

If you have the LD_LIBRARY_PATH environment variable set, be sure that it does NOT include /lib or /usr/lib. If you will be building extensions that call third-party shared libraries (e.g. Berkeley DB) then make sure that your LD_LIBRARY_PATH environment variable includes the directory with that library (e.g. /usr/local/lib).

If you get an error message

```
 dlopen: stub interception failed
```

it is probably because your LD_LIBRARY_PATH environment variable includes a directory which is a symlink to /usr/lib (such as /lib). The reason this causes a problem is quite subtle. The file libdl.so.1.0 actually *only* contains functions which generate 'stub interception failed' errors! The runtime linker intercepts links to "/usr/lib/libdl.so.1.0" and links in internal implementations of those functions instead. [Thanks to Tim Bunce for this explanation.]

122.4 RUN CONFIGURE.

See the INSTALL file for general information regarding Configure. Only Solaris-specific issues are discussed here. Usually, the defaults should be fine.

122.4.1 64-bit perl on Solaris.

See the INSTALL file for general information regarding 64-bit compiles. In general, the defaults should be fine for most people.

By default, perl-5.6.0 (or later) is compiled as a 32-bit application with largefile and long-long support.

General 32-bit vs. 64-bit issues.

Solaris 7 and above will run in either 32 bit or 64 bit mode on SPARC CPUs, via a reboot. You can build 64 bit apps whilst running 32 bit mode and vice-versa. 32 bit apps will run under Solaris running in either 32 or 64 bit mode. 64 bit apps require Solaris to be running 64 bit mode.

Existing 32 bit apps are properly known as LP32, i.e. Longs and Pointers are 32 bit. 64-bit apps are more properly known as LP64. The discriminating feature of a LP64 bit app is its ability to utilise a 64-bit address space. It is perfectly possible to have a LP32 bit app that supports both 64-bit integers (long long) and largefiles (> 2GB), and this is the default for perl-5.6.0.

For a more complete explanation of 64-bit issues, see the "Solaris 64-bit Developer's Guide" at http://docs.sun.com/

You can detect the OS mode using "isainfo -v", e.g.

```
 $ isainfo -v   # Ultra 30 in 64 bit mode
 64-bit sparcv9 applications
 32-bit sparc applications
```

By default, perl will be compiled as a 32-bit application. Unless you want to allocate more than ˜ 4GB of memory inside perl, or unless you need more than 255 open file descriptors, you probably don't need perl to be a 64-bit app.

Large File Support

For Solaris 2.6 and onwards, there are two different ways for 32-bit applications to manipulate large files (files whose size is > 2GByte). (A 64-bit application automatically has largefile support built in by default.)

First is the "transitional compilation environment", described in lfcompile64(5). According to the man page,

```
 The transitional compilation  environment  exports  all  the
 explicit 64-bit functions (xxx64()) and types in addition to
 all the regular functions (xxx()) and types. Both xxx()  and
 xxx64()  functions  are  available to the program source.  A
 32-bit application must use the xxx64() functions in  order
 to  access  large  files.  See the lf64(5) manual page for a
 complete listing of the 64-bit transitional interfaces.
```

The transitional compilation environment is obtained with the following compiler and linker flags:

```
 getconf LFS64_CFLAGS        -D_LARGEFILE64_SOURCE
 getconf LFS64_LDFLAG        # nothing special needed
 getconf LFS64_LIBS          # nothing special needed
```

Second is the "large file compilation environment", described in lfcompile(5). According to the man page,

```
 Each interface named xxx() that needs to access 64-bit entities
 to  access  large  files maps to a xxx64() call in the
 resulting binary. All relevant data types are defined to  be
 of correct size (for example, off_t has a typedef definition
 for a 64-bit entity).

 An application compiled in this environment is able  to  use
 the  xxx()  source interfaces to access both large and small
 files, rather than having to explicitly utilize the  transitional
 xxx64()  interface  calls to access large files.
```

Two exceptions are fseek() and ftell(). 32-bit applications should use fseeko(3C) and ftello(3C). These will get automatically mapped to fseeko64() and ftello64().

The large file compilation environment is obtained with

```
 getconf LFS_CFLAGS      -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64
 getconf LFS_LDFLAGS     # nothing special needed
 getconf LFS_LIBS        # nothing special needed
```

By default, perl uses the large file compilation environment and relies on Solaris to do the underlying mapping of interfaces.

Building an LP64 perl

To compile a 64-bit application on an UltraSparc with a recent Sun Compiler, you need to use the flag "-xarch=v9". getconf(1) will tell you this, e.g.

```
 $ getconf -a | grep v9
 XBS5_LP64_OFF64_CFLAGS:         -xarch=v9
 XBS5_LP64_OFF64_LDFLAGS:        -xarch=v9
 XBS5_LP64_OFF64_LINTFLAGS:      -xarch=v9
 XBS5_LPBIG_OFFBIG_CFLAGS:       -xarch=v9
 XBS5_LPBIG_OFFBIG_LDFLAGS:      -xarch=v9
 XBS5_LPBIG_OFFBIG_LINTFLAGS:    -xarch=v9
 _XBS5_LP64_OFF64_CFLAGS:        -xarch=v9
 _XBS5_LP64_OFF64_LDFLAGS:       -xarch=v9
 _XBS5_LP64_OFF64_LINTFLAGS:     -xarch=v9
 _XBS5_LPBIG_OFFBIG_CFLAGS:      -xarch=v9
 _XBS5_LPBIG_OFFBIG_LDFLAGS:     -xarch=v9
 _XBS5_LPBIG_OFFBIG_LINTFLAGS:   -xarch=v9
```

This flag is supported in Sun WorkShop Compilers 5.0 and onwards (now marketed under the name Forte) when used on Solaris 7 or later on UltraSparc systems.

If you are using gcc, you would need to use -mcpu=v9 -m64 instead. This option is not yet supported as of gcc 2.95.2; from install/SPECIFIC in that release:

```
GCC version 2.95 is not able to compile code correctly for sparc64
targets. Users of the Linux kernel, at least, can use the sparc32
program to start up a new shell invocation with an environment that
causes configure to recognize (via uname -a) the system as sparc-*-*
instead.
```

All this should be handled automatically by the hints file, if requested.

Long Doubles.

As of 5.8.1, long doubles are working if you use the Sun compilers (needed for additional math routines not included in libm).

122.4.2 Threads in perl on Solaris.

It is possible to build a threaded version of perl on Solaris. The entire perl thread implementation is still experimental, however, so beware.

122.4.3 Malloc Issues with perl on Solaris.

Starting from perl 5.7.1 perl uses the Solaris malloc, since the perl malloc breaks when dealing with more than 2GB of memory, and the Solaris malloc also seems to be faster.

If you for some reason (such as binary backward compatibility) really need to use perl's malloc, you can rebuild perl from the sources and Configure the build with

```
$ sh Configure -Dusemymalloc
```

You should not use perl's malloc if you are building with gcc. There are reports of core dumps, especially in the PDL module. The problem appears to go away under -DDEBUGGING, so it has been difficult to track down. Sun's compiler appears to be okay with or without perl's malloc. [XXX further investigation is needed here.]

122.5 MAKE PROBLEMS.

Dynamic Loading Problems With GNU as and GNU ld

If you have problems with dynamic loading using gcc on SunOS or Solaris, and you are using GNU as and GNU ld, see the section §122.3.2 above.

ld.so.1: ./perl: fatal: relocation error:

If you get this message on SunOS or Solaris, and you're using gcc, it's probably the GNU as or GNU ld problem in the previous item §122.3.2.

dlopen: stub interception failed

The primary cause of the 'dlopen: stub interception failed' message is that the LD_LIBRARY_PATH environment variable includes a directory which is a symlink to /usr/lib (such as /lib). See §122.3.3 above.

#error "No DATAMODEL_NATIVE specified"

This is a common error when trying to build perl on Solaris 2.6 with a gcc installation from Solaris 2.5 or 2.5.1. The Solaris header files changed, so you need to update your gcc installation. You can either rerun the fixincludes script from gcc or take the opportunity to update your gcc installation.

sh: ar: not found

This is a message from your shell telling you that the command ’ar’ was not found. You need to check your PATH environment variable to make sure that it includes the directory with the ’ar’ command. This is a common problem on Solaris, where ’ar’ is in the /usr/ccs/bin/ directory.

122.6 MAKE TEST

122.6.1 op/stat.t test 4 in Solaris

op/stat.t test 4 may fail if you are on a tmpfs of some sort. Building in /tmp sometimes shows this behavior. The test suite detects if you are building in /tmp, but it may not be able to catch all tmpfs situations.

