Perl Programmers Reference Guide

Perl Version 5.004 BETA
23−Mar−1997

"There's more than one way to do it."

−− Larry Wall, Author of the Perl Programming Language

Author: Perl5−Porters@perl.org

blank

NAME

perl − Practical Extraction and Report Language

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*]...

For ease of access, the Perl manual has been split up into a number of sections:

```
    perl        Perl overview (this section)
    perldelta   Perl changes since previous version
    perlfaq     Perl frequently asked questions

    perldata    Perl data structures
    perlsyn     Perl syntax
    perlop      Perl operators and precedence
    perlre      Perl regular expressions
    perlrun     Perl execution and options
    perlfunc    Perl builtin functions
    perlvar     Perl predefined variables
    perlsub     Perl subroutines
    perlmod     Perl modules
    perlform    Perl formats
    perllocale  Perl locale support

    perlref     Perl references
    perldsc     Perl data structures intro
    perllol     Perl data structures: lists of lists
    perltoot    Perl OO tutorial
    perlobj     Perl objects
    perltie     Perl objects hidden behind simple variables
    perlbot     Perl OO tricks and examples
    perlipc     Perl interprocess communication

    perldebug   Perl debugging
    perldiag    Perl diagnostic messages
    perlsec     Perl security
    perltrap    Perl traps for the unwary
    perlstyle   Perl style guide

    perlpod     Perl plain old documentation
    perlbook    Perl book information

    perlembed   Perl how to embed perl in your C or C++ app
    perlapio    Perl internal IO abstraction interface
    perlxs      Perl XS application programming interface
    perlxstut   Perl XS tutorial
    perlguts    Perl internal functions for those doing extensions
    perlcall    Perl calling conventions from C
```

(If you‘re intending to read these straight through for the first time, the suggested order will tend to reduce the number of forward references.)

Additional documentation for Perl modules is available in the ***/usr/local/man/*** directory. Some of this is distributed standard with Perl, but you‘ll also find third−party modules there. You should be able to view this with your man(1) program by including the proper directories in the appropriate start−up files. To find out where these are, type:

```
    perl −V:man.dir
```

If the directories were ***/usr/local/man/man1*** and ***/usr/local/man/man3***, you would need to add only ***/usr/local/man*** to your MANPATH. If they are different, 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.

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 quite 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 (previously called "associative arrays") grow as necessary to prevent degraded performance. Perl uses sophisticated pattern matching techniques to scan large amounts of data very 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 which 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...

Perl version 5 is nearly a complete rewrite, and provides the following additional benefits:

- Many usability enhancements

  It is now possible to write much more readable Perl code (even within regular expressions). Formerly cryptic variable names can be replaced by mnemonic identifiers. Error messages are more informative, and the optional warnings will catch many of the mistakes a novice might make. This cannot be stressed enough. Whenever you get mysterious behavior, try the **−w** switch!!! Whenever you don‘t get mysterious behavior, try using **−w** anyway.

- Simplified grammar

  The new yacc grammar is one half the size of the old one. Many of the arbitrary grammar rules have been regularized. The number of reserved words has been cut by 2/3. Despite this, nearly all old Perl scripts will continue to work unchanged.

- Lexical scoping

  Perl variables may now be declared within a lexical scope, like "auto" variables in C. Not only is this more efficient, but it contributes to better privacy for "programming in the large". Anonymous subroutines exhibit deep binding of lexical variables (closures).

- Arbitrarily nested data structures

  Any scalar value, including any array element, may now contain a reference to any other variable or subroutine. You can easily create anonymous variables and subroutines. Perl manages your reference counts for you.

- Modularity and reusability

  The Perl library is now defined in terms of modules which can be easily shared among various packages. A package may choose to import all or a portion of a module‘s published interface. Pragmas (that is, compiler directives) are defined and used by the same mechanism.

- Object−oriented programming

  A package can function as a class. Dynamic multiple inheritance and virtual methods are supported in a straightforward manner and with very little new syntax. Filehandles may now be treated as objects.

- Embeddable and Extensible

  Perl may now be embedded easily in your C or C++ application, and can either call or be called by your routines through a documented interface. The XS preprocessor is provided to make it easy to glue your C or C++ routines into Perl. Dynamic loading of modules is supported, and Perl itself can be made into a dynamic library.

- POSIX compliant

  A major new module is the POSIX module, which provides access to all available POSIX routines and definitions, via object classes where appropriate.

- Package constructors and destructors

  The new BEGIN and END blocks provide means to capture control as a package is being compiled, and after the program exits. As a degenerate case they work just like awk‘s BEGIN and END when you use the **−p** or **−n** switches.

- Multiple simultaneous DBM implementations

  A Perl program may now access DBM, NDBM, SDBM, GDBM, and Berkeley DB files from the same script simultaneously. In fact, the old dbmopen interface has been generalized to allow any variable to be tied to an object class which defines its access methods.

- Subroutine definitions may now be autoloaded

  In fact, the AUTOLOAD mechanism also allows you to define any arbitrary semantics for undefined subroutine calls. It‘s not for just autoloading.

- Regular expression enhancements

  You can now specify non−greedy quantifiers. You can now do grouping without creating a backreference. You can now write regular expressions with embedded whitespace and comments for readability. A consistent extensibility mechanism has been added that is upwardly compatible with all old regular expressions.

- Innumerable Unbundled Modules

  The Comprehensive Perl Archive Network described in *perlmod* contains hundreds of plug−and−play modules full of reusable code. See ***http://www.perl.com/CPAN*** for a site near you.

- Compilability

  While not yet in full production mode, a working perl−to−C compiler does exist. It can generate portable bytecode, simple C, or optimized C code.

Okay, that‘s *definitely* enough hype.

ENVIRONMENT

See *perlrun*.

AUTHOR

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

FILES

```
 "/tmp/perl−e$$"        temporary file for −e commands
 "@INC"                 locations of perl libraries
```

SEE ALSO

```
 a2p    awk to perl translator

 s2p    sed to perl translator
```

DIAGNOSTICS

The **−w** switch produces some lovely diagnostics.

See *perldiag* for explanations of all Perl‘s diagnostics.

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 the case of 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?

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 `sprintf()`. The latter can even trigger a core dump when passed ludicrous input values.

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 255 characters, and no component of your PATH may be longer than 255 if you use **−S**. A regular expression may not compile to more than 32767 bytes internally.

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.com***. 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.

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.

NAME

perldelta − what‘s new for perl5.004

DESCRIPTION

This document describes differences between the 5.003 release (as documented in *Programming Perl*, second edition—the Camel Book) and this one.

Supported Environments

Perl5.004 builds out of the box on Unix, Plan9, LynxOS, VMS, OS/2, QNX, and AmigaOS.

Core Changes

Most importantly, many bugs were fixed. See the ***Changes*** file in the distribution for details.

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.

No Autovivification of Subroutine Parameters

In Perl versions 5.002 and 5.003, array and hash elements used as subroutine parameters were "autovivified"; that is, they were brought into existence if they did not already exist. For example, calling `func($h{foo})` would create `$h{foo}` if it did not already exist, causing `exists $h{foo}` to become true and `keys %h` to return `(‘foo’)`.

Perl 5.004 returns to the pre−5.002 behavior of *not* autovivifying array and hash elements used as subroutine parameters.

Fixed Parsing of `$$<digit`, `&$<digit`, etc.

A bug in previous versions of Perl 5.0 prevented proper parsing of numeric special variables as symbolic references. That bug has been fixed. As a result, the string "`$$0`" is no longer equivalent to `$$."0",` but rather to `${$0}.` To get the old behavior, change "`$$`" followed by a digit to "`${$}".`

No Resetting of `$.` on Implicit Close

The documentation for Perl 5.0 has always stated that `$.` is *not* reset when an already−open file handle is re−opened with no intervening call to `close`. Due to a bug, perl versions 5.000 through 5.0003 *did* reset `$.` under that circumstance; Perl 5.004 does not.

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.

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.

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.

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 a backward−compatible synonym for `*STDOUT{IO}`.

Internal Change: PerlIO internal IO 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.

New and Changed Built−in Variables

`$^E` Extended error message on some platforms. (Also known as `$EXTENDED_OS_ERROR` if you `use English`).

`$^H` 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.

`$^M` By default, running out of memory it is not trappable. However, if compiled for this, Perl may use the contents of `$^M` as an emergency pool after `die()`ing with this message. Suppose that your Perl were compiled with −DEMERGENCY_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.

New and Changed Built−in 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

now support "%i" as a synonym for "%d", and the "h" modifier. So "%hi" means "short integer in decimal", and "%ho" means "unsigned short integer as octal".

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.

`unpack()` and `pack()`

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.

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 run−time 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//g` does not trigger a `pos()` reset on failure

The `m//g` match iteration construct used to reset the iteration when it failed to match (so that the next `m//g` match would start at the beginning of the string). You now have to explicitly do a `pos $str = 0;` to reset the "last match" position, or modify the string in some way. This change makes it practical to chain `m//g` matches together in conjunction with ordinary matches using the `\G` zero−width assertion. See *perlop* and *perlre*.

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, and is fine now:

```
my $i;
foreach $i ( 1 .. 10 ) {
    format =
    my i is @#
    $i
.
    write;
}
```

New Built−in 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.

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} @_), $\;
    }
```

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";
            }
```

Malloc Enhancements

Four new compilation flags are recognized by malloc.c. (They have no effect if perl is compiled with system `malloc()`.)

−DDEBUGGING_MSTATS

If perl is compiled with `DEBUGGING_MSTATS` defined, 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 ares printed only on exit. (If you want the statistics at an arbitrary time, you‘ll need to install the optional module Devel::Peek.)

−DEMERGENCY_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 `$^M`. See *"$^M"*.

−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.

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.

Pragmata

Four new pragmatic modules exist:

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 locale

Tells the compiler to enable (or disable) the use of POSIX locales for built−in 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 formating 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.

Modules

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.

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'`).

Module Information Summary

Brand new modules, arranged by topic rather than strictly alphabetically:

```
CPAN                  interface to Comprehensive Perl Archive Network
CPAN::FirstTime       create a CPAN configuration file
CPAN::Nox             run 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/Template.pm	Structure/member template builder
File/stat.pm	Object−oriented wrapper around CORE::stat
Net/hostent.pm	Object−oriented wrapper around CORE::gethost*
Net/netent.pm	Object−oriented wrapper around CORE::getnet*
Net/protoent.pm	Object−oriented wrapper around CORE::getproto*
Net/servent.pm	Object−oriented wrapper around CORE::getserv*
Time/gmtime.pm	Object−oriented wrapper around CORE::gmtime
Time/localtime.pm	Object−oriented wrapper around CORE::localtime
Time/tm.pm	Perl implementation of "struct tm" for {gm,local}time
User/grent.pm	Object−oriented wrapper around CORE::getgr*
User/pwent.pm	Object−oriented wrapper around CORE::getpw*
Tie/RefHash.pm	Base class for tied hashes with references as keys
UNIVERSAL.pm	Base class for *ALL* classes

IO

The IO module provides a simple mechanism to load all of 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.

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 cbrt root
tan cotan asin acos atan acotan
sinh cosh tanh cotanh asinh acosh atanh acotanh
cplx cplxe
```

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.

Net::Ping

Major rewrite − support added for both udp echo and real icmp pings.

Overridden Built−ins

Many of the Perl built−ins 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);
```

Utility Changes

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 *`.

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).

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 (non−tied 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.

Documentation Changes

Many of the base and library pods were updated. These new pods are included in section 1:

perldelta

This document.

perllocale

Locale support (internationalization and localization).

perltoot

Tutorial on Perl OO programming.

perlapio

Perl internal IO abstraction interface.

perldebug

Although not new, this has been massively updated.

perlsec

Although not new, this has been massively updated.

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 (non−trappable).
(A) An alien error message (not generated by Perl).
```

"my" variable %s masks earlier declaration in same scope

(S) 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 MSDOS machine.

Allocation too large

(F) You can't allocate more than 2^31+"small amount" bytes.

Attempt to free non−existent 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*.

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.

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 non−standard names, or it may indicate that a logical name table has been corrupted.

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*.

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.

Died

(F) You passed `die()` an empty string (the equivalent of `die ""`) or you called it with no args and both $@ and $_ were empty.

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.

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.

Stub found while resolving method '%s' overloading '%s' in package '%s'

(P) Overloading resolution over @ISA tree may be broken by importing stubs. Stubs should never be implicitely created, but explicit calls to `can` may break this.

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).

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.

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 exclamation marks 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.

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.

Value of %s construct can be "0"; test with `defined()`

(W) In a conditional expression, you used <HANDLE, <* (glob), 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 re−bound 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.

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 non−empty 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*.

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*.

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 *Process terminated by SIGTERM/SIGINT in perlos2*.

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.

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.

HISTORY

Constructed by Tom Christiansen, grabbing material with permission from innumerable contributors, with kibitzing by more than a few Perl porters.

Last update: Sat Mar 8 19:51:26 EST 1997

NAME

perlfaq − frequently asked questions about Perl (`$Date:` 1997/03/17 22:17:56 `$`)

DESCRIPTION

This document is structured into the following sections:

perlfaq: Structural overview of the FAQ.

This document.

perlfaq1: General Questions About Perl

Very general, high−level information about Perl.

perlfaq2: Obtaining and Learning about Perl

Where to find source and documentation to Perl, support and training, and related matters.

perlfaq3: Programming Tools

Programmer tools and programming support.

perlfaq4: Data Manipulation

Manipulating numbers, dates, strings, arrays, hashes, and miscellaneous data issues.

perlfaq5: Files and Formats

I/O and the "f" issues: filehandles, flushing, formats and footers.

perlfaq6: Regexps

Pattern matching and regular expressions.

perlfaq7: General Perl Language Issues

General Perl language issues that don‘t clearly fit into any of the other sections.

perlfaq8: System Interaction

Interprocess communication (IPC), control over the user−interface (keyboard, screen and pointing devices).

perlfaq9: Networking

Networking, the Internet, and a few on the web.

Where to get this document

This document is posted regularly to comp.lang.perl.announce and several other related newsgroups. It is available in a variety of formats from CPAN in the /CPAN/doc/FAQs/FAQ/ directory, or on the web at http://www.perl.com/perl/faq/ .

How to contribute to this document

You may mail corrections, additions, and suggestions to perlfaq−suggestions@perl.com. Mail sent to the old perlfaq alias will merely cause the FAQ to be sent to you.

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−suggestions@perl.com 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.

Credits

When I first began the Perl FAQ in the late 80s, I never realized it would have grown to over a hundred pages, nor that Perl would ever become so popular and widespread. This document could not have been written without the tremendous help provided by Larry Wall and the rest of the Perl Porters.

Author and Copyright Information

Non−commercial Reproduction

Commercial Reproduction

Disclaimer

Changes

17/March/97 Version

Various typos fixed throughout.

Added new question on Perl BNF on *perlfaq7*.

Initial Release: 11/March/97

This is the initial release of version 3 of the FAQ; consequently there have been no changes since its initial release.

NAME

perlfaq1 − General Questions About Perl (`$Revision:` 1.10 `$)`

DESCRIPTION

This section of the FAQ answers very general, high−level questions about Perl.

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.

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.

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 news://genetics.upenn.edu/perl.porters−gw/ and http://www.frii.com/~gnat/perl/porters/summary.html.

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.

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. The last production release was 5.003, and the current experimental release for those at the bleeding edge (as of 27/03/97) is 5.003_92, considered a beta for production release 5.004, which will probably be out by the time you read this. Further references to the Perl language in this document refer to the current production release unless otherwise specified.

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 complete rewrite of the perl source code from the ground up. 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.

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.

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).

Is Perl difficult to learn?

Perl is easy to start learning — and easy to keep learning. It looks like most programming languages you‘re likely to have had experience with, so if you‘ve ever written an C program, an awk script, a shell script, or even an Excel macro, you‘re already part way 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, Perl is (frequently) an interpreted language. This means that 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 the CPAN, which is discussed in Part 2.

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.

Can I do [task] in Perl?

Perl is flexible and extensible enough for you to use on almost any task, from one−line file−processing tasks to complex 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.

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.

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 multithreaded 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 reduce the limitations given in the previous statement to some degree, but understand that Perl remains fundamentally a dynamically typed language, and not a statically typed one. You certainly won‘t be chastized 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. :−)

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.

Is it a Perl program or a Perl script?

It doesn‘t matter.

In "standard terminology" a *program* has been compiled to physical machine code once, and can then be be run multiple times, whereas a *script* must be translated by a program each time it‘s used. Perl programs, however, are usually neither strictly compiled nor strictly interpreted. They can be compiled to a bytecode form (something of a Perl virtual machine) or to completely different languages, like C or assembly language. You can‘t tell just by looking 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.

What is a JAPH?

These are the "just another perl hacker" signatures that some people sign their postings with. About 100 of the of the earlier ones are available from http://www.perl.com/CPAN/misc/japh .

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.perl.com/CPAN/misc/lwall−quotes .

How can I convince my sysadmin/supervisor/employees to use version (5/5.004/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, as 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, and 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/or hardware companies throughout the world. In fact, many Unix vendors now ship Perl by default, and support is usually just a news−posting away, if you can‘t find the answer in the *comprehensive* documentation, including this FAQ.

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 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. That probably means running the 5.004 release, although 5.003 isn‘t that bad (it‘s just one year and one release behind). Several important bugs were fixed from the 5.000 through 5.002 versions, though, so try upgrading past them if possible.

AUTHOR AND COPYRIGHT

NAME

perlfaq2 − Obtaining and Learning about Perl (`$Revision:` 1.13 `$)`

DESCRIPTION

This section of the FAQ answers questions about where to find source and documentation for Perl, support and training, and related matters.

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.perl.com/CPAN/src/latest.tar.gz, which is a gzipped archive in POSIX tar format. This source builds with no porting whatsoever on most Unix systems (Perl‘s native environment), as well as Plan 9, VMS, QNX, OS/2, and the Amiga.

Although it‘s rumored that the (imminent) 5.004 release may build on Windows NT, this is yet to be proven. Binary distributions for 32−bit Microsoft systems and for Apple systems can be found
http://www.perl.com/CPAN/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).

A useful FAQ for Win32 Perl users is
http://www.endcontsw.com/people/evangelo/Perl_for_Win32_FAQ.html

How can I get a binary version of Perl?

If you don‘t have a C compiler because for whatever reasons your vendor 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.

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 which perl is looking for.

```
        perl −e ’print join("\n",@INC)’
```

If this command lists any paths which don‘t exist on your system, then you may need to move the appropriate libraries to these locations, or create symlinks, aliases, or shortcuts appropriately.

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 idiosyncracies that the Configure script can‘t work around for any given system or architecture.

What modules and extensions are available for Perl? What is CPAN? What does CPAN/src/... mean?

CPAN stands for Comprehensive Perl Archive Network, a huge archive replicated on dozens of 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 machine for CPAN is
ftp://ftp.funet.fi/pub/languages/perl/CPAN/, but you can use the address
http://www.perl.com/CPAN/CPAN.html to fetch a copy from a "site near you". See
http://www.perl.com/CPAN (without a slash at the end) for how this process works.

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 file is downloadable as ftp://ftp.funet.fi/pub/languages/perl/CPAN/misc/japh .

Considering that there are hundreds of 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.

Is there an ISO or ANSI certified version of Perl?

Certainly not. Larry expects that he'll be certified before Perl is.

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. If you're not on a Unix system, access to the documentation will be different; for example, it might be only in HTML format. But 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 the CPAN/doc directory, which contains the complete documentation in various formats, including native pod, troff, html, and plain text. There's also a web page at http://www.perl.com/perl/info/documentation.html that might help.

It's also worth noting that there's a PDF version of the complete documentation for perl available in the CPAN/authors/id/BMIDD directory.

Many good books have been written about Perl — see the section below for more details.

What are the Perl newsgroups on USENET? Where do I post questions?

The now defunct comp.lang.perl newsgroup has been superseded by the following groups:

```
    comp.lang.perl.announce            Moderated announcement group
    comp.lang.perl.misc                Very busy group about Perl in general
    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.
```

There is also USENET gateway to the mailing list used by the crack Perl development team (perl5−porters) at news://genetics.upenn.edu/perl.porters−gw/ .

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 for details.

Perl Books

A number 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 and his apostles, is now in its second edition and fourth printing.

```
    Programming Perl (the "Camel Book"):
        Authors: Larry Wall, Tom Christiansen, and Randal Schwartz
        ISBN 1-56592-149-6      (English)
        ISBN 4-89052-384-7      (Japanese)
        (French and German translations in progress)
```

Note that O'Reilly books are color−coded: turquoise (some would call it teal) covers indicate perl5 coverage, while magenta (some would call it pink) covers indicate perl4 only. Check the cover color before you buy!

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.

If you're already a hard−core systems programmer, then the Camel Book just might suffice for you to learn Perl from. But if you're not, check out the "Llama Book". It currently doesn't cover perl5, but the 2nd edition is nearly done and should be out by summer 97:

```
    Learning Perl (the Llama Book):
        Author: Randal Schwartz, with intro by Larry Wall
        ISBN 1-56592-042-2      (English)
        ISBN 4-89502-678-1      (Japanese)
        ISBN 2-84177-005-2      (French)
        ISBN 3-930673-08-8      (German)
```

Another stand−out book in the turquoise O'Reilly Perl line is the "Hip Owls" book. It covers regular expressions inside and out, with quite a bit devoted exclusively to Perl:

```
    Mastering Regular Expressions (the Cute Owls Book):
        Author: Jeffrey Friedl
        ISBN 1-56592-257-3
```

You can order any of these 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. See http://www.ora.com/ on the Web.

Recommended Perl books that are not from O'Reilly are the following:

```
   Cross-Platform Perl, (for Unix and Windows NT)
       Author: Eric F. Johnson
       ISBN: 1-55851-483-X

   How to Set up and Maintain a World Wide Web Site, (2nd edition)
        Author: Lincoln Stein, M.D., Ph.D.
        ISBN: 0-201-63462-7

   CGI Programming in C & Perl,
        Author: Thomas Boutell
        ISBN: 0-201-42219-0
```

Note that some of these address specific application areas (e.g. the Web) and are not general−purpose programming books.

Perl in Magazines

The Perl Journal is the first and only magazine dedicated to Perl. It is published (on paper, not online) quarterly by Jon Orwant (orwant@tpj.com), editor. Subscription information is at http://tpj.com or via email to subscriptions@tpj.com.

Beyond this, two other magazines that frequently carry high−quality articles on Perl are Web Techniques (see http://www.webtechniques.com/) and Unix Review (http://www.unixreview.com/).

Perl on the Net: FTP and WWW Access

To get the best (and possibly cheapest) performance, pick a site from the list below and use it to grab the complete list of mirror sites. From there you can find the quickest site for you. Remember, the following list is *not* the complete list of CPAN mirrors.

```
http://www.perl.com/CPAN   (redirects to another mirror)
http://www.perl.org/CPAN
ftp://ftp.funet.fi/pub/languages/perl/CPAN/
http://www.cs.ruu.nl/pub/PERL/CPAN/
ftp://ftp.cs.colorado.edu/pub/perl/CPAN/
```

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. The following are a list of mailing lists related to perl itself.

If you subscribe to a mailing list, it behooves you to know how to unsubscribe from it. Strident pleas to the list itself to get you off will not be favorably received.

MacPerl

There is a mailing list for discussing Macintosh Perl. Contact "mac−perl−request@iis.ee.ethz.ch".

Also see Matthias Neeracher‘s (the creator and maintainer of MacPerl) webpage at http://www.iis.ee.ethz.ch/~neeri/macintosh/perl.html for many links to interesting MacPerl sites, and the applications/MPW tools, precompiled.

Perl5−Porters

The core development team have a mailing list for discussing fixes and changes to the language. Send mail to "perl5−porters−request@perl.org" with help in the body of the message for information on subscribing.

NTPerl

This list is used to discuss issues involving Win32 Perl 5 (Windows NT and Win95). Subscribe by emailing ListManager@ActiveWare.com with the message body:

```
subscribe Perl−Win32−Users
```

The list software, also written in perl, will automatically determine your address, and subscribe you automatically. To unsubscribe, email the following in the message body to the same address like so:

```
unsubscribe Perl−Win32−Users
```

You can also check http://www.activeware.com/ and select "Mailing Lists" to join or leave this list.

Perl−Packrats

Discussion related to archiving of perl materials, particularly the Comprehensive PerlArchive Network (CPAN). Subscribe by emailing majordomo@cis.ufl.edu:

```
subscribe perl−packrats
```

The list software, also written in perl, will automatically determine your address, and subscribe you automatically. To unsubscribe, simple prepend the same command with an "un", and mail to the same address like so:

```
unsubscribe perl−packrats
```

Archives of comp.lang.perl.misc

Have you tried Deja News or Alta Vista?

ftp.cis.ufl.edu:/pub/perl/comp.lang.perl.*/monthly has an almost complete collection dating back to 12/89 (missing 08/91 through 12/93). They are kept as one large file for each month.

You‘ll probably want more a sophisticated query and retrieval mechanism than a file listing, preferably one that allows you to retrieve articles using a fast−access indices, keyed on at least author, date, subject, thread (as in "trn") and probably keywords. The best solution the FAQ authors know of is the MH pick command, but it is very slow to select on 18000 articles.

If you have, or know where can be found, the missing sections, please let perlfaq−suggestions@perl.com know.

Perl Training

While some large training companies offer their own courses on Perl, you may prefer to contact individuals near and dear to the heart of Perl development. Two well−known members of the Perl development team who offer such things are Tom Christiansen <perl−classes@perl.com and Randal Schwartz <perl−training−info@stonehenge.com, plus their respective minions, who offer a variety of professional tutorials and seminars on Perl. These courses include large public seminars, private corporate training, and fly−ins to Colorado and Oregon. See http://www.perl.com/perl/info/training.html for more details.

Where can I buy a commercial version of Perl?

In a sense, Perl already *is* commercial software: It has a licence 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, dozens of software designers and developers, and thousands of 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 wrong. 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.

Or you can purchase a real support contract. Although Cygnus historically provided this service, they no longer sell support contracts for Perl. Instead, the Paul Ingram Group will be taking up the slack through The Perl Clinic. The following is a commercial from them:

"Do you need professional support for Perl and/or Oraperl? Do you need a support contract with defined levels of service? Do you want to pay only for what you need?

"The Paul Ingram Group has provided quality software development and support services to some of the world‘s largest corporations for ten years. We are now offering the same quality support services for Perl at The Perl Clinic. This service is led by Tim Bunce, an active perl porter since 1994 and well known as the author and maintainer of the DBI, DBD::Oracle, and Oraperl modules and author/co−maintainer of The Perl 5 Module List. We also offer Oracle users support for Perl5 Oraperl and related modules (which Oracle is planning to ship as part of Oracle Web Server 3). 20% of the profit from our Perl support work will be donated to The Perl Institute."

For more information, contact the The Perl Clinic:

```
    Tel:    +44 1483 424424
    Fax:    +44 1483 419419
    Web:    http://www.perl.co.uk/
    Email:  perl−support−info@perl.co.uk or Tim.Bunce@ig.co.uk
```

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 email your report to perlbug@perl.com.

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 man page (perl5.004 or later) for more information.

What is perl.com? perl.org? The Perl Institute?

perl.org is the official vehicle for The Perl Institute. The motto of TPI is "helping people help Perl help people" (or something like that). It‘s a non−profit organization supporting development, documentation, and dissemination of perl. Current directors of TPI include Larry Wall, Tom Christiansen, and Randal Schwartz, whom you may have heard of somewhere else around here.

The perl.com domain is Tom Christiansen‘s domain. He created it as a public service long before perl.org came about. It‘s the original PBS of the Perl world, a clearinghouse for information about all things Perlian, accepting no paid advertisements, glossy gifs, or (gasp!) java applets on its pages.

How do I learn about object−oriented Perl programming?

perltoot (distributed with 5.004 or later) is a good place to start. Also, *perlobj*, *perlref*, and *perlmod* are useful references, while *perlbot* has some excellent tips and tricks.

AUTHOR AND COPYRIGHT

Copyright (c) 1997 Tom Christiansen and Nathan Torkington. All rights reserved. See *perlfaq* for distribution information.

NAME

perlfaq3 − Programming Tools (`$Revision:` 1.19 `$)`

DESCRIPTION

This section of the FAQ answers questions related to programmer tools and programming support.

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 man pages? Here‘s a brief index:

```
Objects         perlref, perlmod, perlobj, perltie
Data Structures perlref, perllol, perldsc
Modules         perlmod, perlsub
Regexps         perlre, perlfunc, perlop
Moving to perl5 perltrap, perl
Linking w/C     perlxstut, perlxs, perlcall, perlguts, perlembed
Various         http://www.perl.com/CPAN/doc/FMTEYEWTK/index.html
                (not a man−page but still useful)
```

perltoc provides a crude table of contents for the perl man page set.