122.6.2 nss_delete core dump from op/pwent or op/grent

See nss_delete core dump from op/pwent or op/grent in *perlhpux*.

122.7 PREBUILT BINARIES OF PERL FOR SOLARIS.

You can pick up prebuilt binaries for Solaris from http://www.sunfreeware.com/, http://www.blastwave.org, ActiveState http://www.activestate.com/, and http://www.perl.com/ under the Binaries list at the top of the page. There are probably other sources as well. Please note that these sites are under the control of their respective owners, not the perl developers.

122.8 RUNTIME ISSUES FOR PERL ON SOLARIS.

122.8.1 Limits on Numbers of Open Files on Solaris.

The stdio(3C) manpage notes that for LP32 applications, only 255 files may be opened using fopen(), and only file descriptors 0 through 255 can be used in a stream. Since perl calls open() and then fdopen(3C) with the resulting file descriptor, perl is limited to 255 simultaneous open files, even if sysopen() is used. If this proves to be an insurmountable problem, you can compile perl as a LP64 application, see Building an LP64 perl for details. Note also that the default resource limit for open file descriptors on Solaris is 255, so you will have to modify your ulimit or rctl (Solaris 9 onwards) appropriately.

122.9 SOLARIS-SPECIFIC MODULES.

See the modules under the Solaris:: and Sun::Solaris namespaces on CPAN, see http://www.cpan.org/modules/by-module/Solaris/ and http://www.cpan.org/modules/by-module/Sun/.

122.10 SOLARIS-SPECIFIC PROBLEMS WITH MODULES.

122.10.1 Proc::ProcessTable on Solaris

Proc::ProcessTable does not compile on Solaris with perl5.6.0 and higher if you have LARGEFILES defined. Since largefile support is the default in 5.6.0 and later, you have to take special steps to use this module.

The problem is that various structures visible via procfs use off_t, and if you compile with largefile support these change from 32 bits to 64 bits. Thus what you get back from procfs doesn’t match up with the structures in perl, resulting in garbage. See proc(4) for further discussion.

A fix for Proc::ProcessTable is to edit Makefile to explicitly remove the largefile flags from the ones MakeMaker picks up from Config.pm. This will result in Proc::ProcessTable being built under the correct environment. Everything should then be OK as long as Proc::ProcessTable doesn’t try to share off_t’s with the rest of perl, or if it does they should be explicitly specified as off64_t.

122.10.2 BSD::Resource on Solaris

BSD::Resource versions earlier than 1.09 do not compile on Solaris with perl 5.6.0 and higher, for the same reasons as Proc::ProcessTable. BSD::Resource versions starting from 1.09 have a workaround for the problem.

122.10.3 Net::SSLeay on Solaris

Net::SSLeay requires a /dev/urandom to be present. This device is available from Solaris 9 onwards. For earlier Solaris versions you can either get the package SUNWski (packaged with several Sun software products, for example the Sun WebServer, which is part of the Solaris Server Intranet Extension, or the Sun Directory Services, part of Solaris for ISPs) or download the ANDIrand package from http://www.cosy.sbg.ac.at/˜andi/. If you use SUNWski, make a symbolic link /dev/urandom pointing to /dev/random.

It may be possible to use the Entropy Gathering Daemon (written in Perl!), available from http://www.lothar.com/tech/crypto/.

122.11 SunOS 4.x

In SunOS 4.x you most probably want to use the SunOS ld, /usr/bin/ld, since the more recent versions of GNU ld (like 2.13) do not seem to work for building Perl anymore. When linking the extensions, the GNU ld gets very unhappy and spews a lot of errors like this

```
  ... relocation truncated to fit: BASE13 ...
```

and dies. Therefore the SunOS 4.1 hints file explicitly sets the ld to be /usr/bin/ld.

As of Perl 5.8.1 the dynamic loading of libraries (DynaLoader, XSLoader) also seems to have become broken in in SunOS 4.x. Therefore the default is to build Perl statically.

Running the test suite in SunOS 4.1 is a bit tricky since the *lib/Tie/File/t/09_gen_rs* test hangs (subtest #51, FWIW) for some unknown reason. Just stop the test and kill that particular Perl process.

There are various other failures, that as of SunOS 4.1.4 and gcc 3.2.2 look a lot like gcc bugs. Many of the failures happen in the Encode tests, where for example when the test expects "0" you get "&#48;" which should after a little squinting look very odd indeed. Another example is earlier in *t/run/fresh_perl* where chr(0xff) is expected but the test fails because the result is chr(0xff). Exactly.

This is the "make test" result from the said combination:

```
  Failed 27 test scripts out of 745, 96.38% okay.
```

Running the `harness` is painful because of the many failing Unicode-related tests will output megabytes of failure messages, but if one patiently waits, one gets these results:

```
 Failed Test                     Stat Wstat Total Fail  Failed  List of Failed
 -----------------------------------------------------------------------------
 ...
 ../ext/Encode/t/at-cn.t            4  1024    29    4  13.79%  14-17
 ../ext/Encode/t/at-tw.t           10  2560    17   10  58.82%  2 4 6 8 10 12
                                                                14-17
 ../ext/Encode/t/enc_data.t        29  7424    ??   ??       %  ??
 ../ext/Encode/t/enc_eucjp.t       29  7424    ??   ??       %  ??
 ../ext/Encode/t/enc_module.t      29  7424    ??   ??       %  ??
 ../ext/Encode/t/encoding.t        29  7424    ??   ??       %  ??
 ../ext/Encode/t/grow.t            12  3072    24   12  50.00%  2 4 6 8 10 12 14
                                                                16 18 20 22 24
  Failed Test                     Stat Wstat Total Fail  Failed  List of Failed
 ------------------------------------------------------------------------------
```

```
../ext/Encode/t/guess.t         255 65280    29   40 137.93%  10-29
../ext/Encode/t/jperl.t          29  7424    15   30 200.00%  1-15
../ext/Encode/t/mime-header.t     2   512    10    2  20.00%  2-3
../ext/Encode/t/perlio.t         22  5632    38   22  57.89%  1-4 9-16 19-20
                                                               23-24 27-32
../ext/List/Util/t/shuffle.t      0   139    ??   ??       %  ??
../ext/PerlIO/t/encoding.t                   14    1   7.14%  11
../ext/PerlIO/t/fallback.t                    9    2  22.22%  3 5
../ext/Socket/t/socketpair.t      0     2    45   70 155.56%  11-45
../lib/CPAN/t/vcmp.t                         30    1   3.33%  25
../lib/Tie/File/t/09_gen_rs.t     0    15    ??   ??       %  ??
../lib/Unicode/Collate/t/test.t             199   30  15.08%  7 26-27 71-75
                                                               81-88 95 101
                                                               103-104 106 108-
                                                               109 122 124 161
                                                               169-172
../lib/sort.t                     0   139   119   26  21.85%  107-119
op/alarm.t                                    4    1  25.00%  4
op/utfhash.t                                 97    1   1.03%  31
run/fresh_perl.t                             91    1   1.10%  32
uni/tr_7jis.t                                ??   ??       %  ??
uni/tr_eucjp.t                   29  7424     6   12 200.00%  1-6
uni/tr_sjis.t                    29  7424     6   12 200.00%  1-6
56 tests and 467 subtests skipped.
Failed 27/811 test scripts, 96.67% okay. 1383/75399 subtests failed, 98.17% okay.
```

The alarm() test failure is caused by system() apparently blocking alarm(). That is probably a libc bug, and given that SunOS 4.x has been end-of-lifed years ago, don't hold your breath for a fix. In addition to that, don't try anything too Unicode-y, especially with Encode, and you should be fine in SunOS 4.x.

122.12 AUTHOR

The original was written by Andy Dougherty *doughera@lafayette.edu* drawing heavily on advice from Alan Burlison, Nick Ing-Simmons, Tim Bunce, and many other Solaris users over the years.

Please report any errors, updates, or suggestions to *perlbug@perl.org*.

122.13 LAST MODIFIED

$Id: README.solaris,v 1.4 2000/11/11 20:29:58 doughera Exp $

Chapter 123

perluts

Perl under UTS

123.1 SYNOPSIS

This document can be read *as is*: as *README.uts*, or you can read it after you build your package using "man perluts".

The purpose is to help you build Perl for UTS, which, if you follow these instructions, should be easy, and result in a solidly working installation.