How can I use Perl interactively?

The typical approach uses the Perl debugger, described in the perldebug(1) man page, 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

Is there a Perl shell?

In general, no. 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.

How do I debug my Perl programs?

Have you used `−w`?

Have you tried `use strict`?

Did you check the returns of each and every system call?

Did you read *perltrap*?

Have you tried the Perl debugger, described in *perldebug*?

How do I profile my Perl programs?

You should get the Devel::DProf module from CPAN, and also use Benchmark.pm from the standard distribution. Benchmark lets you time specific portions of your code, while Devel::DProf gives detailed breakdowns of where your code spends its time.

How do I cross−reference my Perl programs?

The B::Xref module, shipped with the new, alpha−release Perl compiler (not the general distribution), can be used to generate cross−reference reports for Perl programs.

```
perl −MO=Xref[,OPTIONS] foo.pl
```

Is there a pretty−printer (formatter) for Perl?

There is no program that will reformat Perl as much as indent(1) will do for C. The complex feedback between the scanner and the parser (this feedback is what confuses the vgrind and emacs programs) makes it challenging at best to write a stand−alone Perl parser.

Of course, if you simply follow the guidelines in *perlstyle*, you shouldn‘t need to reformat.

Your editor can and should help you with source formatting. The perl−mode for emacs can provide a remarkable amount of help with most (but not all) code, and even less programmable editors can provide significant assistance.

If you are using to using vgrind program for printing out nice code to a laser printer, you can take a stab at this using http://www.perl.com/CPAN/doc/misc/tips/working.vgrind.entry, but the results are not particularly satisfying for sophisticated code.

Is there a ctags for Perl?

There‘s a simple one at http://www.perl.com/CPAN/authors/id/TOMC/scripts/ptags.gz which may do the trick.

Where can I get Perl macros for vi?

For a complete version of Tom Christiansen‘s vi configuration file, see ftp://ftp.perl.com/pub/vi/toms.exrc, the standard benchmark file for vi emulators. This 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.perl.com/CPAN/src/misc .

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 hilighting. You should be using "main::foo", anyway.

How can I use curses with Perl?

The Curses module from CPAN provides a dynamically loadable object module interface to a curses library.

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.

How can I generate simple menus without using CGI or Tk?

The http://www.perl.com/CPAN/authors/id/SKUNZ/perlmenu.v4.0.tar.gz module, which is curses−based, can help with this.

Can I dynamically load C routines into Perl?

If your system architecture supports it, then the standard perl on your system should also provide you with this via the DynaLoader module. Read *perlxstut* for details.

What is undump?

See the next questions.

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. Chapter 8 in the Camel has some efficiency tips in it you might want to look at.

Other approaches include autoloading 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 is the use of modules that have critical sections written in C (for instance, the PDL module from CPAN).

In some cases, it may be worth it to use the backend compiler to produce byte code (saving compilation time) or compile into C, which will certainly save compilation time and sometimes a small amount (but not much) execution time. See the question about compiling your Perl programs.

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.

Unsubstantiated reports allege that Perl interpreters that use sfio outperform those that don‘t (for IO intensive applications). To try this, see the ***INSTALL*** file in the source distribution, especially the "Selecting File IO mechanisms" section.

The undump program was an old attempt to speed up your 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.

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 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 for 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 built−in 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`.

Is it unsafe to return a pointer to local data?

No, Perl‘s garbage collection system takes care of this.

```
    sub makeone {
        my @a = ( 1 .. 10 );
        return \@a;
    }

    for $i ( 1 .. 10 ) {
        push @many, makeone();
    }

    print $many[4][5], "\n";

    print "@many\n";
```

How can I free an array or hash so my program shrinks?

You can‘t. Memory the system allocates to a program will never be returned to the system. That‘s why long−running programs sometimes re−exec themselves.

However, judicious use of `my()` on your variables will help make sure that they go out of scope so that Perl can free up their storage for use in other parts of your program. (NB: `my()` variables also execute about 10% faster than globals.) 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.

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 at least 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::* modules (from CPAN), 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. With the FCGI module (from CPAN), a Perl executable compiled with sfio (see the ***INSTALL*** file in the distribution) and the mod_fastcgi module (available from http://www.fastcgi.com/) each of your perl scripts becomes a permanent CGI daemon processes.

Both of these solutions can have far−reaching effects on your system and on the way you write your CGI scripts, so investigate them with care.

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.) 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 (Filter::* from CPAN). 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." We are not lawyers, of course, so you should see a lawyer if you want to be sure your licence‘s wording will stand up in court.

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 as of Feb−1997 in late alpha release, which means it‘s fun to play with if you‘re a programmer but not really for people looking for turn−key solutions.

Please understand that 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 thus will still 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 (like several times faster), but this takes some tweaking of your code.

Malcolm will be in charge of the 5.005 release of Perl itself to try to unify and merge his compiler and multithreading work into the main release.

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 miniscule. For example, on one author‘s system, /usr/bin/perl is only 11k in size!

How can I get ‘#!perl’ to work on [MSDOS,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 `ALTERNATIVE_SHEBANG` (see the ***INSTALL*** file in the source distribution for more information).

The Win95/NT installation, when using the Activeware port of Perl, will modify the Registry to associate the .pl extension with the perl interpreter. If you install another port, or (eventually) build your own Win95/NT Perl using WinGCC, then you‘ll have to modify the Registry yourself.

Macintosh perl scripts will have the 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 scripts working for a web server. This is an EXTREMELY big security risk. Take the time to figure out how to do it correctly.

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 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 entry. :−)

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)

    # VMS
    perl -e "print ""Hello world\n"""
```

The problem is that none of this is reliable: it depends on the command interpreter. Under Unix, the first two often work. Under DOS, it‘s entirely possible neither works. If 4DOS was the command shell, I‘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.

I‘m afraid that there is no general solution to all of this. It is a mess, pure and simple.

[Some of this answer was contributed by Kenneth Albanowski.]

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 these sources:

```
    The Idiot’s Guide to Solving Perl/CGI Problems, by Tom Christiansen
        http://www.perl.com/perl/faq/idiots-guide.html

    Frequently Asked Questions about CGI Programming, by Nick Kew
        ftp://rtfm.mit.edu/pub/usenet/news.answers/www/cgi-faq
        http://www3.pair.com/webthing/docs/cgi/faqs/cgifaq.shtml

    Perl/CGI programming FAQ, by Shishir Gundavaram and Tom Christiansen
        http://www.perl.com/perl/faq/perl-cgi-faq.html

    The WWW Security FAQ, by Lincoln Stein
        http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html

    World Wide Web FAQ, by Thomas Boutell
        http://www.boutell.com/faq/
```

Where can I learn about object−oriented Perl programming?

perltoot is a good place to start, and you can use *perlobj* and *perlbot* for reference. Perltoot didn‘t come out until the 5.004 release, but you can get a copy (in pod, html, or postscript) from http://www.perl.com/CPAN/doc/FMTEYEWTK/ .

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.

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 bugreport with the output of `make test TEST_VERBOSE=1` along with `perl -V`.

When I tried to run my script, I got this message. What does it mean?

perldiag has a complete list of perl‘s error messages and warnings, with explanatory text. 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;
```

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*.

AUTHOR AND COPYRIGHT

NAME

perlfaq4 − Data Manipulation (`$Revision:` 1.15 `$`)

DESCRIPTION

The section of the FAQ answers question related to the manipulation of data as numbers, dates, strings, arrays, hashes, and miscellaneous data issues.

Data: Numbers

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. 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. `oct()` interprets both hex ("0x350") numbers and octal ones ("0350" or even without the leading "0", like "377"), while `hex()` only converts hexadecimal ones, with or without a leading "0x", like "0x255", "3A", "ff", or "deadbeef".

This problem shows up most often when people try using `chmod()`, `mkdir()`, `umask()`, or `sysopen()`, which all want permissions in octal.

```
chmod(644,  $file); # WRONG -- perl -w catches this
chmod(0644, $file); # right
```

Does perl have a round function? What about `ceil()` and `floor()`?

Trig functions?

For rounding to a certain number of digits, `sprintf()` or `printf()` is usually the easiest route.

The POSIX module (part of the standard perl distribution) implements `ceil()`, `floor()`, and a number of other mathematical and trigonometric functions.

The Math::Complex module (part of the standard perl distribution) defines a number of mathematical functions that can also work on real numbers. It‘s not as efficient as the POSIX library, but the POSIX library 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.

How do I convert bits into ints?

To turn a string of 1s and 0s like ‘10110110’ into a scalar containing its binary value, use the `pack()` function (documented in *pack in perlfunc*):

```
$decimal = pack('B8', '10110110');
```

Here‘s an example of going the other way:

```
$binary_string = join('', unpack('B*', "\x29"));
```

How do I multiply matrices?

Use the Math::Matrix or Math::MatrixReal modules (available from CPAN) or the PDL extension (also available from CPAN).

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 $iterator (@array) {
        &my_func($iterator);
    }
```

To call a function on each integer in a (small) range, you **can** use:

```
    @results = map { &my_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; $i++) {
        push(@results, &my_func($i));
    }
```

How can I output Roman numerals?

Get the http://www.perl.com/CPAN/modules/by−module/Roman module.

Why aren‘t my random numbers random?

The short explanation is that you‘re getting pseudorandom numbers, not random ones, because that‘s how these things work. A longer explanation is available on
http://www.perl.com/CPAN/doc/FMTEYEWTK/random, courtesy of Tom Phoenix.

You should also check out the Math::TrulyRandom module from CPAN.

Data: Dates

How do I find the week−of−the−year/day−of−the−year?

The day of the year is in the array returned by `localtime()` (see *localtime in perlfunc*):

```
    $day_of_year = (localtime(time()))[7];
```

or more legibly (in 5.004 or higher):

```
    use Time::localtime;
    $day_of_year = localtime(time())->yday;
```

You can find the week of the year by dividing this by 7:

```
    $week_of_year = int($day_of_year / 7);
```

Of course, this believes that weeks start at zero.

How can I compare two date strings?

Use the Date::Manip or Date::DateCalc modules from CPAN.

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 one of the Date modules from CPAN.

How can I find the Julian Day?

Neither Date::Manip nor Date::DateCalc deal with Julian days. Instead, there is an example of Julian date calculation in http://www.perl.com/CPAN/authors/David_Muir_Sharnoff/modules/Time/JulianDay.pm.gz, which should help.

Does Perl have a year 2000 problem?

Not unless you use Perl to create one. 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). The year returned by these functions when used in an array 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 a 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.

Data: Strings

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, email addresses, etc.) for details.

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;
```

Note that this won't expand \n or \t or any other special escapes.

How do I remove consecutive pairs of characters?

To turn "abbcccd" into "abccd":

```
    s/(.)\1/$1/g;
```

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 a list context) into a string:

```
    print "My sub returned @{[mysub(1,2,3)]} that time.\n";
```

If you prefer scalar context, similar chicanery is also useful for arbitrary expressions:

```
    print "That yields ${\($n + 5)} widgets\n";
```

How do I find matching/nesting anything?

This isn't something that can be tackled 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, nor can they. For that you'll have to write a parser.

How do I reverse a string?

Use `reverse()` in a scalar context, as documented in *reverse*.

```
    $reversed = reverse $string;
```

How do I expand tabs in a string?

You can do it the old−fashioned way:

```
    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);
```

How do I reformat a paragraph?

Use Text::Wrap (part of the standard perl distribution):

```
    use Text::Wrap;
    print wrap("\t", '  ', @paragraphs);
```

How can I access/change the first N letters of a string?

There are many ways. If you just want to grab a copy, use substr:

```
    $first_byte = substr($a, 0, 1);
```

If you want to modify part of a string, the simplest way is often to use `substr()` as an lvalue:

```
    substr($a, 0, 3) = "Tom";
```

Although those with a regexp kind of thought process will likely prefer

```
    $a =~ s/^.../Tom/;
```

How do I change the Nth occurrence of something?

You have to keep track. For example, let‘s say you want to change the fifth occurrence of "whoever" or "whomever" into "whosoever", case insensitively.

```
    $count = 0;
    s{((whom?)ever)}{
        ++$count == 5           # is it the 5th?
            ? "${2}soever"      # yes, swap
            : $1                # renege and leave it there
    }igex;
```

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 charcters 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";
```

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;`

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;`

How can I split a [character] delimited string except when inside

[character]? (Comma−separated files)

Take the example case of trying to split a string that is comma−separated into its different fields. (We‘ll pretend you said comma−separated, not comma−delimited, which is different and almost never what you mean.) 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 a highly recommended book on 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 ’,’;
```

Alternatively, the Text::ParseWords module (part of the standard perl distribution) lets you say:

```
    use Text::ParseWords;
    @new = quotewords(",", 0, $text);
```

How do I strip blank space from the beginning/end of a string?

The simplest approach, albeit not the fastest, is probably like this:

```
    $string =~ s/^\s*(.*?)\s*$/$1/;
```

It would be faster to do this in two steps:

```
    $string =~ s/^\s+//;
    $string =~ s/\s+$//;
```

Or more nicely written as:

```
    for ($string) {
        s/^\s+//;
        s/\s+$//;
    }
```

How do I extract selected columns from a string?

Use `substr()` or `unpack()`, both documented in *perlfunc*.

How do I find the soundex value of a string?

Use the standard Text::Soundex module distributed with perl.

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’;
    $text =~ s/\$(\w+)/${$1}/g;
```

Before version 5 of perl, this had to be done with a double−eval substitution:

```
    $text =~ s/(\$\w+)/$1/eeg;
```

Which is bizarre enough that you‘ll probably actually need an EEG afterwards. :−)

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.

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.

Why don't my <<HERE documents work?

Check for these three things:

1. There must be no space after the << part.
2. There (probably) should be a semicolon at the end.
3. You can't (easily) have any space in front of the tag.

Data: Arrays

What is the difference between `$array[1]` and @array[1]?

The former is a scalar value, the latter an array slice, which makes 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 **−w** flag will warn you about these matters.

How can I extract just the unique elements of an 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:

```
        $prev = 'nonesuch';
        @out = grep($_ ne $prev && ($prev = $_), @in);
```

This is nice in that it doesn't use much extra memory, simulating uniq(1)'s behavior of removing only adjacent duplicates.

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 = @ary;
```

How can I tell whether an array contains a certain element?

There are several ways to approach this. If you are going to make this query many times and the values are arbitrary strings, the fastest way is probably to invert the original array and keep an associative array lying about whose keys are the first array‘s values.

```
    @blues = qw/azure cerulean teal turquoise lapis-lazuli/;
    undef %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);
    undef @is_tiny_prime;
    for (@primes) { $is_tiny_prime[$_] = 1; }
```

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 $read;
    grep (vec($read,$_,1) = 1, @articles);
```

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 regexp characters in `$whatever?).`

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;
    }
```

How do I find the first array element for which a condition is true?

You can use this if you care about the index:

```
    for ($i=0; $i < @array; $i++) {
        if ($array[$i] eq "Waldo") {
            $found_index = $i;
            last;
        }
    }
```

Now `$found_index` has what you want.

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.

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.

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
```

How do I shuffle an array randomly?

Here's a shuffling algorithm which works its way through the list, 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));
}
```

For large arrays, this avoids a lot of the reshuffling:

```
srand;
@new = ();
@old = 1 .. 10000;  # just a demo
for( @old ){
    my $r = rand @new+1;
    push(@new,$new[$r]);
    $new[$r] = $_;
}
```

How do I process/modify each element of an array?

Use `for`/`foreach`:

```
for (@lines) {
    s/foo/bar/;
    tr[a-z][A-Z];
}
```

Here's another; let's compute spherical volumes:

```
for (@radii) {
    $_ **= 3;
    $_ *= (4/3) * 3.14159;  # this will be constant folded
}
```

How do I select a random element from an array?

Use the `rand()` function (see *rand*):

```
srand;                     # not needed for 5.004 and later
$index   = rand @array;
$element = $array[$index];
```

How do I permute N elements of a list?

Here‘s a little program that generates all permutations of all the words on each line of input. The algorithm embodied in the `permut()` function should work on any list:

```
#!/usr/bin/perl −n
# permute − tchrist@perl.com
permut([split], []);
sub permut {
    my @head = @{ $_[0] };
    my @tail = @{ $_[1] };
    unless (@head) {
        # stop recursing when there are no elements in the head
        print "@tail\n";
    } else {
        # for all elements in @head, move one from @head to @tail
        # and call permut() on the new @head and @tail
        my(@newhead,@newtail,$i);
        foreach $i (0 .. $#head) {
            @newhead = @head;
            @newtail = @tail;
            unshift(@newtail, splice(@newhead, $i, 1));
            permut([@newhead], [@newtail]);
        }
    }
}
```

How do I sort an array by (anything)?

Supply a comparison function to `sort()` (described in *sort*):

```
@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 http://www.perl.com/CPAN/doc/FMTEYEWTK/sort.html for more about this approach.

See also the question below on sorting hashes.

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 }
```

And 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.)

Why does `defined()` return true on empty arrays and hashes?

See *defined* in the 5.004 release or later of Perl.

Data: Hashes (Associative Arrays)

How do I process an entire hash?

Use the `each()` function (see *each*) 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.

What happens if I add or remove keys from a hash while iterating over it?

Don't do that.

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.

How can I know how many entries are in a hash?

If you mean how many keys, then all you have to do is take the scalar sense of the `keys()` function:

```
    $num_keys = scalar keys %hash;
```

In void context it just resets the iterator, which is faster for tied hashes.

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, and 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;
```

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 *In Memory Databases in DB_File*.

What's the difference between "delete" and "undef" with hashes?

Hashes are 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 `$key` is present in the array, `exists($key)` will return true. The value for a given key can be `undef`, in which case `$array{$key}` will be `undef` while `$exists{$key}` will return true. This corresponds to (`$key, undef`) being in the hash.

Pictures help... here's the `%ary` table:

```
          keys  values
        +------+------+
        |  a   |  3   |
        |  x   |  7   |
        |  d   |  0   |
        |  e   |  2   |
        +------+------+
```

And these conditions hold

```
        $ary{'a'}                       is true
        $ary{'d'}                       is false
        defined $ary{'d'}               is true
        defined $ary{'a'}               is true
        exists $ary{'a'}                is true (perl5 only)
        grep ($_ eq 'a', keys %ary)     is true
```

If you now say

```
        undef $ary{'a'}
```

your table now reads:

```
          keys  values
        +------+------+
        |  a   | undef|
        |  x   |  7   |
        |  d   |  0   |
        |  e   |  2   |
        +------+------+
```

and these conditions now hold; changes in caps:

```
        $ary{'a'}                       is FALSE
        $ary{'d'}                       is false
        defined $ary{'d'}               is true
        defined $ary{'a'}               is FALSE
        exists $ary{'a'}                is true (perl5 only)
        grep ($_ eq 'a', keys %ary)     is true
```

Notice the last two: you have an undef value, but a defined key!

Now, consider this:

```
        delete $ary{'a'}
```

your table now reads:

```
          keys  values
        +------+------+
        |  x   |  7   |
        |  d   |  0   |
        |  e   |  2   |
        +------+------+
```

and these conditions now hold; changes in caps:

```
        $ary{'a'}                       is false
        $ary{'d'}                       is false
        defined $ary{'d'}               is true
        defined $ary{'a'}               is false
        exists $ary{'a'}                is FALSE (perl5 only)
        grep ($_ eq 'a', keys %ary)     is FALSE
```

See, the whole entry is gone!

Why don‘t my tied hashes make the defined/exists distinction?

They may or may not implement the `EXISTS()` and `DEFINED()` methods differently. For example, there isn‘t the concept of undef with hashes that are tied to DBM* files. This means the true/false tables above will give different results when used on such a hash. 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.

How do I reset an each() operation part−way through?

Using `keys %hash` in a 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.

How can I get the unique keys from two hashes?

First you extract the keys from the hashes into arrays, and then solve the uniquifying the array 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;
```

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.

How can I make my hash remember the order I put elements into it?

Use the Tie::IxHash from CPAN.

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.

How can I make the Perl equivalent of a C structure/C++ class/hash

or array of hashes or arrays?

Use references (documented in *perlref*). Examples of complex data structures are given in *perldsc* and *perllol*. Examples of structures and object−oriented classes are in *perltoot*.

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.

Data: Misc

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 some systems, however, you have to play tedious games with "text" versus "binary" files. See *binmode in perlfunc*.

If you‘re concerned about 8−bit ASCII data, then see *perllocale*.

If you want to deal with multi−byte characters, however, there are some gotchas. See the section on Regular Expressions.

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.

```
   warn "has nondigits"        if     /\D/;
   warn "not a whole number"   unless /^\d+$/;
   warn "not an integer"       unless /^−?\d+$/;  # reject +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+)$/;
   warn "not a C float"
       unless /^([+−]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+−]?\d+))?$/;
```

Or you could check out http://www.perl.com/CPAN/modules/by−module/String/String−Scanf−1.1.tar.gz instead. The POSIX module (part of the standard Perl distribution) provides the `strtol` and `strtod` for converting strings to double and longs, respectively.

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, Storable, or Class::Eroot modules from CPAN.

How do I print out or copy a recursive data structure?

The Data::Dumper module on CPAN is nice for printing out data structures, and FreezeThaw for copying them. For example:

```
    use FreezeThaw qw(freeze thaw);
    $new = thaw freeze $old;
```

Where `$old` can be (a reference to) any kind of data structure you‘d like. It will be deeply copied.

How do I define methods for every class/object?

Use the UNIVERSAL class (see *UNIVERSAL*).

How do I verify a credit card checksum?

Get the Business::CreditCard module from CPAN.

AUTHOR AND COPYRIGHT

NAME

perlfaq5 − Files and Formats (`$Revision:` 1.19 `$)`

DESCRIPTION

This section deals with I/O and the "f" issues: filehandles, flushing, formats, and footers.

How do I flush/unbuffer a filehandle? Why must I do this?

The C standard I/O library (stdio) normally buffers characters sent to devices. This is done for efficiency reasons, so that there isn‘t a system call for each byte. Any time you use `print()` or `write()` in Perl, you go though this buffering. `syswrite()` circumvents stdio and buffering.

In most stdio implementations, the type of buffering and the size of the buffer varies according to the type of device. Disk files are block buffered, often with a buffer size of more than 2k. Pipes and sockets are often buffered with a buffer size between 1/2 and 2k. Serial devices (e.g. modems, terminals) are normally line−buffered, and stdio sends the entire line when it gets the newline.

Perl does not support truly unbuffered output (except insofar as you can `syswrite(OUT, $char, 1)`). What it does instead support is "command buffering", in which a physical write is performed after every output command. This isn‘t as hard on your system as unbuffering, but does get the output where you want it when you want it.

If you expect characters to get to your device when you print them there, you‘ll want to autoflush its handle, as in the older:

```
use FileHandle;
open(DEV, "<+/dev/tty");      # ceci n’est pas une pipe
DEV−>autoflush(1);
```

or the newer IO::* modules:

```
use IO::Handle;
open(DEV, ">/dev/printer");   # but is this?
DEV−>autoflush(1);
```

or even this:

```
use IO::Socket;               # this one is kinda a pipe?
$sock = IO::Socket::INET−>new(PeerAddr => ’www.perl.com’,
                              PeerPort => ’http(80)’,
                              Proto    => ’tcp’);
die "$!" unless $sock;

$sock−>autoflush();
$sock−>print("GET /\015\012");
$document = join(’’, $sock−>getlines());
print "DOC IS: $document\n";
```

Note the hardcoded carriage return and newline in their octal equivalents. This is the ONLY way (currently) to assure a proper flush on all platforms, including Macintosh.

You can use `select()` and the `$|` variable to control autoflushing (see *$|* and *select*):

```
$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]);
```

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?

Although humans have an easy time thinking of a text file as being a sequence of lines that operates much like a stack of playing cards — or punch cards — computers usually see the text file as a sequence of bytes. In general, there‘s no direct way for Perl to seek to a particular line of a file, insert text into a file, or remove text from a file.

(There are exceptions in special circumstances. Replacing a sequence of bytes with another sequence of the same length is one. Another is using the `$DB_RECNO` array bindings as documented in *DB_File*. Yet another is manipulating files with all lines the same length.)

The general solution is to create a temporary copy of the text file with the changes you want, then copy that over the original.

```
$old = $file;
$new = "$file.tmp.$$";
$bak = "$file.bak";

open(OLD, "< $old")         or die "can't open $old: $!";
open(NEW, "> $new")         or die "can't open $new: $!";

# Correct typos, preserving case
while (<OLD>) {
    s/\b(p)earl\b/${1}erl/i;
    (print NEW $_)          or die "can't write to $new: $!";
}

close(OLD)                  or die "can't close $old: $!";
close(NEW)                  or die "can't close $new: $!";

rename($old, $bak)          or die "can't rename $old to $bak: $!";
rename($new, $old)          or die "can't rename $new to $old: $!";
```

Perl can do this sort of thing for you automatically with the `−i` command−line switch or the closely−related `$^I` variable (see *perlrun* for more details). Note that `−i` may require a suffix on some non−Unix systems; see the platform−specific documentation that came with your port.

```
# Renumber a series of tests from the command line
perl −pi −e 's/(^\s+test\s+)\d+/ $1 . ++$count /e' t/op/taint.t

# form a script
local($^I, @ARGV) = ('.bak', glob("*.c"));
while (<>) {
    if ($. == 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 $.
}
```

If you need to seek to an arbitrary line of a file that changes infrequently, you could build up an index of byte positions of where the line ends are in the file. If the file is large, an index of every tenth or hundredth line end would allow you to seek and read fairly efficiently. If the file is sorted, try the look.pl library (part of the standard perl distribution).

In the unique case of deleting lines at the end of a file, you can use `tell()` and `truncate()`. The following code snippet deletes the last line of a file without making a copy or reading the whole file into memory:

```
        open (FH, "+< $file");
        while ( <FH> ) { $addr = tell(FH) unless eof(FH) }
        truncate(FH, $addr);
```

Error checking is left as an exercise for the reader.

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;
```

How do I make a temporary file name?

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 IO::File;
        use Fcntl;
        my $temp_dir = -d '/tmp' ? '/tmp' : $ENV{TMP} || $ENV{TEMP};
        my $base_name = sprintf("%s/%d-%d-0000", $temp_dir, $$, time());
        sub temp_file {
            my $fh = undef;
            my $count = 0;
            until (defined($fh) || $count > 100) {
                $base_name =~ s/-(\d+)$/"-" . (1 + $1)/e;
                $fh = IO::File->new($base_name, O_WRONLY|O_EXCL|O_CREAT, 0644)
            }
            if (defined($fh)) {
                return ($fh, $base_name);
            } else {
                return ();
            }
        }
    }
```

Or you could simply use IO::Handle::new_tmpfile.

How can I manipulate fixed−record−length files?

The most efficient way is using `pack()` and `unpack()`. This is faster than using `substr()`. 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
    $PS_T = 'A6 A4 A7 A5 A*';
    open(PS, "ps|");
    $_ = <PS>; print;
    while (<PS>) {
        ($pid, $tt, $stat, $time, $command) = unpack($PS_T, $_);
        for $var (qw!pid tt stat time command!) {
            print "$var: <$$var>\n";
        }
        print 'line=', pack($PS_T, $pid, $tt, $stat, $time, $command),
                "\n";
    }
```

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?

You may have some success with typeglobs, as we always had to use in days of old:

```
    local(*FH);
```

But while still supported, that isn‘t the best to go about getting local filehandles. Typeglobs have their drawbacks. You may well want to use the `FileHandle` module, which creates new filehandles for you (see *FileHandle*):

```
    use FileHandle;
    sub findme {
        my $fh = FileHandle->new();
        open($fh, "</etc/hosts") or die "no /etc/hosts: $!";
        while (<$fh>) {
            print if /\b127\.(0\.0\.)?1\b/;
        }
        # $fh automatically closes/disappears here
    }
```

Internally, Perl believes filehandles to be of class IO::Handle. You may use that module directly if you‘d like (see *IO::Handle*), or one of its more specific derived classes.

How can I set up a footer format to be used with `write()`?

There‘s no built−in way to do this, but *perlform* has a couple of techniques to make it possible for the intrepid hacker.

How can I `write()` into a string?

See *perlform* for an `swrite()` function.

How can I output my numbers with commas added?

This one will do it for you:

```
    sub commify {
        local $_  = shift;
        1 while s/^(-?\d+)(\d{3})/$1,$2/;
        return $_;
    }

    $n = 23659019423.2331;
    print "GOT: ", commify($n), "\n";

    GOT: 23,659,019,423.2331
```

You can‘t just:

```
    s/^(-?\d+)(\d{3})/$1,$2/g;
```

because you have to put the comma in and then recalculate your position.

How can I translate tildes (~) in a filename?

Use the <> (`glob()`) operator, documented in *perlfunc*. This requires that you have a shell installed that groks tildes, meaning csh or tcsh or (some versions of) ksh, and thus may have portability problems. The Glob::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;
```

How come when I open the 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
```

Whoops. You should instead use this, which will fail if the file doesn‘t exist.

```
    open(FH, "+< /path/name");  # open for update
```

If this is an issue, try:

```
    sysopen(FH, "/path/name", O_RDWR|O_CREAT, 0644);
```

Error checking is left as an exercise for the reader.

Why do I sometimes get an "Argument list too long" when I use <*>?

The <> operator performs a globbing operation (see above). By default `glob()` 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 do the glob yourself with `Dirhandle`s and patterns, or use a module like Glob::KGlob, one that doesn‘t use the shell to do globbing.

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 leak and/or unpredictable behavior. It‘s best therefore to use `glob()` only in list context.

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. To avoid this, you might want to use a routine like this. It makes incomplete pathnames into explicit relative ones, and tacks a trailing null byte on the name to make perl leave it alone:

```
    sub safe_filename {
        local $_  = shift;
        return m#^/#
               ? "$_\0"
               : "./$_\0";
    }

    $fn = safe_filename("<<<something really wicked   ");
    open(FH, "> $fn") or "couldn't open $fn: $!";
```

You could also use the `sysopen()` function (see *sysopen*).

How can I reliably rename a file?

Well, usually you just use Perl‘s `rename()` function. But that may not work everywhere, in particular, renaming files across file systems. If your operating system supports a mv(1) program or its moral equivalent, this works:

```
rename($old, $new) or system("mv", $old, $new);
```

It may be more compelling 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 semantics as a real `rename()`, though, which preserves metainformation like permissions, timestamps, inode info, etc.

How can I lock a file?

Perl‘s built−in `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. See the flock entry of *perlfunc*, and the ***INSTALL*** file in the source distribution for information on building Perl to do this.

The CPAN module File::Lock offers similar functionality and (if you have dynamic loading) won‘t require you to rebuild perl if your `flock()` can‘t lock network files.

What 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, 0644)
            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 involving `link()` have been suggested, but these tend to involve busy−wait, which is also subdesirable.

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?

Anyway, this is what to do:

```
use Fcntl;
sysopen(FH, "numfile", O_RDWR|O_CREAT, 0644) or die "can’t open numfile: $!";
flock(FH, 2)                                 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: $!";
# DO NOT UNLOCK THIS UNTIL YOU CLOSE
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. :−)

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, $recno * $RECSIZE, 0);
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.

Don‘t forget to set `binmode()` under DOS−like platforms when operating on files that have anything other than straight text in them. See the docs on `open()` and on `binmode()` for more details.

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** filetest 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. 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];
print "file $file updated at ", scalar(localtime($file)), "\n";
```

If you prefer something more legible, use the File::stat module (part of the standard distribution in version 5.004 and later):

```
use File::stat;
use Time::localtime;
$date_string = ctime(stat($file)->mtime);
print "file $file updated at $date_string\n";
```

Error checking is left as an exercise for the reader.

How do I set a file‘s timestamp in perl?

You use the `utime()` function documented in *utime*. 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 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 it on those platforms.

How do I print to more than one file at once?

If you only have to do this once, you can do this:

```
    for $fh (FH1, FH2, FH3) { print $fh "whatever\n" }
```

To connect up to one filehandle to several output filehandles, it‘s easiest to use the tee(1) program if you have it, and let it take care of the multiplexing:

```
    open (FH, "| tee file1 file2 file3");
```

Otherwise you‘ll have to write your own multiplexing print function — or your own tee program — or use Tom Christiansen‘s, at http://www.perl.com/CPAN/authors/id/TOMC/scripts/tct.gz, which is written in Perl.

In theory a IO::Tee class could be written, but to date we haven‘t seen such.

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.

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*.

If your system supports 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:

```
    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;
```

For DOS systems, Dan Carson <dbc@tc.fluke.COM reports the following:

To put the PC in "raw" mode, use ioctl with some magic numbers gleaned from msdos.c (Perl source file) and Ralf Brown‘s interrupt list (comes across the net every so often):

```
    $old_ioctl = ioctl(STDIN,0,0);     # Gets device info
    $old_ioctl &= 0xff;
    ioctl(STDIN,1,$old_ioctl | 32);    # Writes it back, setting bit 5
```

Then to read a single character:

```
    sysread(STDIN,$c,1);               # Read a single character
```

And to put the PC back to "cooked" mode:

```
    ioctl(STDIN,1,$old_ioctl);         # Sets it back to cooked mode.
```

So now you have `$c`. If `ord($c) == 0`, you have a two byte code, which means you hit a special key. Read another byte with `sysread(STDIN,$c,1)`, and that value tells you what combination it was according to this table:

```
    # PC 2-byte keycodes = ^@ + the following:

    # HEX     KEYS
    # ---     ----
    # 0F      SHF TAB
    # 10-19   ALT QWERTYUIOP
    # 1E-26   ALT ASDFGHJKL
    # 2C-32   ALT ZXCVBNM
    # 3B-44   F1-F10
    # 47-49   HOME,UP,PgUp
    # 4B      LEFT
    # 4D      RIGHT
    # 4F-53   END,DOWN,PgDn,Ins,Del
    # 54-5D   SHF F1-F10
    # 5E-67   CTR F1-F10
    # 68−71   ALT F1−F10
    # 73−77   CTR LEFT,RIGHT,END,PgDn,HOME
    # 78−83   ALT 1234567890−=
    # 84      CTR PgUp
```

This is all trial and error I did a long time ago, I hope I‘m reading the file that worked.

How can I tell if there‘s a character waiting on a filehandle?

You should 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);
    }
```

You should look into getting the Term::ReadKey extension from CPAN.

How do I open a file without blocking?

You need to use the O_NDELAY or O_NONBLOCK flag from the Fcntl module in conjunction with `sysopen()`:

```
    use Fcntl;
    sysopen(FH, "/tmp/somefile", O_WRONLY|O_NDELAY|O_CREAT, 0644)
    or die "can’t open /tmp/somefile: $!":
```

How do I create a file only if it doesn‘t exist?

You need to use the O_CREAT and O_EXCL flags from the Fcntl module in conjunction with `sysopen()`:

```
    use Fcntl;
    sysopen(FH, "/tmp/somefile", O_WRONLY|O_EXCL|O_CREAT, 0644)
                or die "can’t open /tmp/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 successful create or unlink the same file!

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.

How do I dup( ) a filehandle in Perl?

If you check *open*, you‘ll see that several of the ways to call `open()` should do the trick. For example:

```
    open(LOG, ">>/tmp/logfile");
    open(STDERR, ">&LOG");
```

Or even with a literal numeric descriptor:

```
   $fd = $ENV{MHCONTEXTFD};
   open(MHCONTEXT, "<&=$fd");   # like fdopen(3S)
```

Error checking has been left as an exercise for the reader.

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;
```

Why can‘t I use "C:\temp\foo" in DOS paths? What 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*. Unsurprisingly, you don‘t have a file called "c:(tab)emp(formfeed)oo" or "c:(tab)emp(formfeed)oo.exe" on your 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.

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.

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 "Far More Than You Every Wanted To Know" in http://www.perl.com/CPAN/doc/FMTEYEWTK/file−dir−perms .

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.

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.

AUTHOR AND COPYRIGHT

Copyright (c) 1997 Tom Christiansen and Nathan Torkington. All rights reserved. See *perlfaq* for distribution information.

NAME

perlfaq6 − Regexps (`$Revision:` 1.14 `$)`

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 the section on Data and the Networking one on networking, to be precise).

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 Regexp

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) /ge;
```

Comments Inside the Regexp

The `/x` modifier causes whitespace to be ignored in a regexp 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
```

I‘m having trouble matching over more than one line. What‘s wrong?

Either you don‘t have newlines in your string, or you aren‘t using the correct modifier(s) on your pattern.

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+)(\s+\1)+\b/gi ) {
        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/sm ) { # /s makes . cross line boundaries
        print "$1\n";
    }
}
```

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 −pe '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.

I put a regular expression into $/ but it didn‘t work. What‘s wrong?

`$/` must be a string, not a regular expression. Awk has to be better for something. :−)

Actually, you could do this if you don‘t mind reading the whole file into

```
undef $/;
@records = split /your_pattern/, <FH>;
```

How do I substitute case insensitively on the LHS, but preserving case on the RHS?

It depends on what you mean by "preserving case". The following script makes the substitution have the same case, letter by letter, as the original. 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;
        }

        $a = "this is a TEsT case";
        $a =~ s/(test)/preserve_case($1, "success")/gie;
        print "$a\n";
```

This prints:

```
        this is a SUcCESS case
```

How can I make `\w` match accented characters?

See *perllocale*.

How can I match a locale−smart version of `/[a−zA−Z]/`?

One alphabetic character would be `/[^\W\d_]/`, no matter what locale you're in. Non−alphabetics would be `/[\W\d_]/` (assuming you don't consider an underscore a letter).

How can I quote a variable to use in a regexp?

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 regexp special characters will be acted on unless you precede the substitution with \Q. Here's an example:

```
        $string = "to die?";
        $lhs = "die?";
        $rhs = "sleep no more";

        $string =~ s/\Q$lhs/$rhs/;
        # $string is now "to sleep no more"
```

Without the \Q, the regexp would also spuriously match "di".

What is `/o` really for?

Using a variable in a regular expression match forces a re−evaluation (and perhaps recompilation) each time through. The `/o` modifier locks in the regexp 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 regexp 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 regexp to notice if they do.

For example, here‘s a "paragrep" program:

```
    $/ = '';  # paragraph mode
    $pat = shift;
    while (<>) {
        print if /$pat/o;
    }
```

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:

```
    $/ = undef;
    $_ = <>;
    s#/\*[^*]*\*+([^/*][^*]*\*+)*/|("(\\.|[^"\\])*"|'(\\.|[^'\\])*'|\n+|.[^/"'\\]*)#$
    print;
```

This could, of course, be more legibly written with the `/x` modifier, adding whitespace and comments.

Can I use Perl regular expressions to match balanced text?

Although Perl regular expressions are more powerful than "mathematical" regular expressions, because they feature conveniences like backreferences (`\1` and its ilk), they still aren‘t powerful enough. You still need to use non−regexp techniques to parse balanced text, such as the text enclosed between matching parentheses or braces, for example.

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.perl.com/CPAN/authors/id/TOMC/scripts/pull_quotes.gz .

The C::Scan module from CPAN contains such subs for internal usage, but they are undocumented.

What does it mean that regexps are greedy? How can I get around it?

Most people mean that greedy regexps 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.

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, you might consider

```
while (<>) {
    foreach $word (m/(\w+)/g) {
        # do something with $word here
    }
}
```

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 the section on Hashes.

How can I do approximate matching?

See the module String::Approx available from CPAN.

How do I efficiently match many regular expressions at once?

The following is super−inefficient:

```
while (<FH>) {
    foreach $pat (@patterns) {
        if ( /$pat/ ) {
            # do something
        }
    }
    }
```

Instead, you either need to use one of the experimental Regexp extension modules from CPAN (which might well be overkill for your purposes), or else put together something like this, inspired from a routine in Jeffrey Friedl‘s book:

```
    sub _bm_build {
        my $condition = shift;
        my @regexp = @_;  # this MUST not be local(); need my()
        my $expr = join $condition => map { "m/\$regexp[$_]/o" } (0..$#regexp);
        my $match_func = eval "sub { $expr }";
        die if $@;  # propagate $@; this shouldn't happen!
        return $match_func;
    }

    sub bm_and { _bm_build('&&', @_) }
    sub bm_or  { _bm_build('||', @_) }

    $f1 = bm_and qw{
            xterm
            (?i)window
    };

    $f2 = bm_or qw{
            \b[Ff]ree\b
            \bBSD\B
            (?i)sys(tem)?\s*[V5]\b
    };

    # feed me /etc/termcap, prolly
    while ( <> ) {
        print "1: $_" if &$f1;
        print "2: $_" if &$f2;
    }
```

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 behaviour of all the regexp 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".

Why does using `$&, $‘,` or `$'` slow my program down?

Because once Perl sees that you need one of these variables anywhere in the program, it has to provide them on each and every pattern match. The same mechanism that handles these provides for the use of `$1, $2,` etc., so you pay the same price for each regexp that contains capturing parentheses. But if you never use `$&,` etc., in your script, then regexps *without* capturing parentheses won‘t 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.

What good is `\G` in a regular expression?

The notation `\G` is used in a match or substitution in conjunction the `/g` modifier (and ignored if there's no `/g`) to anchor the regular expression to the point just past where the last match occurred, i.e. the `pos()` point.

For example, suppose you had a line of text quoted in standard mail and Usenet notation, (that is, with leading `>` characters), and you want change each leading `>` into a corresponding `:`. You could do so in this way:

```
     s/^(>+)/':' x length($1)/gem;
```

Or, using `\G`, the much simpler (and faster):

```
    s/\G>/:/g;
```

A more sophisticated use might involve a tokenizer. The following lex−like example is courtesy of Jeffrey Friedl. It did not work in 5.003 due to bugs in that release, but does work in 5.004 or better:

```
    while (<>) {
      chomp;
      PARSER: {
           m/ \G( \d+\b    )/gx     && do { print "number: $1\n";  redo; };
           m/ \G( \w+      )/gx     && do { print "word:   $1\n";  redo; };
           m/ \G( \s+      )/gx     && do { print "space:  $1\n";  redo; };
           m/ \G( [^\w\d]+ )/gx     && do { print "other:  $1\n";  redo; };
      }
    }
```

Of course, that could have been written as

```
    while (<>) {
      chomp;
      PARSER: {
           if ( /\G( \d+\b    )/gx  {
                print "number: $1\n";
                redo PARSER;
           }
           if ( /\G( \w+      )/gx  {
                print "word: $1\n";
                redo PARSER;
           }
           if ( /\G( \s+      )/gx  {
                print "space: $1\n";
                redo PARSER;
           }
           if ( /\G( [^\w\d]+ )/gx  {
                print "other: $1\n";
                redo PARSER;
           }
      }
    }
```

But then you lose the vertical alignment of the regular expressions.

Are Perl regexps 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*).

What‘s wrong with using grep or map in a void context?

Strictly speaking, nothing. Stylistically speaking, it‘s not a good way to write maintainable code. That‘s because you‘re using these constructs not for their return values but rather for their side−effects, and side−effects can be mystifying. There‘s no void `grep()` that‘s not better written as a `for` (well, `foreach`, technically) loop.

How can I match strings with multi−byte characters?

This is hard, and there‘s no good way. Perl does not directly support wide characters. It pretends that a byte and a character are synonymous. 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 ‘‘maritan’’ 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’;
}
```

Or like this:

```
die "sorry, Perl doesn’t (yet) have Martian support )−:\n";
```

In addition, a sample program which converts half−width to full−width katakana (in Shift−JIS or EUC encoding) is available from CPAN as

=for Tom make it so

There are many double− (and multi−) byte encodings commonly used these days. Some versions of these have 1−, 2−, 3−, and 4−byte characters, all mixed.

AUTHOR AND COPYRIGHT

NAME

perlfaq7 − Perl Language Issues (`$Revision:` 1.15 `$`)

DESCRIPTION

This section deals with 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?

No, 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."

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 all types of that symbol name.  In version 4 you used them like
  pointers, but in modern perls you can just use references.
```

While there are a few places where you don‘t actually need these type specifiers, you should always use them.

A couple of others that you‘re likely to encounter that aren‘t really type specifiers are:

```
<> 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 *$/)* 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 talking about the handle, do *not* use the brackets. These are correct: `eof(FH)`, `seek(FH, 0, 2)` and "copying from STDIN to FILE".

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",
);
```

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);
```

How do I temporarily block warnings?

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()`.

A new `use warnings` pragma is in the works to provide finer control over all this. The curious should check the perl5−porters mailing list archives for details.

What‘s an extension?

A way of calling compiled C code from Perl. Reading *perlxstut* is a good place to learn more about extensions.

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.

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*.

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*.

Here's a convenient template you might wish you use when starting your own module. Make sure to change the names appropriately.

```
package Some::Module;  # assumes Some/Module.pm

use strict;

BEGIN {
    use Exporter   ();
    use vars       qw($VERSION @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS);

    ## set the version for version checking; uncomment to use
    ## $VERSION     = 1.00;

    # if using RCS/CVS, this next line may be preferred,
    # but beware two−digit versions.
    $VERSION = do{my@r=q$Revision: 1.15 $=~/\d+/g;sprintf '%d.'.'%02d'x$#r,@r};

    @ISA         = qw(Exporter);
    @EXPORT      = qw(&func1 &func2 &func3);
    %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);
}
use vars      @EXPORT_OK;

# non−exported package globals go here
use vars      qw( @more $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)

1;            # modules must return true
```

How do I create a class?

See *perltoot* for an introduction to classes and objects, as well as *perlobj* and *perlbot*.

How can I tell if a variable is tainted?

See *Laundering and Detecting Tainted Data in perlsec*. Here's an example (which doesn't use any system calls, because the `kill()` is given no processes to signal):

```
sub is_tainted {
    return ! eval { join('',@_), kill 0; 1; };
}
```

This is not `-w` clean, however. There is no `-w` clean way to detect taintedness − take this as a hint that you should untaint all possibly−tainted data.

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.

How can I pass/return a {Function, FileHandle, Array, Hash, Method, Regexp}?

With the exception of regexps, you need to pass references to these objects. See *Pass by Reference in perlsub* for this particular question, and *perlref* for information on references.

Passing Variables and Functions

Regular variables and functions are quite easy: 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

To create filehandles you can pass to subroutines, you can use `*FH` or `\*FH` notation ("typeglobs" − see *perldata* for more information), or create filehandles dynamically using the old FileHandle or the new IO::File modules, both part of the standard Perl distribution.

```
use Fcntl;
use IO::File;
my $fh = new IO::File $filename, O_WRONLY|O_APPEND;
            or die "Can't append to $filename: $!";
func($fh);
```

Passing Regexps

To pass regexps around, you'll need to either use one of the highly experimental regular expression modules from CPAN (Nick Ing−Simmons's Regexp or Ilya Zakharevich's Devel::Regexp), pass around strings and use an exception−trapping eval, or else be be very, very clever. Here's an example of how to pass in a string to be regexp compared:

```
sub compare($$) {
    my ($val1, $regexp) = @_;
    my $retval = eval { $val =~ /$regexp/ };
    die if $@;
    return $retval;
}

$match = compare("old McDonald", q/d.*D/);
```

Make sure you never say something like this:

```
return eval "\$val =~ /$regexp/";   # WRONG
```

or someone can sneak shell escapes into the regexp 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 and 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).

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 it 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.

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 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 also *perlsub*, which explains this all in more detail.

How can I access a dynamic variable while a similarly named lexical is in scope?

You can do this via symbolic references, provided you haven‘t set `use strict "refs"`. So instead of `$var,` use `${‘var‘}.`

```
    local $var = "global";
    my    $var = "lexical";

    print "lexical is $var\n";

    no strict 'refs';
    print "global  is ${'var'}\n";
```

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.` Specifying the package directly makes you hard−code its name, but it executes faster and avoids running afoul of `use strict "refs"`.

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 *"What‘s a closure?"*.

Why doesn‘t "local($foo) = <FILE;" work right?

`local()` gives 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
```

How do I redefine a built−in 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 Builtin 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*.

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 by−pass prototypes. That means that 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.

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, regexp 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.

Here‘s a simple example of a switch based on pattern matching. We‘ll do a multi−way 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";

    }
```

How can I catch accesses to undefined variables/functions/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 `−w`, you can use a handler to trap the pseudo−signal `__WARN__` like this:

```
    $SIG{__WARN__} = sub {

        for ( $_[0] ) {

            /Use of uninitialized value/  && do {
                # promote warning to a fatal
                die $_;
            };

            # other warning cases to catch could go here;

            warn $_;
        }

    };
```

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 on these. 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*.

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 = ref bless [];
```

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";
}
```

AUTHOR AND COPYRIGHT

NAME

perlfaq8 − System Interaction (`$Revision: 1.15 $`)

DESCRIPTION

This section of the Perl FAQ covers questions involving operating system interaction. This involves 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.

How do I find out which operating system I‘m running under?

The `$^O` variable (`$OSTYPE` if you use English) contains the operating system that your perl binary was built for.

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.

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
```

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*). 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*, and Chapter 7 of the Camel), 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.

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, the devices 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 behaviour 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, as in the older

```
use FileHandle;
DEV−>autoflush(1);
```

and the newer

```
use IO::Handle;
DEV−>autoflush(1);
```

You can use `select()` and the `$|` variable to control autoflushing (see *$|* and *select*):

```
$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]);
```

As mentioned in the previous item, this still doesn't work when using socket I/O between Unix and Macintosh. You'll need to hardcode 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*). 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*.

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. 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).

How do I start a process in the background?

You could 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 `open`ing 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 };
```

See *Signals in perlipc* for other examples of code to do this. Zombies are not an issue with `system("prog &")`.

How do I trap control characters/signals?

You don't actually "trap" a control character. Instead, that character generates a signal, which you then trap. Signals are documented in *Signals in perlipc* and chapter 6 of 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: set a variable and exit. And 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 chapter 6 of the Camel.

How do I modify the shadow password file on a Unix system?

If perl was installed correctly, the `getpw*()` functions described in *perlfunc* provide (read−only) access to the shadow password file. To change the file, make a new shadow password file (the format varies from system to system − see *passwd(5)* for specifics) and use pwd_mkdb(8) to install it (see *pwd_mkdb(5)* for more details).

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 timezone, 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";
```

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*. If your system has itimers and `syscall()` support, you can check out the old example in
http://www.perl.com/CPAN/doc/misc/ancient/tutorial/eg/itimers.pl .

How can I measure time under a second?

In general, you may not be able to. The Time::HiRes module (available from CPAN) provides this functionality for some systems.

In general, you may not be able to. But if you 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] );
```

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). It 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* and chapter 6 of the Camel.

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.

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.

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.

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).

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.

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*).

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 versions of these, unless the command redirects them. Backticks and `open()` read **only** the STDOUT of your command.

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 http://www.perl.com/CPAN/doc/FMTEYEWTK/versus/csh.whynot .

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*).

Why doesn‘t open() return an error when a pipe open fails?

It does, but probably not how you expect it to. On systems that follow the standard `fork()/exec()` paradigm (eg, Unix), it works like this: `open()` causes a `fork()`. In the parent, `open()` returns with the process ID of the child. The child `exec()`s the command to be piped to/from. The parent can‘t know whether the `exec()` was successful or not − all it can return is whether the `fork()` succeeded or not. To find out if the command succeeded, you have to catch SIGCHLD and `wait()` to get the exit status.

On systems that follow the `spawn()` paradigm, `open()` *might* do what you expect − unless perl uses a shell to start your command. In this case the `fork()/exec()` description still applies.

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 because backticks have a (potentially humungous) return value, and you‘re ignoring it. It‘s may also not be very efficient, because you have to read in all the lines of output, allocate memory for them, and then throw it away. Too often people are lulled to writing:

```
    `cp file file.bak`;
```

And now they think "Hey, I‘ll just always use backticks to run programs." Bad idea: backticks are for capturing a program‘s output; the `system()` function is for running programs.

Consider this line:

```
    `cat /etc/termcap`;
```

You haven‘t assigned the output anywhere, so it just wastes memory (for a little while). Plus you forgot to check `$?` to see whether the program even ran correctly. Even if you wrote

```
    print `cat /etc/termcap`;
```

In most cases, this could and probably should be written as

```
    system("cat /etc/termcap") == 0
        or die "cat program failed!";
```

Which will get the output quickly (as its 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.

How can I call backticks without shell processing?

This is a bit tricky. Instead of writing

```
    @ok = ‘grep @opts ’$search_string’ @filenames‘;
```

You have to do this:

```
    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.

Why can‘t my script read from STDIN after I gave it EOF (^D on Unix, ^Z on MSDOS)?

Because 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.

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.

Can I use perl to run a telnet or ftp session?

Try the Net::FTP and TCP::Client modules (available from CPAN).
http://www.perl.com/CPAN/scripts/netstuff/telnet.emul.shar will also help for emulating the telnet protocol.

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. These days, your best bet is to look at the Comm.pl library available from CPAN.

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]";
```

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 its own 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.

VMS

Change to %ENV persist after Perl exits, but directory changes do not.

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.

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 the TIOCNOTTY ioctl on it. See *tty(4)* for details.
- Change directory to /
- Reopen STDIN, STDOUT, and STDERR so they‘re not connected to the old tty.
- Background yourself like this:

  ```
  fork && exit;
  ```

How do I make my program run with sh and csh?

See the ***eg/nih*** script (part of the perl source distribution).

How do I keep my own module/library directory?

When you build modules, use the PREFIX option when generating Makefiles:

```
perl Makefile.PL PREFIX=/u/mydir/perl
```

then either set the PERL5LIB environment variable before you run scripts that use the modules/libraries (see *perlrun*) or say

```
use lib ’/u/mydir/perl’;
```

See Perl‘s *lib* for more information.

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(TTY);
    $pgrp = getpgrp();
    if ($tpgrp == $pgrp) {
        print "foreground\n";
    } else {
        print "background\n";
    }
```

How do I timeout a slow event?

Use the `alarm()` function, probably in conjunction with a signal handler, as documented *Signals in perlipc* and chapter 6 of the Camel. You may instead use the more flexible Sys::AlarmCall module available from CPAN.

How do I set CPU limits?

Use the BSD::Resource module from CPAN.

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 *fork*.

How do I use an SQL database?

There are a number of excellent interfaces to SQL databases. See the DBD::* modules available from http://www.perl.com/CPAN/modules/dbperl/DBD .

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.

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, "/tmp/somefile", O_WRONLY|O_NDELAY|O_CREAT, 0644)
        or die "can’t open /tmp/somefile: $!":
```

How do I install a CPAN module?

The easiest way is to have the CPAN module do it for you. This module comes with perl version 5.004 and later. 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.

AUTHOR AND COPYRIGHT

NAME

perlfaq9 − Networking ($Revision: 1.13 $)

DESCRIPTION

This section deals with questions related to networking, the internet, and a few on the web.

My CGI script runs from the command line but not the browser. Can you help me fix it?

Sure, but you probably can‘t afford our contracting rates :−)

Seriously, if you can demonstrate that you‘ve read the following 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, HTML, or the CGI protocols). Questions that appear to be Perl questions but are really CGI ones that are posted to comp.lang.perl.misc may not be so well received.

The useful FAQs are:

```
http://www.perl.com/perl/faq/idiots−guide.html
http://www3.pair.com/webthing/docs/cgi/faqs/cgifaq.shtml
http://www.perl.com/perl/faq/perl−cgi−faq.html
http://www−genome.wi.mit.edu/WWW/faqs/www−security−faq.html
http://www.boutell.com/faq/
```

How do I remove HTML from a string?

The most correct way (albeit not the fastest) is to use HTML::Parse from CPAN (part of the libwww−perl distribution, which is a must−have module for all web hackers).

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.perl.com/CPAN/authors/Tom_Christiansen/scripts/striphtml.gz .

How do I extract URLs?

A quick but imperfect approach is

```
#!/usr/bin/perl −n00
# qxurl − tchrist@perl.com
print "$2\n" while m{
    < \s*
      A \s+ HREF \s* = \s* (["’]) (.*?) \1
    \s* >
}gsix;
```

This version does not adjust relative URLs, understand alternate bases, deal with HTML comments, or accept URLs themselves as arguments. It also runs about 100x faster than a more "complete" solution using the LWP suite of modules, such as the http://www.perl.com/CPAN/authors/Tom_Christiansen/scripts/xurl.gz program.

How do I download a file from the user‘s machine? How do I open a file on another machine?

In the context of an HTML form, you can use what‘s known as **multipart/form−data** encoding. The CGI.pm module (available from CPAN) supports this in the `start_multipart_form()` method, which isn‘t the same as the `startform()` method.

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.

How do I fetch an HTML file?

Use the LWP::Simple module available from CPAN, part of the excellent libwww−perl (LWP) package. On the other hand, and if you have the lynx text−based HTML browser installed on your system, this isn‘t too bad:

```
$html_code = ‘lynx −source $url‘;
$text_data = ‘lynx −dump $url‘;
```

how do I decode or create those %−encodings on the web?

Here‘s an example of decoding:

```
$string = "http://altavista.digital.com/cgi−bin/query?pg=q&what=news&fmt=.&q=%2Bc
$string =~ s/%([a−fA−F0−9]{2})/chr(hex($1))/ge;
```

Encoding is a bit harder, because you can‘t just blindly change all the non−alphanumunder character (`\W`) into their hex escapes. It‘s important that characters with special meaning like `/` and `?` *not* be translated. Probably the easiest way to get this right is to avoid reinventing the wheel and just use the URI::Escape module, which is part of the libwww−perl package (LWP) available from CPAN.

How do I redirect to another page?

Instead of sending back a `Content−Type` as the headers of your reply, send back a `Location:` header. Officially this should be a `URI:` header, so the CGI.pm module (available from CPAN) sends back both:

```
Location: http://www.domain.com/newpage
URI: http://www.domain.com/newpage
```

Note that relative URLs in these headers can cause strange effects because of "optimizations" that servers do.

How do I put a password on my web pages?

That depends. You‘ll need to read the documentation for your web server, or perhaps check some of the other FAQs referenced above.

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, Berkley 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);
```

How do I parse an email header?