123.2 DESCRIPTION

Perl 5.7.2 (Developmental) or Perl 5.8.x (forthcoming) for UTS

123.3 BUILDING PERL ON UTS

NOTE: Some sites have redefined the way uname works, and if yours does this, special steps must be taken so that Configure can recognize your system as a UTS system. To see if you are in this category, issue the command "uname -a". It should look something like:

```
  uts juno 4 4.4 9672 370
```

At any rate, the first field should be "uts". If this is not the case; supposing it is, say telcoUTS, create a script, uts/uname (i.e. uname, in the subdirectory "uts" of the main Perl source dir): # uname /usr/bin/uname "$@" | sed -e 's/^telcoUTS/uts/'

and when you execute Configure, do it as below, except for adding PATH=uts:$PATH as a prefix. I.e. do:

```
   PATH=uts:$PATH ./Configure ...
```

There is no need to do an interactive configure, just type

```
  ./Configure -de [-Dusedevel] [-Doptimize=-g ] 2>&1 | tee Conf.out
```

"-Dusedevel" may be required to configure Perl 5.7.2 non-interactively. Use -Doptimize=-g if you want to run Perl under sdb or gdb, OR if you want to be able to use the -D command line flags to perl, which are occasionally useful in debugging perl scripts.

In this and the following steps, the "2>&1 | tee XXX.out" records all output from the process, which will be useful if anything unexpected goes wrong.

Then do the compilation with

```
make 2>&1 | tee make.out
```

Finally, test using

```
make test 2>&1 | tee make-test.out
```

In the output, the only failures you should see should look like:

```
lib/Math/BigInt/t/bigfltpm.........Use of uninitialized value ...
FAILED at test 57
lib/Math/BigInt/t/bigintc..........ok
lib/Math/BigInt/t/bigintpm.........FAILED at test 204
lib/Math/BigInt/t/mbimbf...........Use of uninitialized value ...
Illegal division by zero at ../lib/Math/BigInt/Calc.pm line 314.
FAILED at test 71
lib/Math/Complex...................exp: OVERFLOW
FAILED at test 250
lib/Math/Trig......................exp: OVERFLOW
ok
lib/Memoize/t/array................ok
     ...
lib/Net/protoent...................ok
lib/Net/servent....................FAILED at test 0
```

This means that everything passes except for some problems in the packages "Math::BigInt", "Math::Complex", and "Math::Trig". The lib/Net/servent failure seems to be a bug in the test program. To confirm this, from the main Perl source dir, do:

```
LD_LIBRARY_PATH=‘pwd‘ ./perl -Ilib lib/Net/servent.t
```

and it should output

```
1..3
ok 1
ok 2
ok 3
```

123.4 Installing the built perl on UTS

Run the command "make install"

123.5 AUTHOR

```
Hal Morris
UTS Global LLC
email: hom00@utsglobal.com
```

Chapter 124

README.vmesa

Building and installing Perl for VM/ESA.

124.1 SYNOPSIS

This document will help you Configure, build, test and install Perl on VM/ESA.

124.2 DESCRIPTION

This is a fully ported perl for VM/ESA 2.3.0. It may work on other versions, but that's the one we've tested it on.

If you've downloaded the binary distribution, it needs to be installed below /usr/local. Source code distributions have an automated ‘make install‘ step that means you do not need to extract the source code below /usr/local (though that is where it will be installed by default). You may need to worry about the networking configuration files discussed in the last bullet below.

124.2.1 Unpacking Perl Distribution on VM/ESA

To extract an ASCII tar archive on VM/ESA, try this:

```
   pax -o to=IBM-1047,from=ISO8859-1 -r < latest.tar
```

124.2.2 Setup Perl and utilities on VM/ESA

GNU make for VM/ESA, which may be required for the build of perl, is available from:

```
  http://vm.marist.edu/~neale/vmoe.html
```

124.2.3 Configure Perl on VM/ESA

Once you've unpacked the distribution, run Configure (see INSTALL for full discussion of the Configure options), and then run make, then "make test" then "make install" (this last step may require UID=0 privileges).

There is a "hints" file for vmesa that specifies the correct values for most things. Some things to watch out for are:

- this port does support dynamic loading but it's not had much testing
- Don't turn on the compiler optimization flag "-O". There's a bug in the compiler (APAR PQ18812) that generates some bad code the optimizer is on.

- As VM/ESA doesn't fully support the fork() API programs relying on this call will not work. I've replaced fork()/exec() with spawn() and the standalone exec() with spawn(). This has a side effect when opening unnamed pipes in a shell script: there is no child process generated under.
- At the moment the hints file for VM/ESA basically bypasses all of the automatic configuration process. This is because Configure relies on: 1. The header files living in the Byte File System (you could put the there if you want); 2. The C preprocessor including the #include statements in the preprocessor output (.i) file.

124.2.4 Testing Anomalies of Perl on VM/ESA

The 'make test' step runs a Perl Verification Procedure, usually before installation. As the 5.6.1 kit was being assembled the following "failures" were known to appear on some machines during 'make test' (mostly due to ASCII vs. EBCDIC conflicts), your results may differ:

[the list of failures being compiled]

124.2.5 Usage Hints for Perl on VM/ESA

When using perl on VM/ESA please keep in mind that the EBCDIC and ASCII character sets are different. Perl builtin functions that may behave differently under EBCDIC are mentioned in the perlport.pod document.

OpenEdition (UNIX System Services) does not (yet) support the #! means of script invocation. See:

```
    head `whence perldoc`
```

for an example of how to use the "eval exec" trick to ask the shell to have perl run your scripts for you.

124.3 AUTHORS

Neale Ferguson.

124.4 SEE ALSO

INSTALL, *perlport*, *perlebcdic*.

124.4.1 Mailing list for Perl on VM/ESA

If you are interested in the VM/ESA, z/OS (formerly known as OS/390) and POSIX-BC (BS2000) ports of Perl then see the perl-mvs mailing list. To subscribe, send an empty message to perl-mvs-subscribe@perl.org.

See also:

```
    http://lists.perl.org/showlist.cgi?name=perl-mvs
```

There are web archives of the mailing list at:

```
    http://www.xray.mpe.mpg.de/mailing-lists/perl-mvs/
    http://archive.develooper.com/perl-mvs@perl.org/
```

Chapter 125

perlvms

VMS-specific documentation for Perl

125.1 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.

125.2 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..

125.3 Organization of Perl Images

125.3.1 Core Images

During the installation 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 extensions. It is used to generate several files needed to build the complete Perl and various extensions. Once you've finished installing Perl, you can delete this image.

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. You should place this image in *Sys$ Share*, or define the logical name *PerlShr* to translate to the full file specification of this image. It should be world readable. (Remember that if a user has execute only access to *PerlShr*, VMS will treat it as if it were a privileged shareable image, and will therefore require all downstream shareable images to be INSTALLed, etc.)

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.

125.3.2 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.

125.3.3 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).

125.3.4 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
```

N.B. The procedure by which extensions are built and tested creates several levels (at least 4) under the directory in which the extension's source files live. For this reason if you are runnning a version of VMS prior to V7.1 you shouldn't nest the source directory too deeply in your directory structure lest you exceed RMS' maximum of 8 levels of subdirectory in a filespec. (You can use rooted logical names to get another 8 levels of nesting, if you can't place the files near the top of the physical directory structure.)

VMS support for this process in the current release of Perl is sufficient to handle most extensions. However, it does not yet recognize extra libraries required to build shareable images which are part of an extension, so these must be added to the linker options file for the extension by hand. For instance, if the *PGPLOT* extension to Perl requires the *PGPLOTSHR.EXE* shareable image in order to properly link the Perl extension, then the line `PGPLOTSHR/Share` must be added to the linker options file *PGPLOT.Opt* produced during the build process for the Perl extension.

By default, the shareable image for an extension is placed in the *[.lib.site_perl.autoArch.Extname]* directory of the installed Perl directory tree (where *Arch* is *VMS_VAX* or *VMS_AXP*, and *Extname* is the name of the extension, with each `::` translated to `.`). (See the MakeMaker documentation for more details on installation options for extensions.) However, it can be manually placed in any of several locations:

- 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.

125.4 File specifications

125.4.1 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.

Filenames are, of course, still case-insensitive. For consistency, most Perl routines return filespecs using lower case letters only, regardless of the case used in the arguments passed to them. (This is true only when running under VMS; Perl respects the case-sensitivity of OSs like Unix.)

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.

125.4.2 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.

125.4.3 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 512. 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 double the MBX size from the default within a Perl program, use `$ENV{'PERL_MBX_SIZE'} = 1024;` and then open and use pipe constructs. An alternative would be to issue the command:

```
    $ Define PERL_MBX_SIZE 1024
```

before running your wide record pipe program. A larger value may improve performance at the expense of the BYTLM UAF quota.

125.5 PERL5LIB and PERLLIB

The PERL5LIB and PERLLIB logical names work as documented in *perl*, except that the element separator is '|' instead of ':'. The directory specifications may use either VMS or Unix syntax.

125.6 Command line

125.6.1 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.

125.6.2 Command line switches

The following command line switches behave differently under VMS than described in *perlrun*. 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.

-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.

125.7 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, endpwent, eof, eval, exec*,
exists, exit, exp, fileno, getc, getlogin, getppid,
getpwent*, getpwnam*, getpwuid*, glob, gmtime*, goto,
grep, hex, import, index, int, join, keys, kill*,
last, lc, lcfirst, length, local, localtime, log, 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, redo, ref, rename,
require, reset, return, reverse, rewinddir, rindex,
rmdir, s///, scalar, seek, seekdir, select(internal),
select (system call)*, setpwent, shift, sin, sleep,
sort, splice, split, sprintf, sqrt, srand, stat,
study, substr, 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, flock, fork*,
getpgrp, getpriority, getgrent, getgrgid,
getgrnam, setgrent, endgrent, ioctl, link, lstat,
msgctl, msgget, msgsend, msgrcv, readlink, semctl,
semget, semop, setpgrp, setpriority, shmctl, shmget,
shmread, shmwrite, socketpair, symlink, 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
```

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.