For a quick−and−dirty solution, try this solution derived from page 222 of the 2nd edition of "Programming Perl":

```
$/ = ’’;
$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).

How do I decode a CGI form?

A lot of people are tempted to code this up themselves, so you‘ve probably all seen a lot of code involving `$ENV{CONTENT_LENGTH}` and `$ENV{QUERY_STRING}`. It‘s true that this can work, but there are also a lot of versions of this floating around that are quite simply broken!

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://www.bio.cam.ac.uk/web/form.html).

How do I check a valid email address?

You can‘t.

Without sending mail to the address and seeing whether it bounces (and even then you face the halting problem), you cannot determine whether an email address is valid. Even if you apply the email 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.

Many are tempted to try to eliminate many frequently−invalid email addresses with a simple regexp, such as `/^[\w.−]+\@([\w.−]\.)+\w+$/`. However, this also throws out many valid ones, and says nothing about potential deliverability, so is not suggested. Instead, see
http://www.perl.com/CPAN/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 email to (say, Bill Clinton or your postmaster), and then makes sure that the hostname given can be looked up in DNS. It‘s not fast, but it works.

How do I decode a MIME/BASE64 string?

The MIME−tools package (available from CPAN) handles this and a lot more. Decoding BASE64 becomes as simple as:

```
    use MIME::base64;
    $decoded = decode_base64($encoded);
```

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
```

How do I return the user‘s email 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;
    $address = sprintf('%s@%s', getpwuid($<), hostname);
```

Company policies on email address can mean that this generates addresses that the company‘s email system will not accept, so you should ask for users’ email 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.

How do I send/read mail?

Sending mail: the Mail::Mailer module from CPAN (part of the MailTools package) is UNIX−centric, while Mail::Internet uses Net::SMTP which is not UNIX−centric. Reading mail: use the Mail::Folder module from CPAN (part of the MailFolder package) or the Mail::Internet module from CPAN (also part of the MailTools package).

How do I find out my hostname/domainname/IP address?

A lot of code has historically cavalierly called the `‘hostname‘` program. While sometimes expedient, this isn‘t very portable. 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 $addr = inet_ntoa(scalar(gethostbyname($name)) || ’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.)

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")’
```

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.

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). No ONC::RPC module is known.

AUTHOR AND COPYRIGHT

NAME

Perl Kit, Version 5.0

Perl is a language that combines some of the features of C, sed, awk and shell. See the manual page for more hype. There are also two Nutshell Handbooks published by O'Reilly & Assoc. See pod/perlbook.pod for more information.

Please read all the directions below before you proceed any further, and then follow them carefully.

After you have unpacked your kit, you should have all the files listed in MANIFEST.

Installation

1) Detailed instructions are in the file INSTALL. In brief, the following should work on most systems:

```
        rm −f config.sh
        sh Configure
        make
        make test
        make install
```

For most systems, it should be safe to accept all the Configure defaults.

2) Read the manual entries before running perl.

3) IMPORTANT! Help save the world! Communicate any problems and suggested patches to me, larry@wall.org (Larry Wall), so we can keep the world in sync. If you have a problem, there‘s someone else out there who either has had or will have the same problem. It‘s usually helpful if you send the output of the "myconfig" script in the main perl directory.

If you‘ve succeeded in compiling perl, the perlbug script in the utils/ subdirectory can be used to help mail in a bug report.

If possible, send in patches such that the patch program will apply them. Context diffs are the best, then normal diffs. Don‘t send ed scripts— I‘ve probably changed my copy since the version you have.

Watch for perl patches in comp.lang.perl.announce. Patches will generally be in a form usable by the patch program. If you are just now bringing up perl and aren‘t sure how many patches there are, write to me and I‘ll send any you don‘t have. Your current patch level is shown in patchlevel.h.

Just a personal note: I want you to know that I create nice things like this because it pleases the Author of my story. If this bothers you, then your notion of Authorship needs some revision. But you can use perl anyway. :−)

```
                                        The author.
```

NAME

perlamiga − Perl under Amiga OS

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***.

DESCRIPTION

Prerequisites

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 following archives from ftp://ftp.ninemoons.com/pub/ade/current or a mirror:

ixemul−45.1−bin.lha ixemul−45.1−env−bin.lha pdksh−4.9−bin.lha ADE−misc−bin.lha

Note that there might be newer versions available by the time you read this.

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.

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 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.)

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
```

INSTALLATION

Change to the installation directory (most probably ADE:), and extract the binary distribution:

lha −mraxe x perl−5.003−bin.lha

or

tar xvzpf perl−5.003−bin.tgz

(Of course you need lha or tar and gunzip for this.)

For installation of the Unix emulation, read the appropriate docs.

Accessing documentation

Manpages

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*.

HTML

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`.

GNU `info` files

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.

`LaTeX` docs

can be constructed using `pod2latex`.

BUILD

Here we discuss how to build Perl under AmigaOS.

Prerequisites

You need to have the latest **ADE** (Amiga Developers Environment) from ftp://ftp.ninemoons.com/pub/ade/current. Also, you need a lot of free memory, probably at least 8MB.

Getting the perl source

You can either get the latest perl−for−amiga source from Ninemoons and extract it with:

```
  tar xvzpf perl−5.004−src.tgz
```

or get the official source from CPAN:

```
  http://www.perl.com/CPAN/src/5.0
```

Extract it like this

```
tar xvzpf perl5.004.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.)

Making

```
sh configure.gnu −−prefix=/ade
```

Now

```
make
```

Testing

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***

Installing the built perl

Run

```
make install
```

AUTHOR

Norbert Pueschel, pueschel@imsdd.meb.uni−bonn.de

SEE ALSO

perl(1).

NAME

perlos2 − Perl under OS/2, DOS, Win0.3*, Win0.95 and WinNT.

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).

Note that if you have ***lynx.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).

DESCRIPTION

Target

The target is to make OS/2 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, but currently `fork()` is not supported after *use*ing dynamically loaded extensions.
- You need a separate perl executable ***perl__.exe*** (see *perl__.exe*) to use PM code in your application (like the forthcoming Perl/Tk).
- There is no simple way to access WPS objects. The only way I know is via `OS2::REXX` extension (see *OS2::REXX*), and we do not have access to convenience methods of Object−REXX. (Is it possible at all? I know of no Object−REXX API.)

Please keep this list up−to−date by informing me about other items.

Other OSes

Since OS/2 port of perl uses a remarkable EMX environment, it can run (and build extensions, and − possibly − be build 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 *"perl_.exe"*.

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*.

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.

Only the latest runtime is supported, currently `0.9c`. Perl may run under earlier versions of EMX, but this is not tested.

One can get different parts of EMX from, say

```
ftp://ftp.cdrom.com/pub/os2/emx09c/
ftp://hobbes.nmsu.edu/os2/unix/emx09c/
```

The runtime component should have the name ***emxrt.zip***.

NOTE. It is enough to have ***emx.exe***/***rsx.exe*** 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 *"Other OSes"*). 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 at

```
ftp://ftp.math.ohio−state.edu/pub/users/ilya/os2/sh_dos.zip
```

HPFS Perl does not care about file systems, but to install the whole perl library 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 *"PERL_SH_DIR"*).

For best results use EMX pdksh. The soon−to−be−available standard binary (5.2.12?) runs under DOS (with *RSX*) as well, meanwhile use the binary from

```
ftp://ftp.math.ohio−state.edu/pub/users/ilya/os2/sh_dos.zip
```

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 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 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*.

Starting OS/2 (and DOS) programs under Perl

This is what `system()` (see *system*), `‘‘` (see *I/O Operators in perlop*), and *open pipe* (see *open*) are for. (Avoid `exec()` (see *exec*) 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 *"Pdksh"*, *"Frequently asked questions"*), and perl should be able to find it (see *"PERL_SH_DIR"*).

The only cases when the shell is not used is the multi−argument `system()` (see *`system`*)`/exec()` (see *exec*), and one−argument version thereof without redirection and shell meta−characters.

Frequently asked questions

I cannot run external programs

Did you run your programs with `−w` switch? See *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.

I cannot embed perl into my program, or use *perl.dll* from my

program.

Is your program EMX−compiled with `−Zmt −Zcrtdll`?

If not, 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*?

I had reports it does not work. Somebody would need to fix it.

`` ` ` `` and pipe−open do not work under DOS.

This may a variant of just *"I cannot run external programs"*, or a deeper problem. Basically: you *need* RSX (see *"Prerequisites"*) 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 *"Prerequisites"* under RSX. Do not forget to set variable *"PERL_SH_DIR"* as well.

DPMI is required for RSX.

Cannot start `find.exe "pattern" file`

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. In fact ***find.exe*** cannot be started at all using C library API only. Otherwise the following command−lines were equivalent:

```
find "pattern" file
find pattern file
```

INSTALLATION

Automatic binary installation

The most convenient way of installing 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
```

A folder is created on your desktop which contains some useful objects.

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 *"PERL_BADLANG"*.

`PERL_BADFREE` see *"PERL_BADFREE"*.

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.

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.

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:

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 preserved, you do not need to change anything. However, for perl to find it if it is changed, you need to `set PERLLIB_PREFIX` in ***Config.sys***, see *"PERLLIB_PREFIX"*.

Additional Perl modules

```
    unzip perl_ste.zip −d f:/perllib/lib/site_perl
```

If you do not change this directory, do nothing. Otherwise 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*.

Tools to compile Perl modules

```
    unzip perl_blb.zip −d f:/perllib/lib
```

If this directory is preserved, you do not need to change anything. However, for perl to find it if it is changed, you need to `set PERLLIB_PREFIX` in ***Config.sys***, see *"PERLLIB_PREFIX"*.

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 by `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.

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 *"PERL_SH_DIR"*) if you move ***sh.exe*** from the above location.

Note. It may be possible to use some other sh−compatible shell (*not tested*).

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:/`).

Warning

The automatic and manual perl installation leave precompiled paths inside perl executables. While these paths are overwriteable (see *"PERLLIB_PREFIX"*, *"PERL_SH_DIR"*), one may get better results by binary editing of paths inside the executables/DLLs.

Accessing documentation

Depending on how you built/installed perl you may have (otherwise identical) Perl documentation in the following formats:

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 *"SYNOPSIS"*.

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.

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.

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
```

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.

GNU `info` files

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.

.PDF files

for `Acrobat` are available on CPAN (for slightly old version of perl).

`LaTeX` docs

can be constructed using `pod2latex`.

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*.

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***.

Possible locations to get this from are

```
  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/
```

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.

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***.

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, press `Ctrl−C`.

Getting perl source

You need to fetch the latest perl source (including developers releases). With some probability it is located in

```
http://www.perl.com/CPAN/src/5.0
http://www.perl.com/CPAN/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.perl.com/CPAN/ports/os2/ilyaz/
```

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.

Application of the patches

You need to apply the patches in ***./os2/diff.**** and ***./os2/POSIX.mkfifo*** like this:

```
gnupatch −p0 < os2\POSIX.mkfifo
gnupatch −p0 < os2\diff.configure
```

You may also need to apply the patches supplied with the binary distribution of perl.

Note also that the ***db.lib*** and ***db.a*** from the EMX distribution are not suitable for multi−threaded compile (note that currently perl is not multithread−safe, but is compiled as multithreaded for compatibility with XFree86−OS/2). Get a corrected one from

```
ftp://ftp.math.ohio−state.edu/pub/users/ilya/os2/db_mt.zip
```

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.

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 *"PERLLIB_PREFIX"*.

Ignore the message about missing `ln`, and about `−c` option to tr. In fact if you 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 most of the build has been finished, and it is the time to move the constructed ***perl.dll*** to some *absolute* location in LIBPATH. After this is done the build should finish without a lot of fuss. *One can avoid the interruption if one has the correct prebuilt version of* ***perl.dll*** *on LIBPATH, but probably this is not needed anymore, since* ***miniperl.exe*** *is linked statically now.*

Warnings which are safe to ignore: `mkfifo() redefined` inside ***POSIX.c***.

Testing

Now run

```
make test
```

Some tests (4..6) should fail. Some perl invocations should end in a segfault (system error `SYS3175`). To get finer error reports,

```
cd t
perl harness
```

The report you get may look like

```
Failed Test  Status Wstat Total Fail  Failed  List of failed
---------------------------------------------------------------
io/fs.t                     26   11  42.31%  2−5, 7−11, 18, 25
lib/io_pipe.t     3   768    6   ??       %  ??
lib/io_sock.t     3   768    5   ??       %  ??
op/stat.t                   56    5   8.93%  3−4, 20, 35, 39
Failed 4/140 test scripts, 97.14% okay. 27/2937 subtests failed, 99.08% okay.
```

Note that using 'make test' target two more tests may fail: `op/exec:1` because of (mis)feature of pdksh, and `lib/posix:15`, which checks that the buffers are not flushed on `_exit` (this is a bug in the test which assumes that tty output is buffered).

I submitted a patch to EMX which makes it possible to `fork()` with EMX dynamic libraries loaded, which makes ***lib/io**** tests pass. This means that soon the number of failing tests may decrease yet more.

However, the test ***lib/io_udp.t*** is disabled, since it never terminates, I do not know why. Comments/fixes welcome.

The reasons for failed tests are:

io/fs.t Checks *file system* operations. Tests:

`2−5`, `7−11`	Check `link()` and `inode count` − nonesuch under OS/2.
`18`	Checks `atime` and `mtime` of `stat()` − I could not understand this test.
`25`	Checks `truncate()` on a filehandle just opened for write − I do not know why this should or should not work.

lib/io_pipe.t

Checks `IO::Pipe` module. Some feature of EMX − test `fork()`s with dynamic extension loaded − unsupported now.

lib/io_sock.t

Checks `IO::Socket` module. Some feature of EMX − test `fork()`s with dynamic extension loaded − unsupported now.

op/stat.t Checks `stat()`. Tests:

3 Checks `inode count` − nonesuch under OS/2.

4 Checks `mtime` and `ctime` of `stat()` − I could not understand this test.

20 Checks `-x` − determined by the file extension only under OS/2.

35 Needs ***/usr/bin***.

39 Checks `-t` of ***/dev/null***. Should not fail!

In addition to errors, you should get a lot of warnings.

A lot of 'bad free'

in databases related to Berkeley DB. This is a confirmed bug of DB. You may disable this warnings, see "PERL_BADFREE".

Process terminated by SIGTERM/SIGINT

This is a standard message issued by OS/2 applications. *nix applications die in silence. It is considered 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.

***/sh.exe**: ln: not found
`ls`: /dev: No such file or directory

The last two should be self−explanatory. The test suite discovers that the system it runs on is not *that much* *nixish.

A lot of 'bad free'... in databases, bug in DB confirmed on other platforms. You may disable it by setting PERL_BADFREE environment variable to 1.

Installing the built perl

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 cmdscripts 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*.

`a.out`−style build

Proceed as above, but make ***perl_.exe*** (see *"perl_.exe"*) by

```
make perl_
```

test and install by

```
make aout_test
make aout_install
```

Manually put ***perl_.exe*** to a location on your PATH.

Since `perl_` has the extensions prebuilt, it does not suffer from the *dynamic extensions* + `fork()` syndrome, thus the failing tests look like

```
Failed Test  Status Wstat Total Fail  Failed  List of failed
---------------------------------------------------------------
io/fs.t                      26   11  42.31%  2-5, 7-11, 18, 25
```

```
  op/stat.t                       56    5   8.93%  3-4, 20, 35, 39
  Failed 2/118 test scripts, 98.31% okay. 16/2445 subtests failed, 99.35% okay.
```

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.

Build FAQ

Some / became \ in pdksh.

You have a very old pdksh. See *Prerequisites*.

`'errno'` − unresolved external

You do not have MT−safe ***db.lib***. See *Prerequisites*.

Problems with tr

reported with very old version of tr.

Some problem (forget which ;−)

You have an older version of ***perl.dll*** on your LIBPATH, which broke the build of extensions.

Library ... not found

You did not run `omflibs`. See *Prerequisites*.

Segfault in make

You use an old version of GNU make. See *Prerequisites*.

Specific (mis)features of OS/2 port

`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.

`system()`

Multi−argument form of `system()` allows an additional numeric argument. The meaning of this argument is described in *OS2::Process*.

`extproc` on the first line

If the first chars of a 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).

Additional modules:

OS2::Process, *OS2::REXX*, *OS2::PrfDB*, *OS2::ExtAttr*. This modules provide access to additional numeric argument for `system`, to DLLs having functions with REXX signature and to 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 my ftp directory, mirrored on CPAN.

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)`
`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 *true*, works with END_LIBPATH, otherwise with `BEGIN_LIBPATH`.

`Cwd::extLibpath_set( path [, type ] )`
: Set current value of extended library search path. If `type` is present and *true*, works with END_LIBPATH, otherwise with `BEGIN_LIBPATH`.

(Note that some of these may be moved to different libraries − eventually).

Misfeatures

Since *flock(3)* is present in EMX, but is not functional, the same is true for perl. 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).
- *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.

Since ***sh.exe*** is used for globing (see *glob*), the bugs of ***sh.exe*** plague perl as well.

In particular, uppercase letters do not work in `[...]`−patterns with the current pdksh.

Modifications

Perl modifies some standard C library calls in the following ways:

popen	`my_popen` uses ***sh.exe*** if shell is required, cf. *"PERL_SH_DIR"*.
`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***.

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:

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()`. Unfortunately, with the current version of EMX it cannot `fork()` with dynamic extensions loaded (may be fixed by patches to EMX).

Note. Keep in mind that `fork()` is needed to open a pipe to yourself.

perl_.exe

This is a statically linked `a.out`−style executable. It can `fork()`, but cannot load dynamic Perl extensions. The supplied executable has a lot of extensions prebuilt, thus there are situations when it can perform tasks not possible using ***perl.exe***, like `fork()`ing when having some standard extension loaded. This executable is a VIO application.

Note. A better behaviour could be obtained from `perl.exe` if it were statically linked with standard *Perl extensions*, but dynamically linked with the *Perl DLL* and CRT DLL. Then it would be able to `fork()` with standard extensions, *and* would be able to dynamically load arbitrary extensions. Some changes to Makefiles and hint files should be necessary to achieve this.

This is also 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 *"Other OSes"*.

perl__.exe

This is the same executable as ***perl___.exe***, but it is a PM application.

Note. Usually 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.

This flavor is required if you load extensions which use PM, like the forthcoming `Perl/Tk`.

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.

Why strange names?

Since Perl processes the `#!`−line (cf. *DESCRIPTION*, *Switches*, *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).

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 stupid−but−quick "hard" dynamic linking used by OS/2.

The address tables of DLLs are patched only once, when they are loaded. The addresses of entry points into DLLs are guaranteed to be the same for all programs which use the same DLL, which reduces the amount of runtime patching − once DLL is loaded, its code is read−only.

While this allows some performance advantages, this makes life terrible for developers, since the above scheme makes it impossible for a DLL to be resolved to a symbol in the .EXE file, since this would need a DLL to have different relocations tables for the executables which use it.

However, a Perl extension is forced to use some symbols from the perl executable, say to know how to find the arguments provided on the perl internal evaluation stack. The solution is that the main code of interpreter should be contained in a DLL, and the ***.EXE*** file just loads this DLL into memory and supplies command−arguments.

This *greatly* increases the load time for the application (as well as the number of problems during compilation). Since interpreter is in a DLL, the CRT is basically forced to reside in a DLL as well (otherwise extensions would not be able to use CRT).

Why chimera build?

Current EMX environment does not allow DLLs compiled using Unixish `a.out` format to export symbols for 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, and

open FH, "|−"
open FH, "−|"

opening pipes to itself.

While these operations are not questions of life and death, a lot of useful scripts use them. This forces `a.out`−style compile of ***perl.exe***.

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.

`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.

`PERL_BADLANG`

If 1, perl ignores `setlocale()` failing. May be useful with some strange *locale*s.

`PERL_BADFREE`

If 1, perl would not warn of in case of unwarranted `free()`. May be useful in conjunction with the module DB_File, since Berkeley DB memory handling code is buggy.

`PERL_SH_DIR`

Specific for EMX port. Gives the directory part of the location for ***sh.exe***.

`TMP` or `TEMP`

Specific for EMX port. Used as storage place for temporary files, most notably `−e` scripts.

Evolution

Here we list major changes which could make you by surprise.

Priorities

`setpriority` and `getpriority` are not compatible with earlier ports by Andreas Kaiser. See `"setpriority, getpriority"`.

DLL name mangling

With the release 5.003_01 the dynamically loadable libraries should be rebuilt. In particular, DLLs are now created with the names which contain a checksum, thus allowing workaround for OS/2 scheme of caching DLLs.

Threading

As of release 5.003_01 perl is linked to multithreaded CRT DLL. Perl itself is not multithread−safe, as is not perl `malloc()`. However, extensions may use multiple thread on their own risk.

Needed to compile `Perl/Tk` for XFree86−OS/2 out−of−the−box.

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 *"PERL_SH_DIR"*.

Thus means that you need to get some copy of a ***sh.exe*** as well (I use one from pdksh). The drive F: 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`. Thus 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 *"Prerequisites"*).

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 asyncroneous 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.

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. Unfortunately, it is also quite frivolous with memory usage as well.

Since kitchen−top machines are usually low on memory, perl is compiled with all the possible memory−saving options. This probably makes perl‘s `malloc()` as greedy with memory as the neighbor‘s `malloc()`, but still much quickier. Note that this is true only for a "typical" usage, it is possible that the perl malloc will be worse for some very special usage.

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.)

AUTHOR

Ilya Zakharevich, ilya@math.ohio−state.edu

SEE ALSO

perl(1).

NAME

perlwin32 − Perl under WindowsNT [XXX and perhaps under Windows95]

SYNOPSIS

These are instructions for building Perl under WindowsNT (versions 3.51 or 4.0), using Visual C++.

DESCRIPTION

Before you start, you should glance through the README file found 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.

Make sure you read the *BUGS AND CAVEATS* section 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 should probably also read the README.os2 file, which gives a different set of rules to build a Perl that will work on Win32 platforms. That method will probably enable you to build a more Unix−compatible perl, but you will also need to download and use various other support software described in that file.

This set of instructions is meant to describe a so−called "native" port of Perl to Win32 platforms. The resulting Perl requires no additional software to run (other than what came with your operating system). Currently, this port is only capable of using Microsoft's Visual C++ compiler. The ultimate goal is to support the other major compilers that can be used on the platforms.

Setting Up

- Use the default "cmd" shell that comes with NT. In particular, do *not* use the 4DOS/NT shell. The Makefile has commands that are not compatible with that shell.
- Run the VCVARS32.BAT file usually found somewhere like C:\MSDEV4.2\BIN. This will set your build environment.
- Depending on how you extracted the distribution, you have to make sure all the files are writable by you. The easiest way to make sure of this is to execute:

  ```
  attrib −R *.* /S
  ```

  from the perl toplevel directory. You don't *have* to do this if you used the right tools to extract the files in the standard distribution, but it doesn't hurt to do so.

Building and Installation

- The "win32" directory contains *.mak files for use with the NMAKE that comes with Visual C++ ver. 4.0 and above. If you wish to build perl using Visual C++ versions between 2.0 and 4.0, do the following three additional steps (these three steps are not required if you are using Visual C++ versions 4.0 and above):

  1. Overwrite the *.mak files in the win32 subdirectory with the versions in the win32\VC−2.0 directory. (The only difference in those makefiles is in how the `$(INCLUDE)` variable is handled—VC 2.0 NMAKE does not grok a path list in `$(INCLUDE)).`
  2. Reset your INCLUDE environment variable to the MSVC include directory. For example:

     ```
     set INCLUDE=E:\MSVC20\INCLUDE
     ```

     This must have only one directory (a list of directories will not work). VCVARS32.BAT may put multiple locations in there, which is why this step is required.
  3. Apply the patch found in win32\VC−2.0\vc2.patch, like so:

     ```
     cd win32
     patch −p2 −N < VC−2.0\vc2.patch
     ```

You may have to edit win32\win32.c manually if you don‘t have GNU patch.

- Make sure you are in the "win32" subdirectory under the perl toplevel.
- Type "nmake" while in the "win32" subdirectory. This should build everything. Specifically, it will create perl.exe, perl.dll, and perlglob.exe at the perl toplevel, and various other extension dll‘s under the lib\auto directory. If the make fails for any reason, make sure you have done the previous steps correctly.
- Type "nmake install". This will put the newly built perl and the libraries under C:\PERL. If you want to alter this location, to say, D:\FOO\PERL, you will have to say:

  ```
  nmake install INST_TOP=D:\FOO\PERL
  ```

  instead. To use the Perl you just installed, make sure you set your PATH environment variable to C:\PERL\BIN (or D:\FOO\PERL\BIN).

Testing

Type "nmake test". This will run most of the tests from the testsuite (many tests will be skipped, and some tests will fail). Most failures are due to UNIXisms in the standard perl testsuite.

To get a more detailed breakdown of the tests that failed, say:

```
cd ..\t
.\perl harness
```

This should produce a summary very similar to the following:

```
Failed Test  Status Wstat Total Fail  Failed  List of failed
-------------------------------------------------------------------------------
io/fs.t                      26   16  61.54%  1−5, 7−11, 16−18, 23−25
io/tell.t                    13    1   7.69%  10
lib/anydbm.t                 12    1   8.33%  2
lib/findbin.t                 1    1 100.00%  1
lib/sdbm.t                   12    1   8.33%  2
op/mkdir.t                    7    2  28.57%  3, 7
op/runlevel.t                 8    1  12.50%  4
op/stat.t                    56    3   5.36%  3−4, 20
op/taint.t                   98   20  20.41%  1−6, 14, 16, 19−21, 24, 26, 35−3
pragma/locale.t              98   40  40.82%  1, 13−14, 21−27, 33, 39, 45−53,
Failed 10/149 test scripts, 93.29% okay. 86/3506 subtests failed, 97.55% okay.
```

Check if any additional tests other than the ones shown here failed. The standard testsuite will ultimately be modified so that the testsuite avoids running irrelevant tests on Win32.

BUGS AND CAVEATS

This is still very much an experimental port, and should be considered alpha quality software. You can expect changes in virtually all of these areas: build process, installation structure, supported utilities/modules, and supported perl functionality. Specifically, functionality that supports the Win32 environment may be ultimately be supported as either core modules or extensions.

Many tests from the standard testsuite either fail or produce different results under this port. Most of the problems fall under one of these categories

- `stat()` and `lstat()` functions may not behave as documented. They may return values that bear no resemblance to those reported on Unix platforms, and some fields may be completely bogus.
- The following functions are currently unavailable: `fork()`, `exec()`, `dump()`, `kill()`, `chown()`, `link()`, `symlink()`, `chroot()`, `setpgrp()`, `getpgrp()`, `setpriority()`, `getpriority()`, `syscall()`, `fcntl()`, `flock()`. This list is possibly incomplete.

- Various `socket()` related calls are supported, but they may not behave as on Unix platforms.
- The four−argument `select()` call is only supported on sockets.
- The behavior of `system()` or the `qx[]` operator (a.k.a. "backticks"), when used to call interactive commands, is ill−defined.
- `$!` doesn‘t work reliably yet.
- Building modules available on CPAN is mostly supported, but this hasn‘t been tested much yet. Expect strange problems, and be prepared to deal with the consequences.
- `utime()`, `times()` and process−related functions may not behave as described in the documentation, and some of the returned values or effects may be bogus.
- Signal handling may not behave as on Unix platforms.
- File globbing may not behave as on Unix platforms.
- Not all of the utilities that come with the Perl distribution are supported yet.

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`.

AUTHORS

Gary Ng <***71564.1743@CompuServe.COM***
Gurusamy Sarathy <***gsar@umich.edu***
Nick Ing−Simmons <***nick@ni−s.u−net.com***

SEE ALSO

perl

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.

Nick Ing−Simmons and Gurusamy Sarathy have made numerous and sundry hacks since then.

Last updated: 19 March 1997

NAME

Install − Build and Installation guide for perl5.

SYNOPSIS

The basic steps to build and install perl5 on a Unix system are:

```
rm −f config.sh
sh Configure
make
make test
make install

# You may also wish to add these:
(cd /usr/include && h2ph *.h sys/*.h)
(cd pod && make html && mv *.html <www home dir>)
(cd pod && make tex  && <process the latex files>)
```

Each of these is explained in further detail below.

For information on non−Unix systems, see the section on *"Porting information"* below.

DESCRIPTION

You should probably at least skim through this entire document before proceeding. Special notes specific to this release are identified by **NOTE**.

This document is written in pod format as an easy way to indicate its structure. The pod format is described in pod/perlpod.pod, but you can read it as is with any pager or editor.

If you‘re building Perl on a non−Unix system, you should also read the README file specific to your operating system, since this may provide additional or different instructions for building Perl.

Space Requirements

The complete perl5 source tree takes up about 7 MB of disk space. The complete tree after completing `make` takes roughly 15 MB, though the actual total is likely to be quite system−dependent. The installation directories need something on the order of 7 MB, though again that value is system−dependent.

Start with a Fresh Distribution

If you have built perl before, you should clean out the build directory with the command

```
make realclean
```

The results of a Configure run are stored in the config.sh file. If you are upgrading from a previous version of perl, or if you change systems or compilers or make other significant changes, or if you are experiencing difficulties building perl, you should probably *not* re−use your old config.sh. Simply remove it or rename it, e.g.

```
mv config.sh config.sh.old
```

If you wish to use your old config.sh, be especially attentive to the version and architecture−specific questions and answers. For example, the default directory for architecture−dependent library modules includes the version name. By default, Configure will reuse your old name (e.g. /opt/perl/lib/i86pc−solaris/5.003) even if you‘re running Configure for a different version, e.g. 5.004. Yes, Configure should probably check and correct for this, but it doesn‘t, presently. Similarly, if you used a shared libperl.so (see below) with version numbers, you will probably want to adjust them as well.

Also, be careful to check your architecture name. Some Linux systems call themselves i486, while others use i586. If you pick up a precompiled binary, it might not use the same name.

In short, if you wish to use your old config.sh, I recommend running Configure interactively rather than blindly accepting the defaults.

Run Configure

Configure will figure out various things about your system. Some things Configure will figure out for itself, other things it will ask you about. To accept the default, just press `RETURN`. The default is almost always ok.

After it runs, Configure will perform variable substitution on all the ***.SH** files and offer to run **make depend**.

Configure supports a number of useful options. Run **Configure −h** to get a listing. To compile with gcc, for example, you can run

```
        sh Configure −Dcc=gcc
```

This is the preferred way to specify gcc (or another alternative compiler) so that the hints files can set appropriate defaults.

If you want to use your old config.sh but override some of the items with command line options, you need to use **Configure −O**.

If you are willing to accept all the defaults, and you want terse output, you can run

```
        sh Configure −des
```

By default, for most systems, perl will be installed in /usr/local/{bin, lib, man}. You can specify a different 'prefix' for the default installation directory, when Configure prompts you or by using the Configure command line option −Dprefix='/some/directory', e.g.

```
        sh Configure −Dprefix=/opt/perl
```

If your prefix contains the string "perl", then the directories are simplified. For example, if you use prefix=/opt/perl, then Configure will suggest /opt/perl/lib instead of /opt/perl/lib/perl5/.

By default, Configure will compile perl to use dynamic loading if your system supports it. If you want to force perl to be compiled statically, you can either choose this when Configure prompts you or you can use the Configure command line option −Uusedl.

GNU−style configure

If you prefer the GNU−style **configure** command line interface, you can use the supplied **configure** command, e.g.

```
        CC=gcc ./configure
```

The **configure** script emulates a few of the more common configure options. Try

```
        ./configure −−help
```

for a listing.

Cross compiling is not supported.

For systems that do not distinguish the files "Configure" and "configure", Perl includes a copy of **configure** named **configure.gnu**.

Extensions

By default, Configure will offer to build every extension which appears to be supported. For example, Configure will offer to build GDBM_File only if it is able to find the gdbm library. (See examples below.) DynaLoader, Fcntl, and IO are always built by default. Configure does not contain code to test for POSIX compliance, so POSIX is always built by default as well. If you wish to skip POSIX, you can set the Configure variable useposix=false either in a hint file or from the Configure command line. Similarly, the Opcode extension is always built by default, but you can skip it by setting the Configure variable useopcode=false either in a hint file for from the command line.

Even if you do not have dynamic loading, you must still build the DynaLoader extension; you should just build the stub dl_none.xs version. (Configure will suggest this as the default.)

In summary, here are the Configure command−line variables you can set to turn off each extension:

```
DB_File             i_db
DynaLoader          (Must always be included as a static extension)
Fcntl               (Always included by default)
GDBM_File           i_gdbm
IO                  (Always included by default)
NDBM_File           i_ndbm
ODBM_File           i_dbm
POSIX               useposix
SDBM_File           (Always included by default)
Opcode              useopcode
Socket              d_socket
```

Thus to skip the NDBM_File extension, you can use

```
        sh Configure −Ui_ndbm
```

Again, this is taken care of automatically if you don‘t have the ndbm library.

Of course, you may always run Configure interactively and select only the extensions you want.

Finally, if you have dynamic loading (most modern Unix systems do) remember that these extensions do not increase the size of your perl executable, nor do they impact start−up time, so you probably might as well build all the ones that will work on your system.

Including locally−installed libraries

Perl5 comes with interfaces to number of database extensions, including dbm, ndbm, gdbm, and Berkeley db. For each extension, if Configure can find the appropriate header files and libraries, it will automatically include that extension. The gdbm and db libraries are **not** included with perl. See the library documentation for how to obtain the libraries.

Note: If your database header (.h) files are not in a directory normally searched by your C compiler, then you will need to include the appropriate **−I/your/directory** option when prompted by Configure. If your database library (.a) files are not in a directory normally searched by your C compiler and linker, then you will need to include the appropriate **−L/your/directory** option when prompted by Configure. See the examples below.

Examples

gdbm in /usr/local

Suppose you have gdbm and want Configure to find it and build the GDBM_File extension. This examples assumes you have ***gdbm.h*** installed in ***/usr/local/include/gdbm.h*** and ***libgdbm.a*** installed in ***/usr/local/lib/libgdbm.a***. Configure should figure all the necessary steps out automatically.

Specifically, when Configure prompts you for flags for your C compiler, you should include `−I/usr/local/include`.

When Configure prompts you for linker flags, you should include `−L/usr/local/lib`.

If you are using dynamic loading, then when Configure prompts you for linker flags for dynamic loading, you should again include `−L/usr/local/lib`.

Again, this should all happen automatically. If you want to accept the defaults for all the questions and have Configure print out only terse messages, then you can just run

```
        sh Configure −des
```

and Configure should include the GDBM_File extension automatically.

This should actually work if you have gdbm installed in any of (/usr/local, /opt/local, /usr/gnu,

/opt/gnu, /usr/GNU, or /opt/GNU).

gdbm in /usr/you

Suppose you have gdbm installed in some place other than /usr/local/, but you still want Configure to find it. To be specific, assume you have ***/usr/you/include/gdbm.h*** and ***/usr/you/lib/libgdbm.a***. You still have to add **−I/usr/you/include** to cc flags, but you have to take an extra step to help Configure find ***libgdbm.a***. Specifically, when Configure prompts you for library directories, you have to add ***/usr/you/lib*** to the list.

It is possible to specify this from the command line too (all on one line):

```
        sh Configure −des \
                −Dlocincpth="/usr/you/include" \
                −Dloclibpth="/usr/you/lib"
```

`locincpth` is a space−separated list of include directories to search. Configure will automatically add the appropriate **−I** directives.

`loclibpth` is a space−separated list of library directories to search. Configure will automatically add the appropriate **−L** directives. If you have some libraries under ***/usr/local/*** and others under ***/usr/you***, then you have to include both, namely

```
        sh Configure −des \
                −Dlocincpth="/usr/you/include /usr/local/include" \
                −Dloclibpth="/usr/you/lib /usr/local/lib"
```

Installation Directories

The installation directories can all be changed by answering the appropriate questions in Configure. For convenience, all the installation questions are near the beginning of Configure.

By default, Configure uses the following directories for library files (archname is a string like sun4−sunos, determined by Configure)

```
        /usr/local/lib/perl5/archname/5.004
        /usr/local/lib/perl5/
        /usr/local/lib/perl5/site_perl/archname
        /usr/local/lib/perl5/site_perl
```

and the following directories for manual pages:

```
        /usr/local/man/man1
        /usr/local/lib/perl5/man/man3
```

(Actually, Configure recognizes the SVR3−style /usr/local/man/l_man/man1 directories, if present, and uses those instead.) The module man pages are stuck in that strange spot so that they don‘t collide with other man pages stored in /usr/local/man/man3, and so that Perl‘s man pages don‘t hide system man pages. On some systems, **man less** would end up calling up Perl‘s less.pm module man page, rather than the **less** program.

If you specify a prefix that contains the string "perl", then the directory structure is simplified. For example, if you Configure with −Dprefix=/opt/perl, then the defaults are

```
        /opt/perl/lib/archname/5.004
        /opt/perl/lib
        /opt/perl/lib/site_perl/archname
        /opt/perl/lib/site_perl

        /opt/perl/man/man1
        /opt/perl/man/man3
```

The perl executable will search the libraries in the order given above.

The directories site_perl and site_perl/archname are empty, but are intended to be used for installing local or site−wide extensions. Perl will automatically look in these directories. Previously, most sites just put their local extensions in with the standard distribution.

In order to support using things like #!/usr/local/bin/perl5.004 after a later version is released, architecture−dependent libraries are stored in a version−specific directory, such as /usr/local/lib/perl5/archname/5.004/. In Perl 5.000 and 5.001, these files were just stored in /usr/local/lib/perl5/archname/. If you will not be using 5.001 binaries, you can delete the standard extensions from the /usr/local/lib/perl5/archname/ directory. Locally−added extensions can be moved to the site_perl and site_perl/archname directories.

Again, these are just the defaults, and can be changed as you run Configure.

Changing the installation directory

Configure distinguishes between the directory in which perl (and its associated files) should be installed and the directory in which it will eventually reside. For most sites, these two are the same; for sites that use AFS, this distinction is handled automatically. However, sites that use software such as **depot** to manage software packages may also wish to install perl into a different directory and use that management software to move perl to its final destination. This section describes how to do this. Someday, Configure may support an option −Dinstallprefix=/foo to simplify this.

Suppose you want to install perl under the ***/tmp/perl5*** directory. You can edit ***config.sh*** and change all the install* variables to point to ***/tmp/perl5*** instead of ***/usr/local/wherever***. You could also set them all from the Configure command line. Or, you can automate this process by placing the following lines in a file ***config.over*** **before** you run Configure (replace /tmp/perl5 by a directory of your choice):

```
installprefix=/tmp/perl5
test −d $installprefix || mkdir $installprefix
test −d $installprefix/bin || mkdir $installprefix/bin
installarchlib=‘echo $installarchlib | sed "s!$prefix!$installprefix!"‘
installbin=‘echo $installbin | sed "s!$prefix!$installprefix!"‘
installman1dir=‘echo $installman1dir | sed "s!$prefix!$installprefix!"‘
installman3dir=‘echo $installman3dir | sed "s!$prefix!$installprefix!"‘
installprivlib=‘echo $installprivlib | sed "s!$prefix!$installprefix!"‘
installscript=‘echo $installscript | sed "s!$prefix!$installprefix!"‘
installsitelib=‘echo $installsitelib | sed "s!$prefix!$installprefix!"‘
installsitearch=‘echo $installsitearch | sed "s!$prefix!$installprefix!"‘
```

Then, you can Configure and install in the usual way:

```
sh Configure −des
make
make test
make install
```

Creating an installable tar archive

If you need to install perl on many identical systems, it is convenient to compile it once and create an archive that can be installed on multiple systems. Here‘s one way to do that:

```
# Set up config.over to install perl into a different directory,
# e.g. /tmp/perl5 (see previous part).
sh Configure −des
make
make test
make install
cd /tmp/perl5
tar cvf ../perl5−archive.tar .
# Then, on each machine where you want to install perl,
cd /usr/local  # Or wherever you specified as $prefix
tar xvf perl5−archive.tar
```

Configure−time Options

There are several different ways to Configure and build perl for your system. For most users, the defaults are sensible and will work. Some users, however, may wish to further customize perl. Here are some of the main things you can change.

Binary Compatibility With Earlier Versions of Perl 5

If you have dynamically loaded extensions that you built under perl 5.003 and that you wish to continue to use with perl 5.004, then you need to ensure that 5.004 remains binary compatible with 5.003.

Starting with Perl 5.003, all functions in the Perl C source code have been protected by default by the prefix Perl_ (or perl_) so that you may link with third−party libraries without fear of namespace collisions. This change broke compatibility with version 5.002, so installing 5.003 or 5.004 over 5.002 or earlier will force you to re−build and install all of your dynamically loadable extensions. (The standard extensions supplied with Perl are handled automatically). You can turn off this namespace protection by adding −DNO_EMBED to your ccflags variable in config.sh.

Perl 5.003‘s namespace protection was incomplete, but this has been fixed in 5.004. However, some sites may need to maintain complete binary compatibility with Perl 5.003. If you are building Perl for such a site, then when **Configure** asks if you want binary compatibility, answer "y".

On the other hand, if you are embedding perl into another application and want the maximum namespace protection, then you probably ought to answer "n" when **Configure** asks if you want binary compatibility.

The default answer of "y" to maintain binary compatibility is probably appropriate for almost everyone.

Selecting File IO mechanisms

Previous versions of perl used the standard IO mechanisms as defined in <stdio.h. Versions 5.003_02 and later of perl allow alternate IO mechanisms via a "PerlIO" abstraction, but the stdio mechanism is still the default and is the only supported mechanism.

This PerlIO abstraction can be enabled either on the Configure command line with

```
        sh Configure −Duseperlio
```

or interactively at the appropriate Configure prompt.

If you choose to use the PerlIO abstraction layer, there are two (experimental) possibilities for the underlying IO calls. These have been tested to some extent on some platforms, but are not guaranteed to work everywhere.

1. AT&T‘s "sfio". This has superior performance to <stdio.h in many cases, and is extensible by the use of "discipline" modules. Sfio currently only builds on a subset of the UNIX platforms perl supports. Because the data structures are completely different from stdio, perl extension modules or external libraries may not work. This configuration exists to allow these issues to be worked on.

   This option requires the ‘sfio’ package to have been built and installed. A (fairly old) version of sfio is in CPAN, and work is in progress to make it more easily buildable by adding Configure support.

   You select this option by

   ```
           sh Configure −Duseperlio −Dusesfio
   ```

   If you have already selected −Duseperlio, and if Configure detects that you have sfio, then sfio will be the default suggested by Configure.

   Note: On some systems, sfio‘s **iffe** configuration script fails to detect that you have an `atexit` function (or equivalent). Apparently, this is a problem at least for some versions of Linux and SunOS 4.

   You can test if you have this problem by trying the following shell script. (You may have to add some extra cflags and libraries. A portable version of this may eventually make its way into Configure.)

```
#!/bin/sh
cat > try.c <<'EOCP'
#include <stdio.h>
main() { printf("42\n"); }
EOCP
cc −o try try.c −lsfio
val=‘./try‘
if test X$val = X42; then
    echo "Your sfio looks ok"
else
    echo "Your sfio has the exit problem."
fi
```

If you have this problem, the fix is to go back to your sfio sources and correct iffe‘s guess about atexit (or whatever is appropriate for your platform.)

There also might be a more recent release of Sfio that fixes your problem.

2. Normal stdio IO, but with all IO going through calls to the PerlIO abstraction layer. This configuration can be used to check that perl and extension modules have been correctly converted to use the PerlIO abstraction.

   This configuration should work on all platforms (but might not).

   You select this option via:

   ```
   sh Configure −Duseperlio −Uusesfio
   ```

   If you have already selected −Duseperlio, and if Configure does not detect sfio, then this will be the default suggested by Configure.

Building a shared libperl.so Perl library

Currently, for most systems, the main perl executable is built by linking the "perl library" libperl.a with perlmain.o, your static extensions (usually just DynaLoader.a) and various extra libraries, such as −lm.

On some systems that support dynamic loading, it may be possible to replace libperl.a with a shared libperl.so. If you anticipate building several different perl binaries (e.g. by embedding libperl into different programs, or by using the optional compiler extension), then you might wish to build a shared libperl.so so that all your binaries can share the same library.

The disadvantages are that there may be a significant performance penalty associated with the shared libperl.so, and that the overall mechanism is still rather fragile with respect to different versions and upgrades.

In terms of performance, on my test system (Solaris 2.5_x86) the perl test suite took roughly 15% longer to run with the shared libperl.so. Your system and typical applications may well give quite different results.

The default name for the shared library is typically something like libperl.so.3.2 (for Perl 5.003_02) or libperl.so.302 or simply libperl.so. Configure tries to guess a sensible naming convention based on your C library name. Since the library gets installed in a version−specific architecture−dependent directory, the exact name isn‘t very important anyway, as long as your linker is happy.

For some systems (mostly SVR4), building a shared libperl is required for dynamic loading to work, and hence is already the default.

You can elect to build a shared libperl by

```
sh Configure −Duseshrplib
```

To actually build perl, you must add the current working directory to your LD_LIBRARY_PATH environment variable before running make. You can do this with

```
    LD_LIBRARY_PATH=‘pwd‘:$LD_LIBRARY_PATH; export LD_LIBRARY_PATH
```

for Bourne−style shells, or

```
    setenv LD_LIBRARY_PATH ‘pwd‘
```

for Csh−style shells. You *MUST* do this before running make. Folks running NeXT OPENSTEP must substitute DYLD_LIBRARY_PATH for LD_LIBRARY_PATH above.

There is also an potential problem with the shared perl library if you want to have more than one "flavor" of the same version of perl (e.g. with and without −DDEBUGGING). For example, suppose you build and install a standard Perl 5.004 with a shared library. Then, suppose you try to build Perl 5.004 with −DDEBUGGING enabled, but everything else the same, including all the installation directories. How can you ensure that your newly built perl will link with your newly built libperl.so.4 rather with the installed libperl.so.4? The answer is that you might not be able to. The installation directory is encoded in the perl binary with the LD_RUN_PATH environment variable (or equivalent ld command−line option). On Solaris, you can override that with LD_LIBRARY_PATH; on Linux you can‘t.

The only reliable answer is that you should specify a different directory for the architecture−dependent library for your −DDEBUGGING version of perl. You can do this with by changing all the *archlib* variables in config.sh, namely archlib, archlib_exp, and installarchlib, to point to your new architecture−dependent library.

Malloc Issues

Perl relies heavily on malloc(3) to grow data structures as needed, so perl‘s performance can be noticeably affected by the performance of the malloc function on your system.

The perl source is shipped with a version of malloc that is very fast but somewhat wasteful of space. On the other hand, your system‘s `malloc()` function is probably a bit slower but also a bit more frugal.

For many uses, speed is probably the most important consideration, so the default behavior (for most systems) is to use the malloc supplied with perl. However, if you will be running very large applications (e.g. Tk or PDL) or if your system already has an excellent malloc, or if you are experiencing difficulties with extensions that use third−party libraries that call malloc, then you might wish to use your system‘s malloc. (Or, you might wish to explore the experimental malloc flags discussed below.)

To build without perl‘s malloc, you can use the Configure command

```
        sh Configure −Uusemymalloc
```

or you can answer ‘n’ at the appropriate interactive Configure prompt.

Malloc Performance Flags

If you are using Perl‘s malloc, you may add one or more of the following items to your `cflags` config.sh variable to change its behavior in potentially useful ways. You can find out more about these flags by reading the ***malloc.c*** source. In a future version of perl, these might be enabled by default.

−DDEBUGGING_MSTATS

If `DEBUGGING_MSTATS` is defined, you can extract malloc statistics from the Perl interpreter. The overhead this imposes is not large (perl just twiddles integers at malloc/free/sbrk time). When you run perl with the environment variable `PERL_DEBUG_MSTATS` set to either 1 or 2, the interpreter will dump statistics to stderr at exit time and (with a value of 2) after compilation. If you install the Devel::Peek module you can get the statistics whenever you like by invoking its `mstat()` function.

−DEMERGENCY_SBRK

If `EMERGENCY_SBRK` 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 *perlvar* for more details.

−DPACK_MALLOC

If `PACK_MALLOC` is defined, malloc.c uses a slightly different algorithm for small allocations (up to 64 bytes long). Such small allocations are quite common in typical Perl scripts.

The expected memory savings (with 8−byte alignment in `alignbytes`) is about 20% for typical Perl usage. The expected slowdown due to the additional malloc overhead is in fractions of a percent. (It is hard to measure because of the effect of the saved memory on speed).

−DTWO_POT_OPTIMIZE

If `TWO_POT_OPTIMIZE` is defined, malloc.c uses a slightly different algorithm for large allocations that are close to a power of two (starting with 16K). Such allocations are typical for big hashes and special−purpose scripts, especially image processing. If you will be manipulating very large blocks with sizes close to powers of two, it might be wise to define this macro.

The expected saving of memory is 0−100% (100% in applications which require most memory in such 2**n chunks). The expected slowdown is negligible.

Building a debugging perl

You can run perl scripts under the perl debugger at any time with **perl −d**. If, however, you want to debug perl itself, you probably want to do

```
        sh Configure −Doptimize='−g'
```

This will do two things: First, it will force compilation to use **cc −g** so that you can use your system's debugger on the executable. Second, it will add a `−DDEBUGGING` to your ccflags variable in ***config.sh*** so that you can use **perl −D** to access perl's internal state.

If you are using a shared libperl, see the warnings about multiple versions of perl under *Building a shared libperl.so Perl library*.

Other Compiler Flags

For most users, all of the Configure defaults are fine. However, you can change a number of factors in the way perl is built by adding appropriate **−D** directives to your ccflags variable in config.sh.

For example, you can replace the `rand()` and `srand()` functions in the perl source by any other random number generator by a trick such as the following:

```
        sh Configure −Dccflags='−Drand=random −Dsrand=srandom'
```

or by adding `−Drand=random` and `−Dsrandom=srandom` to your ccflags at the appropriate Configure prompt. (You may also have to adjust Configure's guess for 'randbits' as well.)

What if it doesn't work?

Running Configure Interactively

If Configure runs into trouble, remember that you can always run Configure interactively so that you can check (and correct) its guesses.

All the installation questions have been moved to the top, so you don't have to wait for them. Once you've handled them (and your C compiler and flags) you can type `&−d` at the next Configure prompt and Configure will use the defaults from then on.

If you find yourself trying obscure command line incantations and config.over tricks, I recommend you run Configure interactively instead. You'll probably save yourself time in the long run.

Hint files

The perl distribution includes a number of system−specific hints files in the hints/ directory. If one of them matches your system, Configure will offer to use that hint file.

Several of the hint files contain additional important information. If you have any problems, it is a good idea to read the relevant hint file for further information. See ***hints/solaris_2.sh*** for an extensive example.

•** WHOA THERE!!! ***

Occasionally, Configure makes a wrong guess. For example, on SunOS 4.1.3, Configure incorrectly concludes that tzname[] is in the standard C library. The hint file is set up to correct for this. You will see a message:

```
    *** WHOA THERE!!! ***
        The recommended value for $d_tzname on this machine was "undef"!
        Keep the recommended value? [y]
```

You should always keep the recommended value unless, after reading the relevant section of the hint file, you are sure you want to try overriding it.

If you are re−using an old config.sh, the word "previous" will be used instead of "recommended". Again, you will almost always want to keep the previous value, unless you have changed something on your system.

For example, suppose you have added libgdbm.a to your system and you decide to reconfigure perl to use GDBM_File. When you run Configure again, you will need to add −lgdbm to the list of libraries. Now, Configure will find your gdbm library and will issue a message:

```
    *** WHOA THERE!!! ***
        The previous value for $i_gdbm on this machine was "undef"!
        Keep the previous value? [y]
```

In this case, you do *not* want to keep the previous value, so you should answer 'n'. (You'll also have to manually add GDBM_File to the list of dynamic extensions to build.)

Changing Compilers

If you change compilers or make other significant changes, you should probably *not* re−use your old config.sh. Simply remove it or rename it, e.g. mv config.sh config.sh.old. Then rerun Configure with the options you want to use.

This is a common source of problems. If you change from **cc** to **gcc**, you should almost always remove your old config.sh.

Propagating your changes to config.sh

If you make any changes to ***config.sh***, you should propagate them to all the .SH files by running **sh Configure −S**. You will then have to rebuild by running

```
        make depend
        make
```

config.over

You can also supply a shell script config.over to over−ride Configure's guesses. It will get loaded up at the very end, just before config.sh is created. You have to be careful with this, however, as Configure does no checking that your changes make sense. See the section on
"Changing the installation directory" for an example.

config.h

Many of the system dependencies are contained in ***config.h***. ***Configure*** builds ***config.h*** by running the ***config_h.SH*** script. The values for the variables are taken from ***config.sh***.

If there are any problems, you can edit ***config.h*** directly. Beware, though, that the next time you run **Configure**, your changes will be lost.

cflags

If you have any additional changes to make to the C compiler command line, they can be made in ***cflags.SH***. For instance, to turn off the optimizer on ***toke.c***, find the line in the switch structure for ***toke.c*** and put the command `optimize='−g'` before the `;;`. You can also edit ***cflags*** directly, but beware that your changes will be lost the next time you run **Configure**.

To change the C flags for all the files, edit ***config.sh*** and change either `$ccflags` or `$optimize`, and then re−run **sh Configure −S ; make depend**.

No sh

If you don‘t have sh, you‘ll have to copy the sample file config_H to config.h and edit the config.h to reflect your system‘s peculiarities. You‘ll probably also have to extensively modify the extension building mechanism.

Porting information

Specific information for the OS/2, Plan9, VMS and Win32 ports is in the corresponding subdirectories. Additional information, including a glossary of all those config.sh variables, is in the Porting subdirectory.

Ports for other systems may also be available. You should check out *ports"* for current information on ports to various other operating systems.

make depend

This will look for all the includes. The output is stored in ***makefile***. The only difference between ***Makefile*** and ***makefile*** is the dependencies at the bottom of ***makefile***. If you have to make any changes, you should edit ***makefile***, not ***Makefile*** since the Unix **make** command reads ***makefile*** first. (On non−Unix systems, the output may be stored in a different file. Check the value of `$firstmakefile` in your config.sh if in doubt.)

Configure will offer to do this step for you, so it isn‘t listed explicitly above.

make

This will attempt to make perl in the current directory.

If you can‘t compile successfully, try some of the following ideas. If none of them help, and careful reading of the error message and the relevant manual pages on your system doesn‘t help, you can send a message to either the comp.lang.perl.misc newsgroup or to perlbug@perl.com with an accurate description of your problem. See *"Reporting Problems"* below.

- If you used a hint file, try reading the comments in the hint file for further tips and information.
- If you can successfully build ***miniperl***, but the process crashes during the building of extensions, you should run

```
        make minitest
```

to test your version of miniperl.

locale

If you have any locale−related environment variables set, try unsetting them. I have some reports that some versions of IRIX hang while running **./miniperl configpm** with locales other than the C locale. See the discussion under *make test* below about locales.

- If you get duplicates upon linking for malloc et al, add −DHIDEMYMALLOC or −DEMBEDMYMALLOC to your ccflags variable in config.sh.

varargs

If you get varargs problems with gcc, be sure that gcc is installed correctly. When using gcc, you should probably have i_stdarg=‘define’ and i_varargs=‘undef’ in config.sh. The problem is usually solved by running fixincludes correctly. If you do change config.sh, don‘t forget to propagate your changes (see *"Propagating your changes to config.sh"* below). See also the *"vsprintf"* item below.

- If you get error messages such as the following (the exact line numbers will vary in different versions of perl):

```
    util.c: In function ‘Perl_croak’:
    util.c:962: number of arguments doesn’t match prototype
    proto.h:45: prototype declaration
```

it might well be a symptom of the gcc "varargs problem". See the previous *"varargs"* item.

Solaris and SunOS dynamic loading

If you have problems with dynamic loading using gcc on SunOS or Solaris, and you are using GNU as and GNU ld, you may need to add **−B/bin/** (for SunOS) or **−B/usr/ccs/bin/** (for Solaris) to your `$ccflags, $ldflags,` and `$lddlflags` so that the system‘s versions of as and ld are used. 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.

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 *"Solaris and SunOS dynamic loading"*.

- If you run into dynamic loading problems, check your setting of the LD_LIBRARY_PATH environment variable. If you‘re creating a static Perl library (libperl.a rather than libperl.so) it should build fine with LD_LIBRARY_PATH unset, though that may depend on details of your local set−up.

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).

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 implementation of those functions instead. [Thanks to Tim Bunce for this explanation.]

nm extraction

If Configure seems to be having trouble finding library functions, try not using nm extraction. You can do this from the command line with

```
sh Configure −Uusenm
```

or by answering the nm extraction question interactively. If you have previously run Configure, you should *not* reuse your old config.sh.

vsprintf

If you run into problems with vsprintf in compiling util.c, the problem is probably that Configure failed to detect your system‘s version of `vsprintf()`. Check whether your system has `vprintf()`. (Virtually all modern Unix systems do.) Then, check the variable d_vprintf in config.sh. If your system has vprintf, it should be:

```
d_vprintf=’define’
```

If Configure guessed wrong, it is likely that Configure guessed wrong on a number of other common functions too. You are probably better off re−running Configure without using nm extraction (see previous item).