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;
    }
```

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 *perlfork*). In the meantime, use `system`, backticks, or piped filehandles to create subprocesses.

getpwent

getpwnam

getpwuid

These operators obtain the information described in *perlfunc*, 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 CRTL's `kill()` function, so it will behave according to that function's documentation. If you send a SIGKILL, however, the $DELPRC system service is called directly. This insures that the target process is actually deleted, if at all possible. (The CRTL's `kill()` function is presently implemented via $FORCEX, which is ignored by supervisor-mode images like DCL.)

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 *perlfunc*, 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.

utime LIST

Since ODS-2, the VMS file structure for disk files, does not keep track of access times, this operator changes only the modification time of the file (VMS revision date).

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.

125.8 Perl variables

The following VMS-specific information applies to the indicated "special" Perl variables, in addition to the general information in *perlvar*. 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 CGI processing by some HTTP servers, then 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.

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 00) 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.) An element of the CRTL `environ` array can be set only if your copy of Perl knows about the CRTL's `setenv()` function. (This is present only in some versions of the DECCRTL; check `$Config{d_setenv}` to see whether your copy of Perl was built with a CRTL that has this function.)

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; this can only be done if your copy of Perl knows about the CRTL `setenv()` function.) 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 logicals 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 logicals representing process-permanent files, such as `SYS$INPUT` and `SYS$OUTPUT`. The translations for these logicals 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 logicals, 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.

$ ˆE

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.

$?

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 are derived from the severity portion of the subprocess' exit status: 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 fatal error.

As a result, `$?` will always be zero if the subprocess' exit status indicated successful completion, and non-zero if a warning or error occurred. Conversely, when setting `$?` 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 exit in *perlport*.

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).

$ |

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.

125.9 Standard modules with VMS-specific differences

125.9.1 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.

125.10 Revision date

This document was last updated on 01-May-2002, for Perl 5, patchlevel 8.

125.11 AUTHOR

Charles Bailey bailey@cor.newman.upenn.edu Craig Berry craigberry@mac.com Dan Sugalski dan@sidhe.org

Chapter 126

README.vos

Perl for Stratus VOS

126.1 SYNOPSIS

This file contains notes for building perl on the Stratus VOS operating system. Perl is a scripting or macro language that is popular on many systems. See *perlbook* for a number of good books on Perl.

These are instructions for building Perl from source. Most people can simply download a pre-compiled distribution from the VOS anonymous FTP site. If you are running VOS Release 14.2.0 or earlier, download Perl from ftp://ftp.stratus.com/pub/vos/posix/alpha/alpha.html If you are running VOS Release 14.3.0 or later, download Perl from ftp://ftp.stratus.com/pub/vos/posix/ga/ga.html Instructions for unbundling the Perl distribution file are at ftp://ftp.stratus.com/pub/vos/utility/utility.html

If you are running VOS Release 14.4.1 or later, you can obtain a pre-compiled, supported copy of perl by purchasing Release 2.0.1 (or later) of the VOS GNU C++ and GNU Tools product from Stratus Technologies.

126.1.1 Multiple methods to build perl for VOS

If you elect to build perl from its source code, you have several different ways that you can build perl. The method that you use depends on the version of VOS that you are using and on the architecture of your Stratus hardware platform.

1. If you have a Stratus XA2000 (Motorola 68k-based) platform, you must build perl using the alpha version of VOS POSIX support and using the VOS Standard C Cross-compiler. You must build perl on VOS Release 14.1.0 (or later) on an XA/R or Continuum platform.

   This version of perl is properly called "miniperl" because it does not contain the complete perl functionality.

   You must build perl with the compile_perl.cm command macro found in the vos subdirectory.

2. If you have a Stratus XA/R (Intel i860-based) platform, you must build perl using the alpha version of VOS POSIX support and using the VOS Standard C compiler or cross-compiler. You must build perl on VOS Release 14.1.0 (or later) on an XA/R or Continuum platform.

   This version of perl is properly called "miniperl" because it does not contain the complete perl functionality.

   You must build perl with the compile_perl.cm command macro found in the vos subdirectory.

3. If you have a Stratus Continuum (PA-RISC-based) platform that is running a version of VOS earlier than VOS 14.3.0, you must build perl using the alpha version of VOS POSIX support and using the VOS Standard C compiler or cross-compiler. You must build perl on VOS Release 14.1.0 (or later) on an XA/R or Continuum platform.

   This version of perl is properly called "miniperl" because it does not contain the complete perl functionality.

   You must build perl with the compile_perl.cm command macro found in the vos subdirectory.

4. If you have a Stratus Continuum (PA-RISC-based) platform that is running VOS Release 14.3.0 through VOS Release 14.4.1, you must build perl using the generally-available version of VOS POSIX support, and using either the VOS Standard C compiler or the VOS GNU C compiler. You must build perl on VOS Release 14.3.0 (or later) on a Continuum platform.

   This version of perl is properly called "miniperl" because it does not contain the complete perl functionality.

   You must build perl with the compile_perl.cm command macro found in the vos subdirectory.

5. If you have a Stratus Continuum (PA-RISC-based) platform that is running VOS Release 14.5.0 or later, you can either use the previous method to build "miniperl" or you can build "full perl", which contains the complete functionality of perl. I strongly recommend that you build full perl. To build full perl, you must use the generally-available version of VOS POSIX support. You must use the VOS GNU C compiler and the VOS GNU C/C++ and GNU Tools Release 2.0.1 (or later) product. You must build full perl on VOS Release 14.5.0 (or later) on a Continuum platform.

   You must build full perl with the compile_full_perl.cm command macro found in the vos subdirectory.

126.1.2 Stratus POSIX Support

Note that there are two different implementations of POSIX.1 support on VOS. There is an alpha version of POSIX that is available from the Stratus anonymous ftp site (ftp://ftp.stratus.com/pub/vos/posix/alpha/alpha.html). There is a generally-available version of POSIX that comes with VOS Release 14.3.0 or higher. This port of POSIX will compile and bind with either version of POSIX.

Most of the Perl features should work on VOS regardless of which version of POSIX that you are using. However, the alpha version of POSIX is missing a number of key functions, and therefore any attempt by perl.pm to call the following unimplemented POSIX functions will result in an error message and an immediate and fatal call to the VOS debugger. They are "dup", "fork", and "waitpid". The lack of these functions prevents you from starting VOS commands and grabbing their output in perl. The workaround is to run the commands outside of perl, then have perl process the output file. These functions are all available in the generally-available version of POSIX.

126.2 INSTALLING PERL IN VOS

126.2.1 Compiling Perl 5 on VOS

Before you can build Perl 5 on VOS, you need to have or acquire the following additional items.

1. The VOS Standard C Compiler (or the VOS Standard C Cross-Compiler) and the VOS C Runtime. If you are using the generally-available version of POSIX support, you may instead use the VOS GNU C/C++ Compiler. These are standard Stratus products.

2. Either the VOS OS TCP/IP or STCP product set. If you are building with the alpha version of POSIX you need the OS TCP/IP product set. If you are building with the generally-available version of POSIX you need the STCP product set. These are standard Stratus products.

3. Either the alpha or generally-available version of the VOS POSIX.1 environment.

   The alpha version of POSIX.1 support is available on the Stratus FTP site. Login anonymously to ftp.stratus.com and get the file /pub/vos/posix/alpha/posix.save.evf.gz in binary file-transfer mode. Or use the Uniform Resource Locator (URL) ftp://ftp.stratus.com/pub/vos/posix/alpha/posix.save.evf.gz from your web browser. Instructions for unbundling this file are at ftp://ftp.stratus.com/pub/vos/utility/utility.html This is NOT a standard Stratus product.

   In VOS Release 14.3.0, the generally-available version of POSIX.1 support is bundled with the VOS Standard C compiler (or Standard C Cross-Compiler). In VOS Release 14.4.0 or higher, it is also bundled with the VOS C Runtime. These are standard Stratus products.

4. You must compile this version of Perl 5 on VOS Release 14.1.0 or higher because some of the perl source files contain more than 32,767 source lines. Due to VOS release-compatibility rules, this port of perl may not execute on VOS Release 12 or earlier.

5. If you are using the generally-available version of VOS POSIX support, then you should also acquire the VOS GNU C/C++ Compiler and GNU Tools product. When perl is built with this version of POSIX support, it assumes that it can find "bash", "sed" and other POSIX-compatible commands in the directory /system/gnu_library/bin.

To build perl using the supplied VOS command macros, change to the "vos" subdirectory and type the command "compile_perl -processor X", where X is the processor type (mc68020, i80860, pa7100, pa8000) that you wish to use. Note that the generally-available version of POSIX.1 support is not available for the mc68020 or i80860 processors.

Use the "-version alpha" control argument to build perl with the alpha version of POSIX support, and use the "-version ga" control argument to build it with the generally-available version of POSIX. The default is "ga".

Use the "-compiler cc" control argument to build perl with the VOS Standard C compiler. Use the "-compiler gcc" control argument to build it with the GNU GCC compiler. The default is "cc".

You must have purchased the VOS Standard C Cross Compiler in order to compile perl for a processor type that is different from the processor type of the module.

Note that code compiled for the pa7100 processor type can execute on the PA7100, PA8000, PA8500 and PA8600 processors, and that code compiled for the pa8000 processor type can execute on the PA8000, PA8500 and PA8600 processors.

To build full perl using the supplied Configure script and makefiles, change to the "vos" subdirectory and type the command "compile_full_perl" or "start_process compile_full_perl". This will configure, build, and test perl.

126.2.2 Installing Perl 5 on VOS

1. If you have built perl using the Configure script, ensure that you have modify permission to `>system>ported` and type

   ```
   gmake install
   ```

2. If you have built perl using any of the other methods, type

   ```
   install_perl -processor PROCESSOR -name NAME
   ```

   where PROCESSOR is mc68020, i80860, pa7100, or pa8000, as appropriate, and NAME is perl or perl5, according to which name you wish to use.

   This command macro will install perl and all of its related files in the proper directories.

3. While there are currently no architecture-specific extensions or modules distributed with perl, the following directories can be used to hold such files:

   ```
   >system>ported>lib>perl5>5.8.0>68k
   >system>ported>lib>perl5>5.8.0>860
   >system>ported>lib>perl5>5.8.0>7100
   >system>ported>lib>perl5>5.8.0>8000
   ```

4. Site-specific perl extensions and modules can be installed in one of two places. Put architecture-independent files into:

   ```
   >system>ported>lib>perl5>site_perl>5.8.0
   ```

   Put site-specific architecture-dependent files into one of the following directories:

   ```
   >system>ported>lib>perl5>site_perl>5.8.0>68k
   >system>ported>lib>perl5>site_perl>5.8.0>860
   >system>ported>lib>perl5>site_perl>5.8.0>7100
   >system>ported>lib>perl5>site_perl>5.8.0>8000
   ```

5. You can examine the @INC variable from within a perl program to see the order in which Perl searches these directories.

126.3 USING PERL IN VOS

126.3.1 Unimplemented Features of Perl on VOS

If perl is built with the alpha version of VOS POSIX.1 support and if it attempts to call an unimplemented VOS POSIX.1 function, it will print a fatal error message and enter the VOS debugger. This error is not recoverable. See vos_dummies.c for a list of the unimplemented POSIX.1 functions. To see what functions are unimplemented and what the error message looks like, compile and execute "test_vos_dummies.c".

126.3.2 Restrictions of Perl on VOS

This port of Perl version 5 to VOS prefers Unix-style, slash-separated pathnames over VOS-style greater-than-separated pathnames. VOS-style pathnames should work in most contexts, but if you have trouble, replace all greater-than characters by slash characters. Because the slash character is used as a pathname delimiter, Perl cannot process VOS pathnames containing a slash character in a directory or file name; these must be renamed.

This port of Perl also uses Unix-epoch date values internally. As long as you are dealing with ASCII character string representations of dates, this should not be an issue. The supported epoch is January 1, 1980 to January 17, 2038.

See the file pod/perlport.pod for more information about the VOS port of Perl.

126.3.3 Handling of underflow and overflow

Prior to VOS Release 14.7.0, VOS does not support automatically mapping overflowed floating-point values to +infinity, nor automatically mapping underflowed floating-point values to zero, unlike many other platforms. The Perl pack function has been modified to perform such mapping in software on VOS. Performing other floating-point computations that underflow or overflow will probably result in SIGFPE. Don't push your luck.

As of VOS Release 14.7.0, the VOS POSIX runtime sets up the PA-RISC hardware floating-point status register so that the overflow and underflow exceptions do not trap, but instead automatically convert the result to infinity or zero, as appropriate. As of this writing, there are still floating-point operations that can trap, for example, subtracting two infinite values. This is recorded as suggestion posix-1022, which is not yet fixed.

126.4 TEST STATUS

When Perl 5.8.3 is built using the native build process on VOS Release 14.7.0 and GNU C++/GNU Tools 2.0.2a, all but three attempted tests either pass or result in TODO (ignored) failures. The tests that fail are:

t/io/tell.t, test 28 t/op/pack.t, test 39 lib/Net/ing/t/450_service.t, test 8

126.5 SUPPORT STATUS

I'm offering this port "as is". You can ask me questions, but I can't guarantee I'll be able to answer them. There are some excellent books available on the Perl language; consult a book seller.

If you want a supported version of perl for VOS, purchase the VOS GNU C++ and GNU Tools Release 2.0.1 (or later) product from Stratus Technologies, along with a support contract (or from anyone else who will sell you support).

126.6 AUTHOR

Paul Green (Paul.Green@stratus.com)

126.7 LAST UPDATE

January 15, 2004

Chapter 127

perlwin32

Perl under Windows

127.1 SYNOPSIS

These are instructions for building Perl under Windows 9x/NT/2000/XP on the Intel x86 and Itanium architectures.

127.2 DESCRIPTION

Before you start, you should glance through the README file found in the top-level directory to which the Perl distribution was extracted. Make sure you read and understand the terms under which this software is being distributed.

Also make sure you read BUGS AND CAVEATS below for the known limitations of this port.

The INSTALL file in the perl top-level has much information that is only relevant to people building Perl on Unix-like systems. In particular, you can safely ignore any information that talks about "Configure".

You may also want to look at two other options for building a perl that will work on Windows NT: the README.cygwin and README.os2 files, each of which give a different set of rules to build a Perl that will work on Win32 platforms. Those two methods will probably enable you to build a more Unix-compatible perl, but you will also need to download and use various other build-time and run-time support software described in those files.

This set of instructions is meant to describe a so-called "native" port of Perl to Win32 platforms. This includes both 32-bit and 64-bit Windows operating systems. The resulting Perl requires no additional software to run (other than what came with your operating system). Currently, this port is capable of using one of the following compilers on the Intel x86 architecture:

```
      Borland C++               version 5.02 or later
      Microsoft Visual C++      version 4.2 or later
      MinGW with gcc            gcc version 2.95.2 or later
```

The last of these is a high quality freeware compiler. Use version 3.2.x or later for the best results with this compiler.

This port can also be built on the Intel IA64 using:

```
      Microsoft Platform SDK    Nov 2001 (64-bit compiler and tools)
```

The MS Platform SDK can be downloaded from http://www.microsoft.com/.

This port fully supports MakeMaker (the set of modules that is used to build extensions to perl). Therefore, you should be able to build and install most extensions found in the CPAN sites. See Usage Hints for Perl on Win32 below for general hints about this.

127.2.1 Setting Up Perl on Win32

Make

You need a "make" program to build the sources. If you are using Visual C++ or the Platform SDK tools under Windows NT/2000/XP, nmake will work. All other builds need dmake.

dmake is a freely available make that has very nice macro features and parallelability.

A port of dmake for Windows is available from:

```
http://www.cpan.org/authors/id/GSAR/dmake-4.1pl1-win32.zip
```

(This is a fixed version of the original dmake sources obtained from http://www.wticorp.com/ As of version 4.1PL1, the original sources did not build as shipped and had various other problems. A patch is included in the above fixed version.)

Fetch and install dmake somewhere on your path (follow the instructions in the README.NOW file).

There exists a minor coexistence problem with dmake and Borland C++ compilers. Namely, if a distribution has C files named with mixed case letters, they will be compiled into appropriate .obj-files named with all lowercase letters, and every time dmake is invoked to bring files up to date, it will try to recompile such files again. For example, Tk distribution has a lot of such files, resulting in needless recompiles every time dmake is invoked. To avoid this, you may use the script "sync_ext.pl" after a successful build. It is available in the win32 subdirectory of the Perl source distribution.

Command Shell

Use the default "cmd" shell that comes with NT. Some versions of the popular 4DOS/NT shell have incompatibilities that may cause you trouble. If the build fails under that shell, try building again with the cmd shell.

The nmake Makefile also has known incompatibilities with the "command.com" shell that comes with Windows 9x. You will need to use dmake and makefile.mk to build under Windows 9x.

The surest way to build it is on Windows NT/2000/XP, using the cmd shell.

Make sure the path to the build directory does not contain spaces. The build usually works in this circumstance, but some tests will fail.

Borland C++

If you are using the Borland compiler, you will need dmake. (The make that Borland supplies is seriously crippled and will not work for MakeMaker builds.)

See §**??** above.

Microsoft Visual C++

The nmake that comes with Visual C++ will suffice for building. You will need to run the VCVARS32.BAT file, usually found somewhere like C:\MSDEV4.2\BIN. This will set your build environment.

You can also use dmake to build using Visual C++; provided, however, you set OSRELEASE to "microsft" (or whatever the directory name under which the Visual C dmake configuration lives) in your environment and edit win32/config.vc to change "make=nmake" into "make=dmake". The latter step is only essential if you want to use dmake as your default make for building extensions using MakeMaker.

Microsoft Platform SDK 64-bit Compiler

The nmake that comes with the Platform SDK will suffice for building Perl. Make sure you are building within one of the "Build Environment" shells available after you install the Platform SDK from the Start Menu.

MinGW release 3 with gcc

The latest release of MinGW at the time of writing is 3.1.0, which comes with gcc-3.2.3, and can be downloaded here:

```
http://www.mingw.org/
```

Perl also compiles with earlier releases of gcc (2.95.2 and up). See below for notes about using earlier versions of MinGW/gcc.

You also need dmake. See §**??** above on how to get it.

MinGW release 1 with gcc

The MinGW-1.1 bundle comes with gcc-2.95.3.

Make sure you install the binaries that work with MSVCRT.DLL as indicated in the README for the GCC bundle. You may need to set up a few environment variables (usually ran from a batch file).

There are a couple of problems with the version of gcc-2.95.2-msvcrt.exe released 7 November 1999:

- It left out a fix for certain command line quotes. To fix this, be sure to download and install the file fixes/quote-fix-msvcrt.exe from the above ftp location.
- The definition of the fpos_t type in stdio.h may be wrong. If your stdio.h has this problem, you will see an exception when running the test t/lib/io_xs.t. To fix this, change the typedef for fpos_t from "long" to "long long" in the file i386-mingw32msvc/include/stdio.h, and rebuild.

A potentially simpler to install (but probably soon-to-be-outdated) bundle of the above package with the mentioned fixes already applied is available here:

```
http://downloads.ActiveState.com/pub/staff/gsar/gcc-2.95.2-msvcrt.zip
ftp://ftp.ActiveState.com/pub/staff/gsar/gcc-2.95.2-msvcrt.zip
```

127.2.2 Building

- Make sure you are in the "win32" subdirectory under the perl toplevel. This directory contains a "Makefile" that will work with versions of nmake that come with Visual C++ or the Platform SDK, and a dmake "makefile.mk" that will work for all supported compilers. The defaults in the dmake makefile are setup to build using Microsoft Visual C++ 6.0 or newer.
- Edit the makefile.mk (or Makefile, if you're using nmake) and change the values of INST_DRV and INST_TOP. You can also enable various build flags. These are explained in the makefiles.

  Note that it is generally not a good idea to try to build a perl with INST_DRV and INST_TOP set to a path that already exists from a previous build. In particular, this may cause problems with the lib/ExtUtils/t/Embed.t test, which attempts to build a test program and may end up building against the installed perl's lib/CORE directory rather than the one being tested.

  You will have to make sure that CCTYPE is set correctly and that CCHOME points to wherever you installed your compiler.

  The default value for CCHOME in the makefiles for Visual C++ may not be correct for some versions. Make sure the default exists and is valid.

  If you have either the source or a library that contains des_fcrypt(), enable the appropriate option in the makefile. A ready-to-use version of fcrypt.c, based on the version originally written by Eric Young at ftp://ftp.funet.fi/pub/crypt/mirrors/dsi/libdes/, is bundled with the distribution. Set CRYPT_SRC to fcrypt.c to use this version. Alternatively, if you have built a library that contains des_fcrypt(), you can set CRYPT_LIB to point to the library name. Perl will also build without des_fcrypt(), but the crypt() builtin will fail at run time.

  Be sure to read the instructions near the top of the makefiles carefully.
- Type "dmake" (or "nmake" if you are using that make).

  This should build everything. Specifically, it will create perl.exe, perl58.dll at the perl toplevel, and various other extension dll's under the lib\auto directory. If the build fails for any reason, make sure you have done the previous steps correctly.

127.2.3 Testing Perl on Win32

Type "dmake test" (or "nmake test"). This will run most of the tests from the testsuite (many tests will be skipped).

There should be no test failures when running under Windows NT/2000/XP. Many tests *will* fail under Windows 9x due to the inferior command shell.

Some test failures may occur if you use a command shell other than the native "cmd.exe", or if you are building from a path that contains spaces. So don't do that.

If you are running the tests from a emacs shell window, you may see failures in op/stat.t. Run "dmake test-notty" in that case.

If you're using the Borland compiler, you may see a failure in op/taint.t arising from the inability to find the Borland Runtime DLLs on the system default path. You will need to copy the DLLs reported by the messages from where Borland chose to install it, into the Windows system directory (usually somewhere like C:\WINNT\SYSTEM32) and rerun the test.

If you're using Borland compiler versions 5.2 and below, you may run into problems finding the correct header files when building extensions. For example, building the "Tk" extension may fail because both perl and Tk contain a header file called "patchlevel.h". The latest Borland compiler (v5.5) is free of this misbehaviour, and it even supports an option -VI- for backward (bugward) compatibility for using the old Borland search algorithm to locate header files.

If you run the tests on a FAT partition, you may see some failures for `link()` related tests:

```
    Failed Test                    Stat Wstat Total Fail  Failed  List

    ../ext/IO/lib/IO/t/io_dup.t                   6    4  66.67%  2-5
    ../lib/File/Temp/t/mktemp.t                   9    1  11.11%  2
    ../lib/File/Temp/t/posix.t                    7    1  14.29%  3
    ../lib/File/Temp/t/security.t                13    1   7.69%  2
    ../lib/File/Temp/t/tempfile.t                20    2  10.00%  2 4
    comp/multiline.t                              6    2  33.33%  5-6
    io/dup.t                                      8    6  75.00%  2-7
    op/write.t                                   47    7  14.89%  1-3 6 9-11
```

Testing on NTFS avoids these errors.

Furthermore, you should make sure that during `make test` you do not have any GNU tool packages in your path: some toolkits like Unixutils include some tools (`type` for instance) which override the Windows ones and makes tests fail. Remove them from your path while testing to avoid these errors.

Please report any other failures as described under BUGS AND CAVEATS.

127.2.4 Installation of Perl on Win32

Type "dmake install" (or "nmake install"). This will put the newly built perl and the libraries under whatever `INST_TOP` points to in the Makefile. It will also install the pod documentation under `$INST_TOP\$VERSION\lib\pod` and HTML versions of the same under `$INST_TOP\$VERSION\lib\pod\html`. To use the Perl you just installed, you will need to add two components to your PATH environment variable, `$INST_TOP\$VERSION\bin` and `$INST_TOP\$VERSION\bin\$ARCHNAME`. For example:

```
    set PATH c:\perl\5.6.0\bin;c:\perl\5.6.0\bin\MSWin32-x86;%PATH%
```

If you opt to comment out INST_VER and INST_ARCH in the makefiles, the installation structure is much simpler. In that case, it will be sufficient to add a single entry to the path, for instance:

```
    set PATH c:\perl\bin;%PATH%
```

127.2.5 Usage Hints for Perl on Win32

Environment Variables

The installation paths that you set during the build get compiled into perl, so you don't have to do anything additional to start using that perl (except add its location to your PATH variable).

If you put extensions in unusual places, you can set PERL5LIB to a list of paths separated by semicolons where you want perl to look for libraries. Look for descriptions of other environment variables you can set in *perlrun*.

You can also control the shell that perl uses to run system() and backtick commands via PERL5SHELL. See *perlrun*.

Perl does not depend on the registry, but it can look up certain default values if you choose to put them there. Perl attempts to read entries from `HKEY_CURRENT_USER\Software\Perl` and `HKEY_LOCAL_MACHINE\Software\Perl`. Entries in the former override entries in the latter. One or more of the following entries (of type REG_SZ or REG_EXPAND_SZ) may be set:

```
    lib-$]              version-specific standard library path to add to @INC
    lib                 standard library path to add to @INC
    sitelib-$]          version-specific site library path to add to @INC
    sitelib             site library path to add to @INC
    vendorlib-$]        version-specific vendor library path to add to @INC
    vendorlib           vendor library path to add to @INC
    PERL*               fallback for all %ENV lookups that begin with "PERL"
```

Note the `$]` in the above is not literal. Substitute whatever version of perl you want to honor that entry, e.g. `5.6.0`. Paths must be separated with semicolons, as usual on win32.

File Globbing

By default, perl handles file globbing using the File::Glob extension, which provides portable globbing.

If you want perl to use globbing that emulates the quirks of DOS filename conventions, you might want to consider using File::DosGlob to override the internal glob() implementation. See *File::DosGlob* for details.

Using perl from the command line

If you are accustomed to using perl from various command-line shells found in UNIX environments, you will be less than pleased with what Windows offers by way of a command shell.

The crucial thing to understand about the Windows environment is that the command line you type in is processed twice before Perl sees it. First, your command shell (usually CMD.EXE on Windows NT, and COMMAND.COM on Windows 9x) preprocesses the command line, to handle redirection, environment variable expansion, and location of the executable to run. Then, the perl executable splits the remaining command line into individual arguments, using the C runtime library upon which Perl was built.

It is particularly important to note that neither the shell nor the C runtime do any wildcard expansions of command-line arguments (so wildcards need not be quoted). Also, the quoting behaviours of the shell and the C runtime are rudimentary at best (and may, if you are using a non-standard shell, be inconsistent). The only (useful) quote character is the double quote ("). It can be used to protect spaces and other special characters in arguments.

The Windows NT documentation has almost no description of how the quoting rules are implemented, but here are some general observations based on experiments: The C runtime breaks arguments at spaces and passes them to programs in argc/argv. Double quotes can be used to prevent arguments with spaces in them from being split up. You can put a double quote in an argument by escaping it with a backslash and enclosing the whole argument within double quotes. The backslash and the pair of double quotes surrounding the argument will be stripped by the C runtime.

The file redirection characters "<", ">", and "|" can be quoted by double quotes (although there are suggestions that this may not always be true). Single quotes are not treated as quotes by the shell or the C runtime, they don't get stripped by the shell (just to make this type of quoting completely useless). The caret "ˆ" has also been observed to behave as a quoting character, but this appears to be a shell feature, and the caret is not stripped from the command line, so Perl still sees it (and the C runtime phase does not treat the caret as a quote character).

Here are some examples of usage of the "cmd" shell:

This prints two doublequotes:

```
perl -e "print '\"\"' "
```

This does the same:

```
perl -e "print \"\\\"\\\"\" "
```

This prints "bar" and writes "foo" to the file "blurch":

```
perl -e "print 'foo'; print STDERR 'bar'" > blurch
```

This prints "foo" ("bar" disappears into nowhereland):

```
perl -e "print 'foo'; print STDERR 'bar'" 2> nul
```

This prints "bar" and writes "foo" into the file "blurch":

```
perl -e "print 'foo'; print STDERR 'bar'" 1> blurch
```

This pipes "foo" to the "less" pager and prints "bar" on the console:

```
perl -e "print 'foo'; print STDERR 'bar'" | less
```

This pipes "foo\nbar\n" to the less pager:

```
perl -le "print 'foo'; print STDERR 'bar'" 2>&1 | less
```

This pipes "foo" to the pager and writes "bar" in the file "blurch":

```
perl -e "print 'foo'; print STDERR 'bar'" 2> blurch | less
```

Discovering the usefulness of the "command.com" shell on Windows 9x is left as an exercise to the reader :)

One particularly pernicious problem with the 4NT command shell for Windows NT is that it (nearly) always treats a % character as indicating that environment variable expansion is needed. Under this shell, it is therefore important to always double any % characters which you want Perl to see (for example, for hash variables), even when they are quoted.

Building Extensions

The Comprehensive Perl Archive Network (CPAN) offers a wealth of extensions, some of which require a C compiler to build. Look in http://www.cpan.org/ for more information on CPAN.

Note that not all of the extensions available from CPAN may work in the Win32 environment; you should check the information at http://testers.cpan.org/ before investing too much effort into porting modules that don't readily build.

Most extensions (whether they require a C compiler or not) can be built, tested and installed with the standard mantra:

```
perl Makefile.PL
$MAKE
$MAKE test
$MAKE install
```

where $MAKE is whatever ’make’ program you have configured perl to use. Use "perl -V:make" to find out what this is. Some extensions may not provide a testsuite (so "$MAKE test" may not do anything or fail), but most serious ones do.

It is important that you use a supported ’make’ program, and ensure Config.pm knows about it. If you don’t have nmake, you can either get dmake from the location mentioned earlier or get an old version of nmake reportedly available from:

```
 http://download.microsoft.com/download/vc15/Patch/1.52/W95/EN-US/nmake15.exe
```

Another option is to use the make written in Perl, available from CPAN.

```
    http://www.cpan.org/modules/by-module/Make/
```

You may also use dmake. See §**??** above on how to get it.

Note that MakeMaker actually emits makefiles with different syntax depending on what ’make’ it thinks you are using. Therefore, it is important that one of the following values appears in Config.pm:

```
    make=’nmake’        # MakeMaker emits nmake syntax
    make=’dmake’        # MakeMaker emits dmake syntax
    any other value     # MakeMaker emits generic make syntax
                            (e.g GNU make, or Perl make)
```

If the value doesn’t match the ’make’ program you want to use, edit Config.pm to fix it.

If a module implements XSUBs, you will need one of the supported C compilers. You must make sure you have set up the environment for the compiler for command-line compilation.

If a module does not build for some reason, look carefully for why it failed, and report problems to the module author. If it looks like the extension building support is at fault, report that with full details of how the build failed using the perlbug utility.

Command-line Wildcard Expansion

The default command shells on DOS descendant operating systems (such as they are) usually do not expand wildcard arguments supplied to programs. They consider it the application’s job to handle that. This is commonly achieved by linking the application (in our case, perl) with startup code that the C runtime libraries usually provide. However, doing that results in incompatible perl versions (since the behavior of the argv expansion code differs depending on the compiler, and it is even buggy on some compilers). Besides, it may be a source of frustration if you use such a perl binary with an alternate shell that *does* expand wildcards.

Instead, the following solution works rather well. The nice things about it are 1) you can start using it right away; 2) it is more powerful, because it will do the right thing with a pattern like */*/*.c; 3) you can decide whether you do/don’t want to use it; and 4) you can extend the method to add any customizations (or even entirely different kinds of wildcard expansion).