Optimizer

If you can‘t compile successfully, try turning off your compiler‘s optimizer. Edit config.sh and change the line

```
optimize=’−O’
```

to something like

```
optimize=’ ’
```

then propagate your changes with **sh Configure −S** and rebuild with **make depend; make**.

- If you still can‘t compile successfully, try adding a `−DCRIPPLED_CC` flag. (Just because you get no errors doesn‘t mean it compiled right!) This simplifies some complicated expressions for compilers that get indigestion easily.

Missing functions

If you have missing routines, you probably need to add some library or other, or you need to undefine some feature that Configure thought was there but is defective or incomplete. Look through config.h for likely suspects.

- Some compilers will not compile or optimize the larger files without some extra switches to use larger jump offsets or allocate larger internal tables. You can customize the switches for each file in ***cflags***. It‘s okay to insert rules for specific files into ***makefile*** since a default rule only takes effect in the absence of a specific rule.

Missing dbmclose

SCO prior to 3.2.4 may be missing `dbmclose()`. An upgrade to 3.2.4 that includes libdbm.nfs (which includes `dbmclose()`) may be available.

Note (probably harmless): No library found for −lsomething

If you see such a message during the building of an extension, but the extension passes its tests anyway (see *"make test"* below), then don‘t worry about the warning message. The extension Makefile.PL goes looking for various libraries needed on various systems; few systems will need all the possible libraries listed. For example, a system may have −lcposix or −lposix, but it‘s unlikely to have both, so most users will see warnings for the one they don‘t have. The phrase ‘probably harmless’ is intended to reassure you that nothing unusual is happening, and the build process is continuing.

On the other hand, if you are building GDBM_File and you get the message

```
    Note (probably harmless): No library found for −lgdbm
```

then it‘s likely you‘re going to run into trouble somewhere along the line, since it‘s hard to see how you can use the GDBM_File extension without the −lgdbm library.

It is true that, in principle, Configure could have figured all of this out, but Configure and the extension building process are not quite that tightly coordinated.

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.

db−recno failure on tests 51, 53 and 55

Old versions of the DB library (including the DB library which comes with FreeBSD 2.1) had broken handling of recno databases with modified bval settings. Upgrade your DB library or OS.

- Some additional things that have been reported for either perl4 or perl5:

  Genix may need to use libc rather than libc_s, or #undef VARARGS.

  NCR Tower 32 (OS 2.01.01) may need −W2,−Sl,2000 and #undef MKDIR.

  UTS may need one or more of −DCRIPPLED_CC, **−K** or **−g**, and undef LSTAT.

  If you get syntax errors on ‘(‘, try −DCRIPPLED_CC.

  Machines with half−implemented dbm routines will need to #undef I_ODBM

make test

This will run the regression tests on the perl you just made. If it doesn‘t say "All tests successful" then something went wrong. See the file ***t/README*** in the ***t*** subdirectory. Note that you can‘t run the tests in background if this disables opening of /dev/tty.

If **make test** bombs out, just **cd** to the ***t*** directory and run ***./TEST*** by hand to see if it makes any difference. If individual tests bomb, you can run them by hand, e.g.,

```
        ./perl op/groups.t
```

Another way to get more detailed information about failed tests and individual subtests is to **cd** to the ***t*** directory and run

```
        ./perl harness
```

(this assumes that *most* tests succeed, since ***harness*** uses complicated constructs).

You can also read the individual tests to see if there are any helpful comments that apply to your system.

Note: One possible reason for errors is that some external programs may be broken due to the combination of your environment and the way `make test` exercises them. For example, this may happen if you have one or more of these environment variables set: `LC_ALL LC_CTYPE LC_COLLATE LANG`. In some versions of UNIX, the non−English locales are known to cause programs to exhibit mysterious errors.

If you have any of the above environment variables set, please try

```
        setenv LC_ALL C
```

(for C shell) or

```
        LC_ALL=C;export LC_ALL
```

for Bourne or Korn shell) from the command line and then retry `make test`. If the tests then succeed, you may have a broken program that is confusing the testing. Please run the troublesome test by hand as shown above and see whether you can locate the program. Look for things like: `exec, ‘backquoted command‘, system, open("|...")` or `open("...|")`. All these mean that Perl is trying to run some external program.

make install

This will put perl into the public directory you specified to **Configure**; by default this is ***/usr/local/bin***. It will also try to put the man pages in a reasonable place. It will not nroff the man pages, however. You may need to be root to run **make install**. If you are not root, you must own the directories in question and you should ignore any messages about chown not working.

If you want to see exactly what will happen without installing anything, you can run

```
        ./perl installperl −n
        ./perl installman −n
```

make install will install the following:

```
        perl,
            perl5.nnn   where nnn is the current release number.  This
                        will be a link to perl.
        suidperl,
            sperl5.nnn  If you requested setuid emulation.
        a2p             awk−to−perl translator
        cppstdin        This is used by perl −P, if your cc −E can’t
                        read from stdin.
        c2ph, pstruct   Scripts for handling C structures in header files.
        s2p             sed−to−perl translator
        find2perl       find−to−perl translator
        h2ph            Extract constants and simple macros from C headers
        h2xs            Converts C .h header files to Perl extensions.
        perlbug         Tool to report bugs in Perl.
        perldoc         Tool to read perl’s pod documentation.
        pl2pm           Convert Perl 4 .pl files to Perl 5 .pm modules
```

```
pod2html,       Converters from perl's pod documentation format
pod2latex,      to other useful formats.
pod2man, and
pod2text
splain          Describe Perl warnings and errors

library files   in $privlib and $archlib specified to
                Configure, usually under /usr/local/lib/perl5/.
man pages       in the location specified to Configure, usually
                something like /usr/local/man/man1.
module          in the location specified to Configure, usually
man pages       under /usr/local/lib/perl5/man/man3.
pod/*.pod       in $privlib/pod/.
```

Installperl will also create the library directories `$siteperl` and `$sitearch` listed in config.sh. Usually, these are something like

/usr/local/lib/perl5/site_perl/
/usr/local/lib/perl5/site_perl/`$archname`

where `$archname` is something like sun4−sunos. These directories will be used for installing extensions.

Perl's *.h header files and the libperl.a library are also installed under `$archlib` so that any user may later build new extensions, run the optional Perl compiler, or embed the perl interpreter into another program even if the Perl source is no longer available.

Coexistence with earlier versions of perl5

You can safely install the current version of perl5 and still run scripts under the old binaries for versions 5.003 and later ONLY. Instead of starting your script with #!/usr/local/bin/perl, just start it with #!/usr/local/bin/perl5.003 (or whatever version you want to run.) If you want to retain a version of Perl 5 prior to 5.003, you'll need to install the current version in a separate directory tree, since some of the architecture−independent library files have changed in incompatible ways.

The architecture−dependent files are stored in a version−specific directory (such as ***/usr/local/lib/perl5/sun4−sunos/5.004***) so that they are still accessible. *Note:* Perl 5.000 and 5.001 did not put their architecture−dependent libraries in a version−specific directory. They are simply in ***/usr/local/lib/perl5/`$archname`***. If you will not be using 5.000 or 5.001, you may safely remove those files.

The standard library files in ***/usr/local/lib/perl5*** should be usable by all versions of perl5.

Most extensions will probably not need to be recompiled to use with a newer version of perl. If you do run into problems, and you want to continue to use the old version of perl along with your extension, simply move those extension files to the appropriate version directory, such as ***/usr/local/lib/perl/archname/5.003***. Then Perl 5.003 will find your files in the 5.003 directory, and newer versions of perl will find your newer extension in the site_perl directory.

Some users may prefer to keep all versions of perl in completely separate directories. One convenient way to do this is by using a separate prefix for each version, such as

```
sh Configure −Dprefix=/opt/perl5.004
```

and adding /opt/perl5.004/bin to the shell PATH variable. Such users may also wish to add a symbolic link /usr/local/bin/perl so that scripts can still start with #!/usr/local/bin/perl.

Coexistence with perl4

You can safely install perl5 even if you want to keep perl4 around.

By default, the perl5 libraries go into ***/usr/local/lib/perl5/***, so they don't override the perl4 libraries in ***/usr/local/lib/perl/***.

In your /usr/local/bin directory, you should have a binary named ***perl4.036***. That will not be touched by the perl5 installation process. Most perl4 scripts should run just fine under perl5. However, if you have any scripts that require perl4, you can replace the `#!` line at the top of them by `#!/usr/local/bin/perl4.036` (or whatever the appropriate pathname is). See pod/perltrap.pod for possible problems running perl4 scripts under perl5.

cd /usr/include; h2ph *.h sys/*.h

Some perl scripts need to be able to obtain information from the system header files. This command will convert the most commonly used header files in ***/usr/include*** into files that can be easily interpreted by perl. These files will be placed in the architectural library directory you specified to **Configure**; by default this is ***/usr/local/lib/perl5/ARCH/VERSION***, where **ARCH** is your architecture (such as `sun4-solaris`) and **VERSION** is the version of perl you are building (for example, `5.004`).

Note: Due to differences in the C and perl languages, the conversion of the header files is not perfect. You will probably have to hand−edit some of the converted files to get them to parse correctly. For example, h2ph breaks spectacularly on type casting and certain structures.

cd pod && make html && mv *.html (www home dir)

Some sites may wish to make the documentation in the pod/ directory available in HTML format. Type

```
        cd pod && make html && mv *.html <www home dir>
```

where ***www home dir*** is wherever your site keeps HTML files.

cd pod && make tex && (process the latex files)

Some sites may also wish to make the documentation in the pod/ directory available in TeX format. Type

```
        (cd pod && make tex && <process the latex files>)
```

Reporting Problems

If you have difficulty building perl, and none of the advice in this file helps, and careful reading of the error message and the relevant manual pages on your system doesn‘t help either, then you should send a message to either the comp.lang.perl.misc newsgroup or to perlbug@perl.com with an accurate description of your problem.

Please include the *output* of the **./myconfig** shell script that comes with the distribution. Alternatively, you can use the **perlbug** program that comes with the perl distribution, but you need to have perl compiled and installed before you can use it.

You might also find helpful information in the ***Porting*** directory of the perl distribution.

DOCUMENTATION

Read the manual entries before running perl. The main documentation is in the pod/ subdirectory and should have been installed during the build process. Type **man perl** to get started. Alternatively, you can type **perldoc perl** to use the supplied **perldoc** script. This is sometimes useful for finding things in the library modules.

Under UNIX, you can produce a documentation book in postscript form along with its *Table of Contents* by going to the pod/ subdirectory and running (either):

```
        ./roffitall −groff             # If you have GNU groff installed
        ./roffitall −psroff            # If you have psroff
```

This will leave you with two postscript files ready to be printed. (You may need to fix the roffitall command to use your local troff set−up.)

Note that you must have performed the installation already before running the above, since the script collects the installed files to generate the documentation.

AUTHOR

Andy Dougherty <doughera@lafcol.lafayette.edu, borrowing *very* heavily from the original README by Larry Wall.

LAST MODIFIED

`$Id:` INSTALL,v 1.8 1997/03/21 16:21:53 doughera Released `$`

NAME

perldata − Perl data types

DESCRIPTION

Variable names

Perl has three data structures: scalars, arrays of scalars, and associative arrays of scalars, known as "hashes". Normal arrays are indexed by number, starting with 0. (Negative subscripts count from the end.) Hash arrays are indexed by string.

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. Most often, it consists of 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 `'`, but that‘s deprecated); all but the last are interpreted as names of packages, to locate the namespace in which to look up the final identifier (see *Packages* for details). It‘s possible to substitute for a simple identifier an expression which produces a reference to the value at runtime; this is described in more detail below, and in *perlref*.

There are also special variables whose names don‘t follow these rules, so that they don‘t accidentally collide with one of your normal variables. Strings which 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 which provide windows into the inner working of Perl have names containing punctuation characters (see *perlvar*).

Scalar values are always named with ‘`$`‘, even when referring to a scalar that is part of an array. It works like the English word "the". Thus we have:

```
$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
```

but entire arrays or array slices are denoted by ‘@‘, which works much like the word "these" or "those":

```
@days               # ($days[0], $days[1],... $days[n])
@days[3,4,5]        # same as @days[3..5]
@days{'a','c'}      # same as ($days{'a'},$days{'c'})
```

and entire hashes are denoted by ‘%‘:

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

In addition, subroutines are named with an initial ‘`&`‘, though this is optional when it‘s otherwise unambiguous (just as "do" is often redundant in English). Symbol table entries can be named with an initial ‘*‘, but you don‘t really care about that yet.

Every variable type has its own namespace. You can, without fear of conflict, use the same name for a scalar variable, an array, or a hash (or, for that matter, a filehandle, a subroutine 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 and array 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 an object of that type. For a description of this, see *perlref*.

Names that start with a digit may contain only more digits. Names which do not start with a letter, underscore, or digit 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.)

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: scalar and list. 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 a scalar context for the <STDIN> 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 a list context for <STDIN>, 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 righthand side in a scalar context, while assignment to an array or array slice evaluates the righthand side in a list context. Assignment to a list also evaluates the righthand side in a list context.

User defined subroutines may choose to care whether they are being called in a scalar or list context, but most subroutines do not need to care, because scalars are automatically interpolated into lists. See *wantarray*.

Scalar values

All data in Perl is a scalar or an array of scalars or a hash of scalars. Scalar variables may contain various kinds of singular data, such as numbers, strings, and references. In general, conversion from one form to another is transparent. (A scalar may not contain multiple values, but may contain a reference to an array or hash containing multiple values.) Because of the automatic conversion of scalars, operations, and functions that return scalars don‘t need to care (and, in fact, can‘t care) whether the context is looking for a string or a number.

Scalars aren‘t necessarily one thing or another. There‘s no place to declare a scalar variable to be of type "string", or of type "number", or type "filehandle", or anything else. Perl is a contextually polymorphic language whose scalars can be strings, numbers, or references (which includes objects). While strings and numbers are considered pretty much the same thing for nearly all purposes, references are strongly−typed uncastable pointers with built−in 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.

There are actually two varieties of null scalars: defined and undefined. Undefined null scalars are returned when 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. An undefined null scalar may become defined the first time you use it as if it were defined, but prior to that you can use the `defined()` operator to determine whether the value is defined or not.

To find out whether a given string is a valid non−zero number, it‘s usually enough to test it against both numeric 0 and also lexical "0" (although this will cause **−w** noises). 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‘s usually preferable because otherwise you won‘t treat IEEE notations like `NaN` or `Infinity` properly. At other times you might prefer to use a regular expression to check whether data is numeric. See *perlre* for details on regular expressions.

```
    warn "has nondigits"        if     /\D/;
    warn "not a whole number"   unless /^\d+$/;
    warn "not an integer"       unless /^[+−]?\d+$/
    warn "not a decimal number" unless /^[+−]?\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 `$#days,` as in **csh**. (Actually, it‘s not the length of the array, it‘s the subscript of the last element, because there is (ordinarily) a 0th element.) Assigning to `$#days` changes the length of the array. Shortening an array by this method destroys intervening values. Lengthening an array that was previously shortened *NO LONGER* recovers the values that were in those elements. (It used to in Perl 4, but we had to break this to make sure destructors were called when expected.) You can also gain some measure of efficiency by preextending 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 a named array in a 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.) 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 nothing‘s left to doubt:

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

If you evaluate a hash in a scalar context, it returns a value which is true if and only if the hash contains any key/value pairs. (If there are any key/value pairs, 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 (compiled in) 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.)

Scalar value constructors

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

```
    12345
    12345.67
    .23E−10
    0xffff       # hex
    0377         # octal
    4_294_967_296# underline for legibility
```

String literals are usually delimited by either single or double quotes. They work much like shell quotes: double−quoted string literals are subject to backslash and variable substitution; single−quoted strings are not (except for "\'" and "\\"). The usual Unix backslash rules apply for making characters such as newline, tab, etc., as well as some more exotic forms. See *Quote and Quotelike Operators* for a list.

Octal or hex representations in string literals (e.g. ‘0xffff’) are not automatically converted to their integer representation. The `hex()` and `oct()` functions make these conversions for you. See *hex* and *oct* 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 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 interpreted
    print "The price is $Price.\n";     # interpreted
```

As in some shells, you can put curly brackets around the name to delimit it from following alphanumerics. In fact, an identifier within such curlies is forced to be a string, as is any single identifier within a hash subscript. 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.

Note that a single−quoted string must be separated from a preceding word by a space, because single quote is a valid (though deprecated) character in a variable name (see *Packages*).

Three special literals are __FILE__, __LINE__, and __PACKAGE__, which 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 a `package;` directive), __PACKAGE__ is the undefined value.

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, but may be read via a DATA filehandle: main::DATA for __END__, or PACKNAME::DATA (where PACKNAME is the current package) for __DATA__. The two control characters ^D and ^Z are synonyms for __END__ (or __DATA__ in a module). See *SelfLoader* for more description of __DATA__, and an example of its use.

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 **−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 variables are interpolated into double−quoted strings by joining all the elements of the array with the delimiter specified in the `$"` variable (`$LIST_SEPARATOR` in English), 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 a bad 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 brackets as above.

A line−oriented form of quoting is based on the shell "here−doc" 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. (If you put a space it will be treated as a null identifier, which is valid, and matches the first blank 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 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;
```

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

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

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

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

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

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

contains all the elements of @foo followed by all the elements of @bar, followed by all the elements returned by the subroutine named SomeSub when it's called in a list context. 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.

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

```
    # 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];
```

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);
```

Lists may be assigned to if and only if each element of the list is legal to assign to:

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

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

Array assignment in a 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 very 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.

The final element may be an array or a hash:

```
($a, $b, @rest) = split;
local($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 get a null value. This may be useful in a `local()` or `my()`.

A hash literal contains pairs of values 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 usually 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 which would be a legal identifier. 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* for examples of how to arrange for an output ordering.

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. It also used to be the preferred way to pass filehandles into a function, but now that we have the *foo{THING} notation it isn't often needed for that, either. It is still needed to pass new filehandles into functions (*HANDLE{IO} only works if HANDLE has already been used).

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;
```

This is also a way to create a local filehandle. For example:

```
    sub newopen {
        my $path = shift;
        local *FH;  # not my!
        open (FH, $path) || return undef;
        return *FH;
    }
    $fh = newopen('/etc/passwd');
```

Another way to create local filehandles is with IO::Handle and its ilk, see the bottom of *`open()`*.

See *perlref*, *perlsub*, and *Symbol Tables in perlmod* for more discussion on typeglobs.

NAME

perlsyn − Perl syntax

DESCRIPTION

A Perl script consists of a sequence of declarations and statements. The only things that need to be declared in Perl are report formats and subroutines. See the sections below for more information on those declarations. All uninitialized user−created objects are assumed to start with a null or 0 value until they are defined by some explicit operation such as assignment. (Though you can get warnings about the use of undefined values if you like.) The sequence of statements is executed just once, unlike in **sed** and **awk** scripts, where the sequence of statements is executed for each input line. While this means that you must explicitly loop over the lines of your input file (or files), it also means you have much more control over which files and which lines you look at. (Actually, I'm lying—it is possible to do an implicit loop with either the **−n** or **−p** switch. It's just not the mandatory default like it is in **sed** and **awk**.)

Declarations

Perl is, for the most part, a free−form language. (The only exception to this is format declarations, for obvious reasons.) Comments are indicated by the "#" character, and extend to the end of the line. If you attempt to use `/* */` C−style comments, it will be interpreted either as division or pattern matching, depending on the context, and C++ `//` comments just look like a null regular expression, so don't do that.

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 (prototyped to take one scalar parameter) without defining it by saying just:

```
sub myname ($);
$me = myname $0             or die "can't get myname";
```

Note that it functions as a list operator though, not as a unary operator, so be careful to use `or` instead of `||` there.

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.

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 there 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.

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
```

The `if` and `unless` modifiers have the expected semantics, presuming you‘re a speaker of English. The `while` and `until` modifiers also have the usual "while loop" semantics (conditional evaluated first), except when applied to a do−BLOCK (or to the now−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*. 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 wrap another block around it to do that sort of thing.

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 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 **−w** flag.

If there is a `continue` BLOCK, 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).

Loop Control

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
        ...
    }
```

The `last` command is like the `break` statement in C (as used in loops); it 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 ($line = <ARGV>) {
        chomp($line);
        if ($line =~ s/\\$//) {
            $line .= <ARGV>;
            redo LINE unless eof(); # not eof(ARGV)!
        }
        # now process $line
    }
```

Or here‘s 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;
    }
```

Note that if there were a `continue` block on the above code, it would get executed even on discarded lines.

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 form `while/if BLOCK BLOCK`, available in Perl 4, is no longer available. Replace any occurrence of `if BLOCK` by `if (do BLOCK)`.

For Loops

Perl‘s C−style `for` loop works exactly 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: The first form implies a lexical scope for variables declared with `my` in the initialization expression.)

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
    }
```

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. (Note that a lexically scoped variable can cause problems with you have subroutine or format declarations.)

The `foreach` keyword is actually a synonym for the `for` keyword, so you can use `foreach` for readability or `for` for brevity. If VAR is omitted, `$_` is set to each value. If LIST is an actual array (as opposed to an expression returning a list value), you can modify each element of the array by modifying VAR inside the loop. That‘s because the `foreach` loop index variable is an implicit alias for each item in the list that you‘re looping over.

Examples:

```
    for (@ary) { s/foo/bar/ }

    foreach 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: foreach my $wid (@ary1) {
    INNER:   foreach 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.

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. In addition to the above, 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 normal C comma operator.)

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" }
        elsif  ($flag & O_WRONLY) { ($flag & O_APPEND) ? "a" : "w" }
        elsif  ($flag & O_RDWR)   {
            if ($flag & O_CREAT)  { "w+" }
            else                  { ($flag & O_APPEND) ? "a+" : "r+" }
        }
    };
```

Goto

Although not for the faint of heart, Perl does support a `goto` statement. A loop‘s LABEL is not actually a valid target for a `goto`; it‘s just the name of the loop. There are three forms: goto−LABEL, goto−EXPR, and 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. It can be used to go almost anywhere else within the dynamic scope, including out of subroutines, but it‘s usually better to use some other construct such as last or die. The author of Perl has never felt the need to use this form of goto (in Perl, that is—C is another matter).

The goto−EXPR form expects a label name, whose scope will be resolved dynamically. This allows for computed gotos per FORTRAN, but isn‘t necessarily recommended if you‘re optimizing for maintainability:

```
    goto ("FOO", "BAR", "GLARCH")[$i];
```

The goto−`&NAME` form is highly magical, and substitutes a call to the named subroutine for the currently running subroutine. This is used by `AUTOLOAD()` subroutines that wish to load another subroutine and then pretend that the other subroutine had been called in the first place (except that any modifications to @_ in the current subroutine are propagated to the other subroutine.) After the `goto`, not even `caller()` will be able to tell that this routine was called first.

In almost all cases like this, it‘s usually a far, far better idea to use the structured control flow mechanisms of `next`, `last`, or `redo` instead of resorting to a `goto`. For certain applications, the catch and throw pair of `eval{}` and `die()` for exception processing can also be a prudent approach.

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.

Plain Old Comments (Not!)

Much like the C preprocessor, perl can process line directives. 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 `/^#\s*line\s+(\d+)\s*(?:\s"([^"]*)")?/` with `$1` being the line number for the next line, and `$2` being the optional filename (specified within quotes).

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.
```

NAME

perlop − Perl operators and precedence

SYNOPSIS

Perl operators have the following associativity and precedence, listed from highest precedence to lowest. Note that all 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.

DESCRIPTION

Terms and List Operators (Leftward)

Any TERM is of highest precedence of Perl. These includes 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 look at the left side of operator or the right side of it. 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 the arguments that follow them, and then act like a simple TERM with regard to the preceding expression. Note that you have to 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. 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 *O Operators"*.

The Arrow Operator

Just as in C and C++, "`−>`" is an infix dereference operator. If the right side is either a `[...]` or `{...}` subscript, then the left side must be either a hard or symbolic reference to an array or hash (or a location capable of holding a hard reference, if it‘s an lvalue (assignable)). See *perlref*.

Otherwise, the right side is a method name or a simple scalar variable containing the method name, and the left side must either be an object (a blessed reference) or a class name (that is, a package name). See *perlobj*.

Auto−increment and Auto−decrement

"++" and "—" work as in C. That is, if placed before a variable, they increment or decrement the variable before returning the value, and if placed after, increment or decrement the variable after returning the value.

The auto−increment operator has a little extra built−in 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 null and matches the pattern `/^[a−zA−Z]*[0−9]*$/`, 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’
```

The auto−decrement operator is not magical.

Exponentiation

Binary "**" is the exponentiation operator. Note that 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.)

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. (See also *Integer Arithmetic*.)

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 *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 interpretation.

Binding Operators

Binary "=~" binds a scalar expression to a pattern match. Certain operations search or modify the string `$_` by default. This operator makes that kind of operation work on some other string. The right argument is a search pattern, substitution, or translation. The left argument is what is supposed to be searched, substituted, or translated instead of the default `$_`. The return value indicates the success of the operation. (If the right argument is an expression rather than a search pattern, substitution, or translation, it is interpreted as a search pattern at run time. This can be is less efficient than an explicit search, because the pattern must be compiled every time the expression is evaluated.

Binary "!~" is just like "=~" except the return value is negated in the logical sense.

Multiplicative Operators

Binary "*" multiplies two numbers.

Binary "/" divides two numbers.

Binary "%" computes the modulus of the two numbers.

Binary "x" is the repetition operator. In a scalar context, it returns a string consisting of the left operand repeated the number of times specified by the right operand. In a list context, if the left operand is a list in parentheses, it repeats the list.

```
    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
```

Additive Operators

Binary "+" returns the sum of two numbers.

Binary "−" returns the difference of two numbers.

Binary "." concatenates two strings.

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*.)

Named Unary Operators

The various named unary operators are treated as functions with one argument, with optional parentheses. These include the filetest operators, like `-f`, `-M`, etc. See *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. Examples:

```
    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 ||:

```
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)
```

See also *"Terms and List Operators (Leftward)"*.

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.

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.

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*.

Bitwise And

Binary "&" returns its operators ANDed together bit by bit. (See also *Integer Arithmetic*.)

Bitwise Or and Exclusive Or

Binary "|" returns its operators ORed together bit by bit. (See also *Integer Arithmetic*.)

Binary "^" returns its operators XORed together bit by bit. (See also *Integer Arithmetic*.)

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.

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 differ from C's in that, rather than returning 0 or 1, they return the last value evaluated. Thus, a reasonably portable way to find out the home directory (assuming it's not "0") might be:

```
    $home = $ENV{'HOME'} || $ENV{'LOGDIR'} ||
        (getpwuid($<))[7] || die "You're homeless!\n";
```

As more readable alternatives to && and `||`, 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);
```

Range Operator

Binary ".." is the range operator, which is really two different operators depending on the context. In a list context, it returns an array of values counting (by ones) from the left value to the right value. This is useful for writing `for (1..10)` loops and for doing slice operations on arrays. Be aware that under the current implementation, a temporary array is created, so you'll burn a lot of memory if you write something like this:

```
    for (1 .. 1_000_000) {
        # code
    }
```

In a 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**), use three dots ("...") instead of two.) 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 null 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 numeric literal, that operand is implicitly compared to the `$.` variable, the current line number. Examples:

As a scalar operator:

```
    if (101 .. 200) { print; }  # print 2nd hundred lines
    next line if (1 .. /^$/);   # skip header lines
    s/^/> / if (/^$/ .. eof()); # quote body
```

As a list operator:

```
    for (101 .. 200) { print; } # print $_ 100 times
    @foo = @foo[$[ .. $#foo];   # an expensive no−op
    @foo = @foo[$#foo−4 .. $#foo];      # slice last 5 items
```

The range operator (in a list context) makes use of the magical auto−increment algorithm if the operands are strings. You can say

```
    @alphabet = ('A' .. 'Z');
```

to get all the letters of the 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.

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;
```

This is not necessarily guaranteed to contribute to the readability of your program.

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=
```

Note that while these are grouped by family, they all have the precedence of assignment.

Unlike in C, the 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;
```

Comma Operator

Binary "," is the comma operator. In a 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 a list context, it‘s just the list argument separator, and inserts both its arguments into the list.

The => digraph is mostly just a synonym for the comma operator. It‘s useful for documenting arguments that come in pairs. As of release 5.001, it also forces any word to the left of it to be interpreted as a string.

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)*.

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.

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.

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 means that it short−circuits: i.e., the right expression is evaluated only if the left expression is false.

Binary "xor" returns the exclusive−OR of the two surrounding expressions. It cannot short circuit, of course.