```
        C:\> copy con c:\perl\lib\Wild.pm
        # Wild.pm - emulate shell @ARGV expansion on shells that don’t
        use File::DosGlob;
        @ARGV = map {
                      my @g = File::DosGlob::glob($_) if /[*?]/;
                      @g ? @g : $_;
                    } @ARGV;
        1;
        ^Z
        C:\> set PERL5OPT=-MWild
        C:\> perl -le "for (@ARGV) { print }" */*/perl*.c
        p4view/perl/perl.c
        p4view/perl/perlio.c
```

```
p4view/perl/perly.c
perl5.005/win32/perlglob.c
perl5.005/win32/perllib.c
perl5.005/win32/perlglob.c
perl5.005/win32/perllib.c
perl5.005/win32/perlglob.c
perl5.005/win32/perllib.c
```

Note there are two distinct steps there: 1) You'll have to create Wild.pm and put it in your perl lib directory. 2) You'll need to set the PERL5OPT environment variable. If you want argv expansion to be the default, just set PERL5OPT in your default startup environment.

If you are using the Visual C compiler, you can get the C runtime's command line wildcard expansion built into perl binary. The resulting binary will always expand unquoted command lines, which may not be what you want if you use a shell that does that for you. The expansion done is also somewhat less powerful than the approach suggested above.

Win32 Specific Extensions

A number of extensions specific to the Win32 platform are available from CPAN. You may find that many of these extensions are meant to be used under the Activeware port of Perl, which used to be the only native port for the Win32 platform. Since the Activeware port does not have adequate support for Perl's extension building tools, these extensions typically do not support those tools either and, therefore, cannot be built using the generic steps shown in the previous section.

To ensure smooth transitioning of existing code that uses the ActiveState port, there is a bundle of Win32 extensions that contains all of the ActiveState extensions and most other Win32 extensions from CPAN in source form, along with many added bugfixes, and with MakeMaker support. This bundle is available at:

```
http://www.cpan.org/authors/id/GSAR/libwin32-0.18.zip
```

See the README in that distribution for building and installation instructions. Look for later versions that may be available at the same location.

Notes on 64-bit Windows

Windows .NET Server supports the LLP64 data model on the Intel Itanium architecture.

The LLP64 data model is different from the LP64 data model that is the norm on 64-bit Unix platforms. In the former, `int` and `long` are both 32-bit data types, while pointers are 64 bits wide. In addition, there is a separate 64-bit wide integral type, `__int64`. In contrast, the LP64 data model that is pervasive on Unix platforms provides `int` as the 32-bit type, while both the `long` type and pointers are of 64-bit precision. Note that both models provide for 64-bits of addressability.

64-bit Windows running on Itanium is capable of running 32-bit x86 binaries transparently. This means that you could use a 32-bit build of Perl on a 64-bit system. Given this, why would one want to build a 64-bit build of Perl? Here are some reasons why you would bother:

*
A 64-bit native application will run much more efficiently on Itanium hardware.

*
There is no 2GB limit on process size.

*
Perl automatically provides large file support when built under 64-bit Windows.

*
Embedding Perl inside a 64-bit application.

127.2.6 Running Perl Scripts

Perl scripts on UNIX use the "#!" (a.k.a "shebang") line to indicate to the OS that it should execute the file using perl. Win32 has no comparable means to indicate arbitrary files are executables.

Instead, all available methods to execute plain text files on Win32 rely on the file "extension". There are three methods to use this to execute perl scripts:

1. There is a facility called "file extension associations" that will work in Windows NT 4.0. This can be manipulated via the two commands "assoc" and "ftype" that come standard with Windows NT 4.0. Type "ftype /?" for a complete example of how to set this up for perl scripts (Say what? You thought Windows NT wasn't perl-ready? :).

2. Since file associations don't work everywhere, and there are reportedly bugs with file associations where it does work, the old method of wrapping the perl script to make it look like a regular batch file to the OS, may be used. The install process makes available the "pl2bat.bat" script which can be used to wrap perl scripts into batch files. For example:

   ```
   pl2bat foo.pl
   ```

   will create the file "FOO.BAT". Note "pl2bat" strips any .pl suffix and adds a .bat suffix to the generated file.

   If you use the 4DOS/NT or similar command shell, note that "pl2bat" uses the "%*" variable in the generated batch file to refer to all the command line arguments, so you may need to make sure that construct works in batch files. As of this writing, 4DOS/NT users will need a "ParameterChar = *" statement in their 4NT.INI file or will need to execute "setdos /p*" in the 4DOS/NT startup file to enable this to work.

3. Using "pl2bat" has a few problems: the file name gets changed, so scripts that rely on `$0` to find what they must do may not run properly; running "pl2bat" replicates the contents of the original script, and so this process can be maintenance intensive if the originals get updated often. A different approach that avoids both problems is possible.

   A script called "runperl.bat" is available that can be copied to any filename (along with the .bat suffix). For example, if you call it "foo.bat", it will run the file "foo" when it is executed. Since you can run batch files on Win32 platforms simply by typing the name (without the extension), this effectively runs the file "foo", when you type either "foo" or "foo.bat". With this method, "foo.bat" can even be in a different location than the file "foo", as long as "foo" is available somewhere on the PATH. If your scripts are on a filesystem that allows symbolic links, you can even avoid copying "runperl.bat".

   Here's a diversion: copy "runperl.bat" to "runperl", and type "runperl". Explain the observed behavior, or lack thereof. :) Hint: .gnidnats llits er'uoy fi ,"lrepnur" eteled :tniH

4. Miscellaneous Things

   A full set of HTML documentation is installed, so you should be able to use it if you have a web browser installed on your system.

   `perldoc` is also a useful tool for browsing information contained in the documentation, especially in conjunction with a pager like `less` (recent versions of which have Win32 support). You may have to set the PAGER environment variable to use a specific pager. "perldoc -f foo" will print information about the perl operator "foo".

   One common mistake when using this port with a GUI library like `Tk` is assuming that Perl's normal behavior of opening a command-line window will go away. This isn't the case. If you want to start a copy of `perl` without opening a command-line window, use the `wperl` executable built during the installation process. Usage is exactly the same as normal `perl` on Win32, except that options like `-h` don't work (since they need a command-line window to print to).

   If you find bugs in perl, you can run `perlbug` to create a bug report (you may have to send it manually if `perlbug` cannot find a mailer on your system).

127.3 BUGS AND CAVEATS

Norton AntiVirus interferes with the build process, particularly if set to "AutoProtect, All Files, when Opened". Unlike large applications the perl build process opens and modifies a lot of files. Having the the AntiVirus scan each and every one slows build the process significantly. Worse, with PERLIO=stdio the build process fails with peculiar messages as the virus checker interacts badly with miniperl.exe writing configure files (it seems to either catch file part written and treat it as suspicious, or virus checker may have it "locked" in a way which inhibits miniperl updating it). The build does complete with

```
    set PERLIO=perlio
```

but that may be just luck. Other AntiVirus software may have similar issues.

Some of the built-in functions do not act exactly as documented in *perlfunc*, and a few are not implemented at all. To avoid surprises, particularly if you have had prior exposure to Perl in other operating environments or if you intend to write code that will be portable to other environments. See *perlport* for a reasonably definitive list of these differences.

Not all extensions available from CPAN may build or work properly in the Win32 environment. See §**??**.

Most `socket()` related calls are supported, but they may not behave as on Unix platforms. See *perlport* for the full list.

Signal handling may not behave as on Unix platforms (where it doesn't exactly "behave", either :). For instance, calling `die()` or `exit()` from signal handlers will cause an exception, since most implementations of `signal()` on Win32 are severely crippled. Thus, signals may work only for simple things like setting a flag variable in the handler. Using signals under this port should currently be considered unsupported.

Please send detailed descriptions of any problems and solutions that you may find to <*perlbug@perl.com*>, along with the output produced by `perl -V`.

127.4 ACKNOWLEDGEMENTS

The use of a camel with the topic of Perl is a trademark of O'Reilly and Associates, Inc. Used with permission.

127.5 AUTHORS

Gary Ng <71564.1743@CompuServe.COM>

Gurusamy Sarathy <gsar@activestate.com>

Nick Ing-Simmons <nick@ing-simmons.net>

This document is maintained by Gurusamy Sarathy.

127.6 SEE ALSO

perl

127.7 HISTORY

This port was originally contributed by Gary Ng around 5.003_24, and borrowed from the Hip Communications port that was available at the time. Various people have made numerous and sundry hacks since then.

Borland support was added in 5.004_01 (Gurusamy Sarathy).

GCC/mingw32 support was added in 5.005 (Nick Ing-Simmons).

Support for PERL_OBJECT was added in 5.005 (ActiveState Tool Corp).

Support for fork() emulation was added in 5.6 (ActiveState Tool Corp).

Win9x support was added in 5.6 (Benjamin Stuhl).

Support for 64-bit Windows added in 5.8 (ActiveState Corp).

Last updated: 20 April 2002

Contents