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.

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. Non−bracketing delimiters use the same character fore and aft, but the 4 sorts of brackets (round, angle, square, curly) will all nest.

```
    Customary  Generic     Meaning    Interpolates
        ’’       q{}       Literal        no
        ""      qq{}       Literal        yes
        ‘‘      qx{}       Command        yes
                qw{}      Word list       no
        //       m{}    Pattern match     yes
                 s{}{}   Substitution     yes
                tr{}{}   Translation      no
```

For constructs that do interpolation, variables beginning with "`$`" or "`@`" are interpolated, as are the following sequences:

```
\t          tab              (HT, TAB)
\n          newline          (LF, NL)
\r          return           (CR)
\f          form feed        (FF)
\b          backspace        (BS)
\a          alarm (bell)     (BEL)
\e          escape           (ESC)
\033        octal char
\x1b        hex char
\c[         control char
\l          lowercase next char
\u          uppercase next char
\L          lowercase till \E
\U          uppercase till \E
\E          end case modification
\Q          quote regexp 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*.

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 above, there are no 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.

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.

This usage is vaguely deprecated, and may be removed in some future version of Perl.

m/PATTERN/gimosx
/PATTERN/gimosx

Searches a string for a pattern match, and in a scalar context returns true (1) or false (‘’). 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 which apply when `use locale` is in effect.

Options are:

```
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 Unix path names that contain "/", to avoid LTS (leaning toothpick syndrome).

PATTERN may contain variables, which will be interpolated (and the pattern recompiled) every time the pattern search is evaluated. (Note that `$)` and `$|` might not be 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.

If the PATTERN evaluates to a null string, the last successfully executed regular expression is used instead.

If used in a context that requires a list value, a pattern match 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.) If the match fails, a null array is returned. If the match succeeds, but there were no parentheses, a list value of (1) is returned.

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 a list context, it returns a list of all the substrings matched by all the 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 a scalar context, `m//g` iterates through the string, returning TRUE each time it matches, and FALSE when it eventually runs out of matches. (In other words, it remembers where it left off last time and restarts the search at that point. You can actually find the current match position of a string or set it using the `pos()` function—see *pos*.) Note that you can use this feature to stack `m//g` matches or intermix `m//g` matches with `m/\G.../`.

If you modify the string in any way, the match position is reset to the beginning. Examples:

```
# list context
($one,$five,$fifteen) = (`uptime` =~ /(\d+\.\d+)/g);

# scalar context
$/ = ""; $* = 1;  # $* deprecated in modern perls
while ($paragraph = <>) {
    while ($paragraph =~ /[a-z]['")]*[.!?]+['")]*\s/g) {
        $sentences++;
    }
}
print "$sentences\n";

# using m//g with \G
$_ = "ppooqppq";
while ($i++ < 2) {
    print "1: '";
    print $1 while /(o)/g; print "', pos=", pos, "\n";
    print "2: '";
    print $1 if /\G(q)/;   print "', pos=", pos, "\n";
    print "3: '";
    print $1 while /(p)/g; print "', pos=", pos, "\n";
}
```

The last example should print:

```
    1: 'oo', pos=4
    2: 'q', pos=4
    3: 'pp', pos=7
    1: '', pos=7
    2: 'q', pos=7
    3: '', pos=7
```

Note how `m//g` matches change the value reported by `pos()`, but the non−global match doesn‘t.

A useful idiom for `lex`−like scanners is `/\G.../g`. You can combine several regexps like this to process a string part−by−part, doing different actions depending on which regexp matched. The next regexp would step in at the place the previous one left off.

```
    $_ = <<'EOL';
      $url = new URI::URL "http://www/";   die if $url eq "xXx";
EOL
 LOOP:
   {
     print(" digits"),              redo LOOP if /\G\d+\b[,.;]?\s*/g;
     print(" lowercase"), redo LOOP if /\G[a−z]+\b[,.;]?\s*/g;
     print(" UPPERCASE"),           redo LOOP if /\G[A−Z]+\b[,.;]?\s*/g;
     print(" Capitalized"),         redo LOOP if /\G[A−Z][a−z]+\b[,.;]?\s*/g;
     print(" MiXeD"),               redo LOOP if /\G[A−Za−z]+\b[,.;]?\s*/g;
     print(" alphanumeric"),        redo LOOP if /\G[A−Za−z0−9]+\b[,.;]?\s*/g;
     print(" line−noise"),redo LOOP if /\G[^A−Za−z0−9]+/g;
     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 /(tcl|rexx|python)/;      # :-)
$baz = "\n";                # a one-character string
```

qx/STRING/
‘STRING‘ A string which is interpolated and then 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).`

```
$today = qx{ date };
```

See *O Operators* for more discussion.

qw/STRING/

Returns a list of the words extracted out of STRING, using embedded whitespace as the word delimiters. It is exactly equivalent to

```
split(' ', q/STRING/);
```

Some frequently seen examples:

```
use POSIX qw( setlocale localeconv )
@EXPORT = qw( foo bar baz );
```

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 a scalar variable, an array element, a hash element, or an assignment to one of those, i.e., an lvalue.)

If the delimiter chosen is single quote, no variable 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 a null string, the last successfully executed regular expression is used instead. See *perlre* for further explanation on these. See *perllocale* for discussion of additional considerations which 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 back−ticks 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 interpreter as a full−fledged Perl expression and `eval()`ed right then and there. It is, however, syntax checked at compile−time.

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/;

    $count = ($paragraph =~ s/Mister\b/Mr./g);

    $_ = 'abc123xyz';
    s/\d+/$&*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

    # /e's can even nest;  this will expand
    # simple embedded variables in $_
    s/(\$\w+)/$1/eeg;

    # Delete 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

    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. Here are two common cases:

```
    # put commas in the right places in an integer
    1 while s/(.*\d)(\d\d\d)/$1,$2/g;      # perl4
    1 while s/(\d)(\d\d\d)(?!\d)/$1,$2/g;  # perl5

    # expand tabs to 8−column spacing
    1 while s/\t+/' ' x (length($&)*8 − length($`)%8)/e;
```

tr/SEARCHLIST/REPLACEMENTLIST/cds
y/SEARCHLIST/REPLACEMENTLIST/cds

Translates 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 translated. (The string specified with

=~ must be a scalar variable, an array element, or an assignment to one of those, i.e., an lvalue.) 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/`.

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 translated 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 null, 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 translations are given for a character, only the first one is used:

```
tr/AAA/XYZ/
```

will translate any A to X.

Note that because the translation 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 $@;
```

I/O Operators

There are several I/O operators you should know about. A string is enclosed by back−ticks (grave accents) first undergoes variable substitution just like a double quoted string. It is then interpreted as a command, and the output of that command is the value of the pseudo−literal, like in a shell. In a scalar context, a single string consisting of all the output is returned. In a list context, a list of values is returned, one for each 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 `$?).` Unlike in **csh**, no translation is done on the return data—newlines remain newlines. Unlike in any of the shells, single quotes do not hide variable names in the command from interpretation. To pass a `$` through to the shell you need to hide it with a backslash. The generalized form of back−ticks is `qx//`. (Because back−ticks always undergo shell expansion as well, see *perlsec* for security concerns.)

Evaluating a filehandle in angle brackets yields the next line from that file (newline, if any, included), or `undef` at end of file. Ordinarily you must assign that 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` or `for(;;)` loop, the value is automatically assigned to the variable `$_.` The assigned value is then tested to see if it is defined. (This may seem like an odd thing to you, but you'll use the construct in almost every Perl script you write.) Anyway, the following lines are equivalent to each other:

```
while (defined($_ = <STDIN>)) { print; }
while (<STDIN>) { print; }
for (;<STDIN>;) { print; }
print while defined($_ = <STDIN>);
print while <STDIN>;
```

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. See *`open()`* for details on this.

If a <FILEHANDLE> is used in a context that is looking for a list, a list consisting of all the input lines is returned, one line per list element. It's easy to make a *LARGE* data space this way, so use with care.

The null filehandle <> is special and 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 null, `$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, '−') if $#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 array @ARGV and put the current filename into variable `$ARGV.` 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 if the input were one big happy file. (But see example under `eof()` for how to reset line numbers on each file.)

If you want to set @ARGV to your own list of files, go right ahead. 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 FALSE 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 input from STDIN.

If the string inside the angle brackets is a reference to a scalar variable (e.g., <`$foo>),` then that variable contains the name of the filehandle to input from, or a reference to the same. For example:

```
    $fh = \*STDIN;
    $line = <$fh>;
```

If the string inside angle brackets is not a filehandle or a scalar variable containing a filehandle name or reference, then 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. One level of `$` interpretation is done first, but you can‘t say `<$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.) Example:

```
    while (<*.c>) {
        chmod 0644, $_;
    }
```

is equivalent to

```
    open(FOO, "echo *.c | tr -s ' \t\r\f' '\\012\\012\\012\\012'|");
    while (<FOO>) {
        chop;
        chmod 0644, $_;
    }
```

In fact, it‘s currently implemented that way. (Which means it will not work on filenames with spaces in them unless you have csh(1) on your machine.) Of course, the shortest way to do the above is:

```
    chmod 0644, <*.c>;
```

Because globbing invokes a shell, it‘s often faster to call `readdir()` yourself and do your own `grep()` on the filenames. Furthermore, due to its current implementation of using a shell, the `glob()` routine may get "Arg list too long" errors (unless you‘ve installed tcsh(1L) as ***/bin/csh***).

A glob evaluates its (embedded) argument only when it is starting a new list. All values must be read before it will start over. In a list context this isn‘t important, because you automatically get them all anyway. In a scalar context, however, the operator returns the next value each time it is called, or a FALSE value if you‘ve just run out. Again, FALSE 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.

It 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]);
```

Constant Folding

Like C, Perl does a certain amount of expression evaluation at compile time, whenever it determines that all of the 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 interpretation 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 pre−compute the number that expression represents so that the interpreter won‘t have to.

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 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.

The bitwise operators ("&", "|", "^", "~", "<<", and "") always produce integral results. However, `use integer` still has meaning for them. By default, their results are interpreted as unsigned integers. However, 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.

Floating−point Arithmetic

While `use integer` provides integer−only arithmetic, there is no similar ways to provide rounding or truncation at a certain number of decimal places. For rounding to a certain number of digits, `sprintf()` or `printf()` is usually the easiest route.

The POSIX module (part of the standard perl distribution) implements `ceil()`, `floor()`, and a number of other mathematical and trigonometric functions. The Math::Complex module (part of the standard perl distribution) defines a number of mathematical functions that can also work on real 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.

NAME

perlre − Perl regular expressions

DESCRIPTION

This page describes the syntax of regular expressions in Perl. For a description of how to *use* regular expressions in matching operations, plus various examples of the same, see `m//` and `s///` in *perlop*.

The matching operations can have various modifiers. The modifiers which relate to the interpretation of the regular expression inside are listed below. For the modifiers that alter the behaviour of the operation, see *m// in perlop* and *s// 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 at only the very start or end of the string to 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 it normally would not match.

`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 actually be a slash. In fact, any of these modifiers may also be embedded within the regular expression itself using the new `(?...)` 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 meta−character introducing a comment, just as in ordinary Perl code. This also means that if you want real whitespace or # characters in the pattern that you‘ll have to either 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. See the C comment deletion code in *perlop*.

Regular Expressions

The patterns used in pattern matching are regular expressions such as those supplied in the Version 8 regexp routines. (In fact, 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 meta−character
^    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 at only the beginning of the string, the "`$`" character at only the end (or before the newline at the end) and Perl does certain optimizations with the assumption that the string contains only one line. Embedded newlines will not be matched by "^" or "`$`". 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 facilitate 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.) 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 65536.

By default, a quantified sub−pattern is "greedy", that is, it will match as many times as possible without causing the rest of the pattern not to match. The standard quantifiers are all "greedy", in that they match as many occurrences as possible (given a particular starting location) without causing the pattern to fail. If you want it to match the minimum number of times possible, follow the quantifier with a "?" after any of them. Note that the meanings don‘t change, just the "gravity":

```
*?     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
\c[         control 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 regexp 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*.

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
```

Note that `\w` matches a single alphanumeric character, not a whole word. To match a word you'd need to say `\w+`. 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 (though not as either end of a range).

Perl defines the following zero−width assertions:

```
    \b  Match a word boundary
    \B  Match a non−(word boundary)
    \A  Match at only beginning of string
    \Z  Match at only end of string (or before newline at the end)
    \G  Match only where previous m//g left off
```

A word boundary (`\b`) is defined as 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.) 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, not ignoring newline, you can use `\Z(?!\n)`. The `\G` assertion can be used to mix global matches (using `m//g`) and non−global ones, as described in *Regexp Quote−Like Operators in perlop*. It is also useful when writing `lex`−like scanners, when you have several regexps which 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*.

When the bracketing construct `( ... )` is used, \<digit> matches the digit'th substring. Outside of the pattern, always use "`$`" instead of "\" in front of the digit. (While the \<digit> notation can on rare occasion work outside the current pattern, this should not be relied upon. See the WARNING below.) The scope of `$<digit>` (and `$‘, $&, and $’)` extends to the end of the enclosing BLOCK or eval string, or to the next successful pattern match, whichever comes first. If you want to use parentheses to delimit a subpattern (e.g., a set of alternatives) without saving it as a subpattern, follow the (with a ?:.

You may have as many parentheses as you wish. If you have more than 9 substrings, the variables `$10, $11,` ... refer to the corresponding substring. Within the pattern, \10, \11, etc. refer back to substrings if there have been at least that many left parentheses before the backreference. Otherwise (for backward compatibility) \10 is the same as \010, a backspace, and \11 the same as \011, a tab. And so on. (\1 through \9 are always backreferences.)

`$+` returns whatever the last bracket match matched. `$&` returns the entire matched string. (`$0` used to return the same thing, but not any more.) `$‘` returns everything before the matched string. `$’` returns everything after the matched string. Examples:

```
    s/^([^ ]*) *([^ ]*)/$2 $1/;     # swap first two words

    if (/Time: (..):(..):(..)/) {
        $hours = $1;
        $minutes = $2;
        $seconds = $3;
    }
```

Once perl sees that you need one of `$&, $‘` or `$’` anywhere in the program, it has to provide them on each and every pattern match. This can slow your program down. The same mechanism that handles these provides for the use of `$1, $2,` etc., so you pay the same price for each regexp that contains capturing parentheses. But if you never use `$&,` etc., in your script, then regexps *without* capturing parentheses won't 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.

You will note that all 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 meta−character. This makes it simple to quote a string that you want to use for a pattern but that you are afraid might contain metacharacters. Quote simply all the non−alphanumeric characters:

```
    $pattern =~ s/(\W)/\\$1/g;
```

You can also use the built−in `quotemeta()` function to do this. An even easier way to quote metacharacters right in the match operator is to say

```
    /$unquoted\Q$quoted\E$unquoted/
```

Perl defines a consistent extension syntax for regular expressions. The syntax is a pair of parentheses with a question mark as the first thing within the parentheses (this was a syntax error in older versions of Perl). The character after the question mark gives the function of the extension. Several extensions are already supported:

(?#text)
A comment. The text is ignored. If the `/x` switch is used to enable whitespace formatting, a simple # will suffice.

(?:regexp)
This groups things like "`()`" but doesn't make backreferences like "`()`" does. So

```
    split(/\b(?:a|b|c)\b/)
```

is like

```
    split(/\b(a|b|c)\b/)
```

but doesn't spit out extra fields.

(?=regexp)
A zero−width positive lookahead assertion. For example, `/\w+(?=\t)/` matches a word followed by a tab, without including the tab in `$&.`

(?!regexp)
A zero−width negative lookahead assertion. For example `/foo(?!bar)/` matches any occurrence of "foo" that isn't followed by "bar". Note however that lookahead and lookbehind are NOT the same thing. You cannot use this for lookbehind: `/(?!foo)bar/` will not find an occurrence of "bar" that is preceded by something which is not "foo". 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)...|^..?)bar/`. Sometimes it's still easier just to say:

```
    if (/foo/ && $‘ =~ /bar$/)
```

(?imsx)
One or more embedded pattern−match modifiers. This is particularly useful for patterns that are specified in a table somewhere, some of which want to be case sensitive, and some of which don't. 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/ )
```

The specific choice of question mark for this and the new minimal matching construct was because 1) question mark is pretty rare in older regular expressions, and 2) whenever you see one, you should stop and "question" exactly what is going on. That's psychology...

Backtracking

A fundamental feature of regular expression matching involves the notion called *backtracking*. which is used (when needed) by all regular expression quantifiers, namely `*`, `*?`, `+`, `+?`, `{n,m}`, and `{n,m}?`.

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". In this case, 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 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 lookahead assertions and negations, this can all get even tricker. 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 it 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, of course, 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.

Well now, the pattern really, *really* wants to succeed, so it uses the standard regexp 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 in fact "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 by a digit, and in fact, it must also be followed by something that‘s not "123". Remember that the lookaheads 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 like they‘re ANDed together, just as you‘d use any builtin 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.

One warning: particularly complicated regular expressions can take exponential time to solve due to the immense number of possible ways they can use backtracking to try match. For example this will take a very long time to run

```
    /((a{0,5}){0,5}){0,5}/
```

And if you used `*`‘s instead of limiting it to 0 through 5 matches, then it would take literally forever—or until you ran out of stack space.

Version 8 Regular Expressions

In case you‘re not familiar with the "regular" Version 8 regexp routines, here are the pattern−matching rules not described above.

Any single character matches itself, unless it is a *meta−character* with a special meaning described here or above. You can cause characters which 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 of the characters in the list. If the first character after the "[" is "^", the class matches any character not in the list. Within a list, the "−" character is used to specify a range, so that `a−z` represents all the characters between "a" and "z", inclusive.

Characters may be specified using a meta−character 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 ASCII value is *nnn*. Similarly, \x*nn*, where *nn* are hexadecimal digits, matches the character whose ASCII value is *nn*. The expression \c*x* matches the ASCII character control−*x*. Finally, the "." meta−character 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`). Note that 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. For this reason, it‘s common practice to include alternatives in parentheses, to minimize confusion about where they start and end. Note however that "|" is interpreted as a literal with square brackets, so if you write `[fee|fie|foe]` you‘re really only matching `[feio|]`.

Within a pattern, you may designate sub−patterns for later reference by enclosing them in parentheses, and you may refer back to the *n*th sub−pattern later in the pattern using the meta−character *n*. Sub−patterns are numbered based on the left to right order of their opening parenthesis. Note that a backreference matches whatever actually matched the sub−pattern in the string being examined, not the rules for that sub−pattern. Therefore, `(0|0x)\d*\s\1\d*` will match "0x1234 0x4321",but not "0x1234 01234", because sub−pattern 1 actually matched "0x", even though the rule `0|0x` could potentially match the leading 0 in the second number.

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 a `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;
```

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.` Basically, 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///`.

NAME

perlrun − how to execute the Perl interpreter

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*]...

DESCRIPTION

Upon startup, Perl looks for your script 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.)
3. Passed in implicitly via standard input. This works only if there are no filename arguments—to pass arguments to a STDIN script you must explicitly specify a "−" for the script 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 script embedded in a larger message. (In this case you would indicate the end of the script 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 script.

Because many operating systems silently chop 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 script. And a partial **−I** switch could also cause odd results.

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 /usr/bin/perl $0 -S ${1+"$@"}'
    if $running_under_some_shell;
```

to let Perl see the **−p** switch.

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 script, Perl compiles the entire script to an internal form. If there are any compilation errors, execution of the script is not attempted. (This is unlike the typical shell script, which might run partway through before finding a syntax error.)

If the script is syntactically correct, it is executed. If the script runs off the end without hitting an `exit()` or `die()` operator, an implicit `exit(0)` is provided to indicate successful completion.

#! 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).

DOS

Create a batch file to run your script, and codify it in `ALTERNATIVE_SHEBANG` (see the ***dosish.h*** file in the source distribution for more information).

Win95/NT

The Win95/NT installation, when using the Activeware port of Perl, will modify the Registry to associate the .pl extension with the perl interpreter. If you install another port of Perl, including the one in the win32 directory of the Perl distribution, then you‘ll have to modify the Registry yourself.

Macintosh

Macintosh perl scripts will have the the appropriate Creator and Type, so that double−clicking them will invoke the perl application.

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 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)

# VMS
perl −e "print ""Hello world\n"""
```

The problem is that none of this is reliable: it depends on the command tirely possible neither works. If 4DOS was 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 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.

There is no general solution to all of this. It‘s just a mess.

Switches

A single−character switch may be combined with the following switch, if any.

```
    #!/usr/bin/perl −spi.bak    # same as −s −p −i.bak
```

Switches include:

−0[*digits*]

specifies the input record separator (`$/`) as an octal 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 ’*.bak’ −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 character with that value.

−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 causes Perl to check the syntax of the script and then exit without executing it. Actually, it *will* execute `BEGIN`, `END`, and `use` blocks, because these are considered as occurring outside the execution of your program.

−d runs the script under the Perl debugger. See *perldebug*.

−d:*foo*

runs the script under the control of a debugging or tracing module installed as Devel::foo. E.g., **−d:DProf** executes the script using the Devel::DProf profiler. See *perldebug*.

−D*number*
−D*list*

sets debugging flags. To watch how it executes your script, use **−D14**. (This works only if debugging is compiled into your Perl.) Another nice value is **−D1024**, which lists your compiled syntax tree. And **−D512** displays compiled regular expressions. As an alternative specify a list of letters instead of numbers (e.g., **−D14** is equivalent to **−Dtls**):

```
        1  p  Tokenizing and Parsing
        2  s  Stack Snapshots
        4  l  Label Stack Processing
        8  t  Trace Execution
       16  o  Operator Node Construction
       32  c  String/Numeric Conversions
       64  P  Print Preprocessor Command for −P
      128  m  Memory Allocation
      256  f  Format Processing
      512  r  Regular Expression Parsing
     1024  x  Syntax Tree Dump
     2048  u  Tainting Checks
     4096  L  Memory Leaks (not supported anymore)
```

```
         8192  H  Hash Dump -- usurps values()
        16384  X  Scratchpad Allocation
        32768  D  Cleaning Up
```

−e *commandline*

may be used to enter one line of script. If **−e** is given, Perl will not look for a script 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 added to the name of the old file to make a backup copy. If no extension is supplied, no backup is made. From the shell, saying

```
    $ perl −p −i.bak −e "s/foo/bar/; ... "
```

is the same as using the script:

```
    #!/usr/bin/perl −pi.bak
    s/foo/bar/;
```

which is equivalent to

```
    #!/usr/bin/perl
    while (<>) {
        if ($ARGV ne $oldargv) {
            rename($ARGV, $ARGV . '.bak');
            open(ARGVOUT, ">$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.

You can use `eof` without parenthesis 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*).

−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 effects: first, it automatically chomps `"$/"` (the input record separator) when used with **−n** or **−p**, and second, it assigns `"$\"` (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 script.

`-M`*module* executes `use` *module* `;` before executing your script. You can use quotes to add extra code after the module name, e.g., `-M'module qw(foo bar)'`.

If the first character after the `-M` or `-m` is a dash (`-`) then the 'use' is replaced with 'no'.

A little built−in syntactic sugar means you can also say `-mmodule=foo,bar` or `-Mmodule=foo,bar` as a shortcut for `-M'module qw(foo bar)'`. This avoids the need to use quotes when importing symbols. The actual code generated by `-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 script, which makes it iterate over filename arguments somewhat like **sed −n** or **awk**:

```
    while (<>) {
        ...             # your script goes here
    }
```

Note that the lines are not printed by default. See **−p** to have lines printed. Here is an efficient way to delete all files older than 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.

`BEGIN` and `END` blocks may be used to capture control before or after the implicit loop, just as in **awk**.

−p causes Perl to assume the following loop around your script, which makes it iterate over filename arguments somewhat like **sed**:

```
    while (<>) {
        ...             # your script goes here
    } continue {
        print;
    }
```

Note that the lines are printed automatically. 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 causes your script 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".)

−s enables some rudimentary switch parsing for switches on the command line after the script name but before any filename arguments (or before a **−−**). Any switch found there is removed from @ARGV and sets the corresponding variable in the Perl script. The following script prints "true" if and only if the script is invoked with a **−xyz** switch.

```
#!/usr/bin/perl −s
if ($xyz) { print "true\n"; }
```

−S makes Perl use the PATH environment variable to search for the script (unless the name of the script starts with a slash). Typically this is used to emulate #! startup on machines that don‘t support #!, in the following manner:

```
#!/usr/bin/perl
eval ’exec /usr/bin/perl −S $0 ${1+"$@"}’
        if $running_under_some_shell;
```

The system ignores the first line and feeds the script to /bin/sh, which proceeds to try to execute the Perl script 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 `$0` doesn‘t always contain the full pathname, so the **−S** tells Perl to search for the script if necessary. After Perl locates the script, it parses the lines and ignores them because the variable `$running_under_some_shell` is never true. A better construct than `$*` would be `${1+"$@"}`, which handles embedded spaces and such in the filenames, but doesn‘t work if the script is being interpreted by csh. 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 /usr/bin/perl −S $0 ${1+"$@"}’
    & eval ’exec /usr/bin/perl −S $0 $argv:q’
            if $running_under_some_shell;
```

−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 run on another‘s behalf, such as CGI programs. See *perlsec*.

−u causes Perl to dump core after compiling your script. You can then 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 script before dumping, use the `dump()` operator instead. Note: availability of **undump** is platform specific and may not be available for a specific port of Perl.

−U allows Perl to do unsafe operations. Currently the only "unsafe" operations are the unlinking of directories while running as superuser, and running setuid programs with fatal taint checks turned into warnings.

−v prints the version and patchlevel of your Perl executable.

−V prints summary of the major perl configuration values and the current value of @INC.

−V:*name*

Prints to STDOUT the value of the named configuration variable.

−w prints warnings about variable names that are mentioned only once, and scalar variables that are used before being set. Also warns about redefined subroutines, and references to undefined filehandles or filehandles opened read−only that you are attempting to write on. Also warns you if you use values 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.

You can disable specific warnings using `__WARN__` hooks, as described in *perlvar* and *warn*. See also *perldiag* and *perltrap*.

−x *directory*

tells Perl that the script is embedded in a 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 script. The **−x** switch controls only the disposal of leading garbage. The script must be terminated with `__END__` if there is trailing garbage to be ignored (the script can process any or all of the trailing garbage via the DATA filehandle if desired).

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 script if **−S** is used.

PERL5LIB
A colon−separated list of directories in which to look for Perl library files before looking in the standard library and the current directory. If PERL5LIB is not defined, PERLLIB is used. When running taint checks (because the script was running setuid or setgid, or the **−T** switch was used), neither variable is used. The script should instead say

```
    use lib "/my/directory";
```

PERLLIB
A colon−separated 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' }
```

PERL_DEBUG_MSTATS
Relevant only if your perl executable was built with **−DDEBUGGING_MSTATS**, 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.

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 script being executed, and to child processes. However, scripts 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 defined $ENV{'SHELL'};
    $ENV{'IFS'} = ''          if defined $ENV{'IFS'};
```

NAME

perlfunc − Perl builtin functions

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 and 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 list argument.) For instance, `splice()` has three scalar arguments followed by a list.

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.
```

For functions that can be used in either a scalar or list context, non−abortive 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 rule:

THERE IS NO GENERAL RULE FOR CONVERTING A LIST INTO A SCALAR!

Each operator and function decides which sort of value it would be most appropriate to return in a scalar context. Some operators return the length of the list that would have been returned in a 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.

Perl Functions by Category

Here are Perl‘s functions (including things that look like functions, like some of the 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

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, 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, 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, package, use

Miscellaneous functions

defined, dump, eval, formline, local, my, 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, 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

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.
−s  File has non−zero size (returns size).

−f  File is a plain file.
−d  File is a directory.
−l  File is a symbolic link.
−p  File is a named pipe (FIFO).
−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 a text file.
−B  File is a binary file (opposite of −T).

−M  Age of file in days when script started.
−A  Same for access time.
−C  Same for inode change time.
```

The interpretation of the file permission operators `−r`, `−R`, `−w`, `−W`, `−x`, and `−X` is 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. Also note that, for the superuser, `−r`, `−R`, `−w`, and `−W` 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 the uid to something else.

Example:

```
while (<>) {
    chop;
    next unless -f $_;      # ignore specials
    ...
}
```

Note that `-s/a/b/` does not do a negated substitution. Saying `-exp($foo)` still works as expected, however—only single letters following a minus are interpreted as file tests.

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 odd 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 stdio 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.) 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 example in *Sockets: Client/Server Communication in perlipc*.

alarm SECONDS

alarm Arranges to have a SIGALRM delivered to this process after the specified number of 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 than you specified because of how seconds are counted.) 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 `syscall()` interface to access setitimer(2) if your system supports it, or else see *`/select()`* below. It is not advised to intermix `alarm()` and `sleep()` calls.

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.

```
    eval {
        local $SIG{ALRM} = sub { die "alarm\n" };       # NB \n required
        alarm $timeout;
        $nread = sysread SOCKET, $buffer, $size;
        alarm 0;
    };
    die if $@ && $@ ne "alarm\n";       # propagate errors
    if ($@) {
        # timed out
    }
    else {
        # didn't
    }
```

atan2 Y,X

Returns the arctangent of Y/X in the range −PI to PI.

For the tangent operation, you may use the `POSIX::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

Arranges for the file to be read or written in "binary" mode in operating systems that distinguish between binary and text files. Files that are not in binary mode have CR LF sequences translated to LF on input and LF translated to CR LF on output. Binmode has no effect under Unix; in DOS and similarly archaic systems, it may be imperative—otherwise your DOS−damaged C library may mangle your file. The key distinction between systems that need binmode and those that don‘t is their text file formats. Systems like Unix and Plan9 that delimit lines with a single character, and that encode that character in C as ‘\n‘, do not need `binmode`. The rest need it. If FILEHANDLE is an expression, the value is taken as the name of the filehandle.

bless REF,CLASSNAME
bless REF

This function tells the thingy referenced by REF that it is now an object in the CLASSNAME package—or the current package if no CLASSNAME is specified, which is often the case. It returns the reference for convenience, because a `bless()` is often the last thing in a constructor. Always use the two−argument version if the function doing the blessing might be inherited by a derived class. See *perlobj* for more about the blessing (and blessings) of objects.

caller EXPR
caller

Returns the context of the current subroutine call. In a 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 a 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, $is_require) = caller($i);
```

Here `$subroutine` may be `"(eval)"` if the frame is not a subroutine call, but *eval*. In such a case additional elements `$evaltext` and `$is_require` are set: `$is_require` is true if the frame is created by *require* or *use* statement, `$evaltext` contains the text of *eval EXPR* statement. In particular, for *eval BLOCK* statement `$filename` is `"(eval)"`, but `$evaltext` is undefined. (Note also that *use* statement creates a *require* frame inside an *eval EXPR*) frame.

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 that subroutine was invoked.

chdir EXPR

Changes the working directory to EXPR, if possible. If EXPR is omitted, changes to home directory. Returns TRUE upon success, FALSE otherwise. See 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
```

chomp VARIABLE
chomp LIST
chomp

This is a slightly safer version of chop (see below). It removes any line ending 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 (`$/ = ""`), it removes all trailing newlines from the string. If VARIABLE is omitted, it chomps `$_.` Example:

```
while (<>) {
    chomp;  # avoid \n on last field
    @array = split(/:/);
    ...
}
```

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.

chop VARIABLE
chop LIST
chop

Chops off the last character of a string and returns the character chopped. It's used primarily to remove the newline from the end of an input record, but is much more efficient than `s/\n//` because it neither scans nor copies the string. If VARIABLE is omitted, chops `$_.` Example:

```
    while (<>) {
        chop;   # avoid \n on last field
        @array = split(/:/);
        ...
    }
```

You can actually chop anything that‘s an lvalue, including an assignment:

```
    chop($cwd = ‘pwd‘);
    chop($answer = <STDIN>);
```

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)`.

chown LIST

Changes the owner (and group) of a list of files. The first two elements of the list must be the *NUMERICAL* uid and gid, in that order. Returns the number of files successfully changed.

```
    $cnt = chown $uid, $gid, ’foo’, ’bar’;
    chown $uid, $gid, @filenames;
```

Here‘s an example that looks up non−numeric uids in the passwd file:

```
    print "User: ";
    chop($user = <STDIN>);
    print "Files: "
    chop($pattern = <STDIN>);

    ($login,$pass,$uid,$gid) = getpwnam($user)
        or die "$user not in passwd file";

    @ary = <${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.

chr NUMBER

chr Returns the character represented by that NUMBER in the character set. For example, `chr(65)` is "A" in ASCII. For the reverse, use *ord*.

If NUMBER is omitted, uses `$_`.

chroot FILENAME

chroot This function works as 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 of 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 chroot to `$_`.

close FILEHANDLE

Closes the file or pipe associated with the file handle, returning TRUE only if stdio successfully flushes buffers and closes the system file descriptor. 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 (`$.`), while the implicit close done by `open()` does not. Also, closing a pipe will wait for the process executing on the pipe to complete, in case you want to look at the output of the pipe afterwards. Closing a pipe explicitly also puts the status value of the command into `$?`. Example:

```
    open(OUTPUT, ’|sort >foo’); # pipe to sort
    ...                            # print stuff to output
    close OUTPUT;       # wait for sort to finish
    open(INPUT, 'foo');            # get sort's results
```

FILEHANDLE may be an expression whose value gives the real filehandle name.

closedir DIRHANDLE

Closes a directory opened by `opendir()`.

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).

cos EXPR

Returns the cosine of EXPR (expressed in radians). If EXPR is omitted takes cosine of `$_.`

For the inverse cosine operation, you may use the `POSIX::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. As a result, this function isn‘t all that useful for cryptography. (For that, see your nearby CPAN mirror.)

Here‘s an example that makes sure that whoever runs this program knows their own password:

```
    $pwd = (getpwuid($<))[1];
    $salt = substr($pwd, 0, 2);

    system "stty −echo";
    print "Password: ";
    chop($word = <STDIN>);
    print "\n";
    system "stty echo";

    if (crypt($word, $salt) ne $pwd) {
        die "Sorry...\n";
    } else {
        print "ok\n";
    }
```

Of course, typing in your own password to whomever asks you for it is unwise.

dbmclose HASH

[This function has been superseded by the `untie()` function.]

Breaks the binding between a DBM file and a hash.

dbmopen HASH,DBNAME,MODE

[This function has been superseded by the `tie()` function.]

This binds a dbm(3), ndbm(3), sdbm(3), `gdbm()`, 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 MODE (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 array values 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.

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()` to check whether a subroutine exists. On the other hand, use of `defined()` upon aggregates (hashes and arrays) is not guaranteed to produce intuitive results, and should probably be avoided.

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 0 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.

Currently, using `defined()` on an entire array or hash reports whether memory for that aggregate has ever been allocated. So an array you set to the empty list appears undefined initially, and one that once was full and that you then set to the empty list still appears defined. 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"   }
```

Using `undef()` on these, however, does clear their memory and then report them as not defined anymore, but you shoudln‘t do that unless you don‘t plan to use them again, because it saves time when you load them up again to have memory already ready to be filled.

This counter−intuitive behaviour of `defined()` on aggregates may be changed, fixed, or broken in a future release of Perl.

See also *undef*, *exists*, *ref*.

delete EXPR

Deletes the specified key(s) and their associated values from a hash. For each key, returns the deleted value associated with that key, or the undefined value if there was no such key. Deleting from `$ENV{}` modifies the environment. Deleting from a hash tied to a DBM file deletes the entry from the DBM file. (But deleting from a `tie()`d hash doesn‘t necessarily return anything.)

The following deletes all the values of a hash:

```
foreach $key (keys %HASH) {
    delete $HASH{$key};
}
```

And so does this:

```
delete @HASH{keys %HASH}
```

(But both of these are slower than the `undef()` command.) Note that the EXPR can be arbitrarily complicated as long as the final operation is a hash element lookup or hash slice:

```
delete $ref−>[$x][$y]{$key};
delete @{$ref−>[$x][$y]}{$key1, $key2, @morekeys};
```

die LIST

Outside of 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 `($? >> 8)` (back−tick ‘command‘ status). If `($? >> 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 value of EXPR 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. 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()` and `warn()`.

You can arrange for a callback to be called 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 *perlvar* for details on setting `%SIG` entries, and `eval()` for some examples.

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 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, more concise, keeps track of the current filename for error messages, and searches all the **−I** libraries if the file isn‘t in the current directory (see also the @INC array in *Predefined Names*). It‘s the same, however, in that it does re−parse the file every time you call it, so you probably don‘t want to do this inside a loop.

Note that inclusion of library modules is better done with the `use()` and `require()` operators, which also do error checking and raise an exception if there‘s a problem.

dump LABEL

This causes an immediate core dump. Primarily this is so that you can use the **undump** program 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. See also **−u** option in *perlrun*.

Example:

```
#!/usr/bin/perl
require 'getopt.pl';
require 'stat.pl';
%days = (
    'Sun' => 1,
    'Mon' => 2,
    'Tue' => 3,
    'Wed' => 4,
    'Thu' => 5,
    'Fri' => 6,
    'Sat' => 7,
);

dump QUICKSTART if $ARGV[0] eq '-d';

QUICKSTART:
Getopt('f');
```

each HASH

When called in a list context, returns a 2−element array consisting of the key and value for the next element of a hash, so that you can iterate over it. When called in a scalar context, returns the key for only the next element in the hash. (Note: Keys may be "0" or "", which are logically false; you may wish to avoid constructs like `while ($k = each %foo) {}` for this reason.)

Entries are returned in an apparently random order. When the hash is entirely read, a null array is returned in list context (which when assigned produces a FALSE (0) value), and `undef` is returned in a 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.

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()` and `values()`.

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 name. (Note that this function actually reads a character and then `ungetc()`s it, so it is not 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. Filetypes such as terminals may lose the end−of−file condition if you do.

An `eof` without an argument uses the last file read as argument. Empty parentheses `()` may be used to indicate the pseudo file formed of the files listed on the command line, i.e., `eof()` is reasonable to use inside a `while (<>)` loop to detect the end of only the last file. Use `eof(ARGV)` or eof without the parentheses to test *EACH* file in a while (<>) loop. Examples:

```
    # reset line numbering on each input file
    while (<>) {
        print "$.\t$_";
        close(ARGV) if (eof);   # Not eof().
    }

    # insert dashes just before last line of last file
    while (<>) {
        if (eof()) {
            print "--------------\n";
            close(ARGV);        # close or break; is needed if we
                                # are reading from the terminal
        }
        print;
    }
```

Practical hint: you almost never need to use `eof` in Perl, because the input operators return undef when they run out of data.

eval EXPR
eval BLOCK

EXPR is parsed and executed as if it were a little Perl program. It is executed in the context of the current Perl program, so that any variable settings or subroutine and format definitions remain afterwards. The value returned is the value of the last expression evaluated, or a return statement may be used, just as with subroutines. The last expression is evaluated in scalar or array context, depending on the context of the eval.

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. If EXPR is omitted, evaluates `$_.` The final semicolon, if any, may be omitted from the expression. Beware that using `eval()` neither silences perl from printing warnings to STDERR, nor does it stuff the text of warning messages into `$@.` To do either of those, you have to use the `$SIG{__WARN__}` facility. See `warn()` and *perlvar*.

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 non−fatal
    eval { $answer = $a / $b; }; warn $@ if $@;

    # same thing, but less efficient
    eval '$answer = $a / $b'; warn $@ if $@;

    # a compile−time error
    eval { $answer = };

    # a run−time error
    eval '$answer =';   # sets $@
```

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 foofs here" };
       print $@ if $@;                # prints "bar barfs here"
    }
```

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 that particular situation, you can just use symbolic references instead, as in case 6.

exec LIST

The `exec()` function executes a system command *AND NEVER RETURNS*, unless the command does not exist and is executed directly instead of via `/bin/sh −c` (see below). Use `system()` instead of `exec()` if you want it to return.

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, the argument is checked for shell metacharacters. If there are any, the entire argument is passed to `/bin/sh −c` for parsing. If there are none, the argument is split into words and passed directly to `execvp()`, which is more efficient. Note: `exec()` and `system()` do not flush your output buffer, so you may need to set `$|` to avoid lost output. 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 multi−valued 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
```

exists EXPR

Returns TRUE if the specified hash key exists in its hash array, even if the corresponding value is undefined.

```
    print "Exists\n" if exists $array{$key};
    print "Defined\n" if defined $array{$key};
    print "True\n" if $array{$key};
```

A hash element can be TRUE only if it‘s defined, and defined if it exists, but the reverse doesn‘t necessarily hold true.

Note that the EXPR can be arbitrarily complicated as long as the final operation is a hash key lookup:

```
    if (exists $ref−>[$x][$y]{$key}) { ... }
```

exit EXPR

Evaluates EXPR and exits immediately with that value. (Actually, it calls any defined `END` routines first, but the `END` routines may not abort the exit. Likewise any object destructors that need to be called are called before exit.) Example:

```
    $ans = <STDIN>;
    exit 0 if $ans =~ /^[Xx]/;
```

See also `die()`. If EXPR is omitted, exits with 0 status. The only univerally portable values for EXPR are 0 for success and 1 for error; all other values are subject to unpredictable interpretation depending on the environment in which the Perl program is running.

You shouldn‘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()`.

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 function definitions. Argument processing and value return works just like `ioctl()` below. Note that `fcntl()` will produce a fatal error if used on a machine that doesn‘t implement fcntl(2). For example:

```
    use Fcntl;
    fcntl($filehandle, F_GETLK, $packed_return_buffer);
```

fileno FILEHANDLE

Returns the file descriptor for a filehandle. This is useful for constructing bitmaps for `select()`. If FILEHANDLE is an expression, the value is taken as the name of the filehandle.

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.

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 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 added to 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 mis−coordination, Perl flushes FILEHANDLE before (un)locking 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 (all?) systems implement lockf(3) in terms of fcntl(2) locking, though, so the differing semantics shouldn‘t bite too many people.

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();
```

See also *DB_File* for other `flock()` examples.

fork

Does a fork(2) system call. Returns the child pid to the parent process and 0 to the child process, or `undef` if the fork is unsuccessful. Note: unflushed buffers remain unflushed in both processes, which means you may need to set `$|` (`$AUTOFLUSH` in English) or call the `autoflush()` method of IO::Handle to avoid duplicate output.

If you `fork()` without ever waiting on your children, you will accumulate zombies:

```
$SIG{CHLD} = sub { wait };
```

There‘s also the double−fork trick (error checking on `fork()` returns omitted);

```
unless ($pid = fork) {
    unless (fork) {
        exec "what you really wanna do";
        die "no exec";
        # ... or ...
        ## (some_perl_code_here)
        exit 0;
    }
    exit 0;
}
waitpid($pid,0);
```

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, the remote server (such as, say, httpd or rsh) won‘t think you‘re done. You should reopen those to /dev/null if it‘s any issue.

format

Declare a picture format with 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 `format`s, 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 a null string at end of file. If FILEHANDLE is omitted, reads from STDIN. This is not particularly efficient. It cannot be used to get unbuffered single−characters, however. 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 alleging POSIX compliance. See also the `Term::ReadKey` module from your nearest CPAN site; details on CPAN can be found on *CPAN*.

getlogin 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, $iaddr) = unpack_sockaddr_in($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.

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. Within a list context, the return values from the various get routines are as follows:

```
($name,$passwd,$uid,$gid,
   $quota,$comment,$gcos,$dir,$shell) = getpw*
($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.)

Within a 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 = getpwuid
    $name = getpwent
    $gid = getgrnam
    $name = getgrgid
    $name = getgrent
    etc.
```

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]);
```

getsockname SOCKET

Returns the packed sockaddr address of this end of the SOCKET connection.

```
    use Socket;
    $mysockaddr = getsockname(SOCK);
    ($port, $myaddr) = unpack_sockaddr_in($mysockaddr);
```

getsockopt SOCKET,LEVEL,OPTNAME

Returns the socket option requested, or undefined if there is an error.

glob EXPR

glob Returns the value of EXPR with filename expansions such as a shell would do. 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*.

gmtime EXPR

Converts a time as returned by the time function to a 9−element array with the time localized for the standard Greenwich time zone. Typically used as follows:

```
    ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
                                            gmtime(time);
```

All array elements are numeric, and come straight out of a struct tm. In particular this means that `$mon` has the range 0..11 and `$wday` has the range 0..6. Also, `$year` is the number of years since 1900, *not* simply the last two digits of the year.

If EXPR is omitted, does `gmtime(time())`.

In a scalar context, prints out the ctime(3) value:

```
    $now_string = gmtime;  # e.g., "Thu Oct 13 04:54:34 1994"
```

Also see the ***timegm.pl*** library, and the strftime(3) function available via the POSIX module.

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 goto−EXPR form expects a label name, whose scope will be resolved dynamically. This allows for computed gotos per FORTRAN, but isn‘t necessarily recommended if you‘re optimizing for maintainability:

```
    goto ("FOO", "BAR", "GLARCH")[$i];
```

The goto−`&NAME` form is highly magical, and substitutes a call to the named subroutine for the currently running subroutine. This is used by AUTOLOAD subroutines that wish to load another subroutine and then pretend that the other subroutine had been called in the first place (except that any modifications to @_ in the current subroutine are propagated to the other subroutine.) After the goto, not even `caller()` will be able to tell that this routine was called first.

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 a 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, because `$_` is a reference into the list value, it can be used to modify the elements of the array. While this is useful and supported, it can cause bizarre results if the LIST is not a named array. Similarly, grep returns aliases into the original list, much like the way that *Foreach Loops*‘s index variable aliases the list elements. That is, modifying an element of a list returned by grep actually modifies the element in the original list.

hex EXPR
hex Interprets EXPR as a hex string and returns the corresponding value. (To convert strings that might start with either 0 or 0x see *oct*.) If EXPR is omitted, uses `$_`.

```
    print hex '0xAf'; # prints '175'
    print hex 'aF';   # same
```

import There is no built−in `import()` function. It is merely 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

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 `$_`.

ioctl FILEHANDLE,FUNCTION,SCALAR

Implements the ioctl(2) function. You'll probably have to say

```
require "ioctl.ph"; # probably in /usr/local/lib/perl/ioctl.ph
```

first 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 which may help you in this, but it's non−trivial.) 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 are useful for manipulating the values of structures used by `ioctl()`. The following example sets the erase character to DEL.

```
require 'ioctl.ph';
$getp = &TIOCGETP;
die "NO TIOCGETP" if $@ || !$getp;
$sgttyb_t = "ccccs";                # 4 chars and a short
if (ioctl(STDIN,$getp,$sgttyb)) {
    @ary = unpack($sgttyb_t,$sgttyb);
    $ary[2] = 127;
    $sgttyb = pack($sgttyb_t,@ary);
    ioctl(STDIN,&TIOCSETP,$sgttyb)
        || die "Can't 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:

```
($retval = ioctl(...)) || ($retval = -1);
printf "System returned %d\n", $retval;
```

join EXPR,LIST

Joins the separate strings of LIST or ARRAY into a single string with fields separated by the value of EXPR, and returns the string. Example:

```
$_ = join(':', $login,$passwd,$uid,$gid,$gcos,$home,$shell);
```

See *split*.

keys HASH

Returns a normal array consisting of all the keys of the named hash. (In a scalar context, returns the number of keys.) The keys are returned in an apparently random order, but it is the same order as either the `values()` or `each()` function produces (given that the hash has not been modified). As a side effect, it resets HASH's iterator.

Here is yet another way to print your environment:

```
@keys = keys %ENV;
@values = values %ENV;
while ($#keys >= 0) {
    print pop(@keys), '=', pop(@values), "\n";
}
```

or how about sorted by key:

```
foreach $key (sort(keys %ENV)) {
    print $key, '=', $ENV{$key}, "\n";
}
```

To sort an array 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. 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).

kill LIST

Sends a signal to a list of processes. The first element of the list must be the signal to send. Returns the number of processes successfully signaled.

```
$cnt = kill 1, $child1, $child2;
kill 9, @goners;
```

Unlike in the shell, in Perl if the *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 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
    ...
}
```

lc EXPR
lc

Returns an 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*.

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*.

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 `$_`.

link OLDFILE,NEWFILE

Creates a new filename linked to the old filename. Returns 1 for success, 0 otherwise.

listen SOCKET,QUEUESIZE

Does the same thing that the listen system call does. Returns TRUE if it succeeded, FALSE otherwise. See example in *Sockets: Client/Server Communication in perlipc*.

local EXPR

A local modifies the listed variables to be local to the enclosing block, subroutine, `eval{}`, or `do`. If more than one value is listed, the list must be placed in parentheses. See *"Temporary Values via `local()`"* for details.

But 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()`"* for details.

localtime EXPR

Converts a time as returned by the time function to a 9−element array with the time analyzed for the local time zone. Typically used as follows:

```
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
                                            localtime(time);
```

All array elements are numeric, and come straight out of a struct tm. In particular this means that `$mon` has the range 0..11 and `$wday` has the range 0..6 and `$year` is year−1900, that is, `$year` is 123 in year 2023. If EXPR is omitted, uses the current time ("localtime(time)").

In a scalar context, returns the ctime(3) value:

```
$now_string = localtime;  # e.g., "Thu Oct 13 04:54:34 1994"
```

Also see the Time::Local module, and the strftime(3) function available via the POSIX module.

log EXPR
log
Returns logarithm (base *e*) of EXPR. If EXPR is omitted, returns log of `$_`.

lstat FILEHANDLE
lstat EXPR
lstat
Does the same thing as the `stat()` function, 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.

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. Evaluates BLOCK or EXPR in a 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) {
    $hash{getkey($_)} = $_;
}
```

mkdir FILENAME,MODE

Creates the directory specified by FILENAME, with permissions specified by MODE (as modified by umask). If it succeeds it returns 1, otherwise it returns 0 and sets `$!` (errno).

msgctl ID,CMD,ARG

Calls the System V IPC function msgctl(2). 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.

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.

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 long integer message type, which may be created with `pack("l", $type).` Returns TRUE if successful, or FALSE if there is an error.

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 if a message is received, the message type will be the first thing in VAR, and the maximum length of VAR is SIZE plus the size of the message type. Returns TRUE if successful, or FALSE if there is an error.

my EXPR

A "my" declares the listed variables to be local (lexically) to the enclosing block, subroutine, `eval`, or `do/require/use`'d file. If more than one value is listed, the list must be placed in parentheses. See *"Private Variables via `my()`"* for details.

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.

no Module LIST

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 instead.) The following will handle decimal, octal, and hex in the standard Perl or C notation:

```
$val = oct($val) if $val =~ /^0/;
```

If EXPR is omitted, uses `$_.` This 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

Opens the file whose filename is given by EXPR, and associates it with FILEHANDLE. If FILEHANDLE is an expression, its value is used as the name of the real filehandle wanted. 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 the filename begins with ‘<’ or nothing, the file is opened for input. If the filename begins with ’‘, the file is truncated and opened for output. If the filename begins with ’‘, the file is opened for appending. 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. The prefix and the filename may be separated with spaces. These various prefixes correspond to the fopen(3) modes of ‘r‘, ‘r+‘, ‘w‘, ‘w+‘, ‘a‘, and ‘a+’.

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 See *"Using `open()` for IPC"* for more examples of this. as command which pipes input to us. (You may not have a raw `open()` to a command that pipes both in *and* out, but see *IPC::Open2*, *IPC::Open3*, and *Bidirectional Communication in perlipc* for alternatives.)

Opening ‘−’ opens STDIN and opening ‘>−’ opens STDOUT. Open returns non−zero 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 unfortunate enough to be running Perl on a system that distinguishes between text files and binary files (modern operating systems don‘t care), 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 and Plan9 that delimit lines with a single character, and that encode that character in C as ‘\n‘, do not need `binmode`. The rest need it.

Examples:

```
$ARTICLE = 100;
open ARTICLE or die "Can’t find article $ARTICLE: $!\n";
while (<ARTICLE>) {...

open(LOG, ’>>/usr/spool/news/twitlog’); # (log is reserved)

open(DBASE, ’+<dbase.mine’);            # open for update

open(ARTICLE, "caesar <$article |");    # decrypt article

open(EXTRACT, "|sort >/tmp/Tmp$$");     # $$ is our process id

# process argument list of files along with any includes

foreach $file (@ARGV) {
    process($file, ’fh00’);
}

sub process {
    local($filename, $input) = @_;
    $input++;               # this is a string increment
    unless (open($input, $filename)) {
        print STDERR "Can’t open $filename: $!\n";
        return;
    }
```

```
            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) which is to be duped 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 stdio buffers.) Here is a script that saves, redirects, and restores STDOUT and STDERR:

```
    #!/usr/bin/perl
    open(SAVEOUT, ">&STDOUT");
    open(SAVEERR, ">&STDERR");

    open(STDOUT, ">foo.out") || die "Can't redirect stdout";
    open(STDERR, ">&STDOUT") || 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

    close(STDOUT);
    close(STDERR);

    open(STDOUT, ">&SAVEOUT");
    open(STDERR, ">&SAVEERR");

    print STDOUT "stdout 2\n";
    print STDERR "stderr 2\n";
```

If you specify "<&=N", where N is a number, then Perl will do an equivalent of C's `fdopen()` of that file descriptor; this is more parsimonious of file descriptors. For example:

```
    open(FILEHANDLE, "<&=$fd")
```

If you open a pipe on the command "−", i.e., either "|−" or "−|", 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 pairs are more or less equivalent:

```
    open(FOO, "|tr '[a−z]' '[A−Z]'");
    open(FOO, "|−") || exec 'tr', '[a−z]', '[A−Z]';

    open(FOO, "cat −n '$file'|");
    open(FOO, "−|") || exec 'cat', '−n', $file;
```

See *Safe Pipe Opens in perlipc* for more examples of this.

Explicitly closing any piped filehandle causes the parent process to wait for the child to finish, and returns the status value in `$?`. Note: on any operation which may do a fork, unflushed buffers remain unflushed in both processes, which means you may need to set `$|` to avoid duplicate output.

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 which 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.
    }
```

The filename that is passed to open will have leading and trailing whitespace deleted. To open a file with arbitrary weird characters in it, it‘s necessary to protect any leading and trailing whitespace thusly:

```
    $file =~ s#^(\s)#./$1#;
    open(FOO, "< $file\0");
```

If you want a "real" C `open()` (see *open(2)* on your system), then you should use the `sysopen()` function. This is another way to protect your filenames from interpretation. For example:

```
    use IO::Handle;
    sysopen(HANDLE, $path, O_RDWR|O_CREAT|O_EXCL, 0700)
        or die "sysopen $path: $!";
    HANDLE->autoflush(1);
    HANDLE->print("stuff $$\n");
    seek(HANDLE, 0, 0);
    print "File contains: ", <HANDLE>;
```

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. DIRHANDLEs have their own namespace separate from FILEHANDLEs.

ord EXPR

ord Returns the numeric ascii value of the first character of EXPR. If EXPR is omitted, uses `$_`. For the reverse, see *chr*.

pack TEMPLATE,LIST

Takes an array or list of values and packs it into a binary structure, returning the string containing the structure. The TEMPLATE is a sequence of characters that give the order and type of values, as follows:

```
    A   An ascii string, will be space padded.
    a   An ascii string, will be null padded.
    b   A bit string (ascending bit order, like vec()).
    B   A bit string (descending bit order).
    h   A hex string (low nybble first).
    H   A hex string (high nybble first).

    c   A signed char value.
    C   An unsigned char value.
    s   A signed short value.
    S   An unsigned short value.
    i   A signed integer value.
    I   An unsigned integer value.
    l   A signed long value.
    L   An unsigned long value.

    n   A short in "network" order.
    N   A long in "network" order.
    v   A short in "VAX" (little−endian) order.
    V   A long in "VAX" (little−endian) order.

    f   A single−precision float in the native format.
    d   A double−precision float in the native format.

    p   A pointer to a null−terminated string.
    P   A pointer to a structure (fixed−length string).

    u   A uuencoded string.

    w A BER compressed integer.  Bytes give an unsigned integer base
      128, most significant digit first, with as few digits as
      possible, and with the bit 8 of each byte except the last set
      to "1."

    x   A null byte.
    X   Back up a byte.
    @   Null fill to absolute position.
```

Each letter may optionally be followed by a number which gives a repeat count. With all types except "a", "A", "b", "B", "h", "H", 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. The "a" and "A" 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, but "a" does not.) Likewise, the "b" and "B" fields pack a string that many bits long. The "h" and "H" fields pack a string that many nybbles long. The "P" packs a pointer to a structure of the size indicated by the length. 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). 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).`

Examples:

```
    $foo = pack("cccc",65,66,67,68);
    # foo eq "ABCD"
    $foo = pack("c4",65,66,67,68);
    # same thing
```

```
$foo = pack("ccxxcc",65,66,67,68);
# foo eq "AB\0\0CD"

$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)

sub bintodec {
    unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
}
```

The same template may generally also be used in the unpack function.

package NAMESPACE

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 (the same scope as the `local()` 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 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 influences merely 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.`

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 stdio 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.

pop ARRAY

pop Pops and returns the last value of the array, shortening the array by 1. Has a similar effect to

```
$tmp = $ARRAY[$#ARRAY--];
```

If there are no elements in the array, returns the undefined value. 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 is 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 comma−separated 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 STDOUT. To set the default output channel to something other than STDOUT use the select operation. Note that, because print takes a LIST, anything in the LIST is evaluated in a list context, and any subroutine that you call will have one or more of its expressions evaluated in a 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)`. The first argument of the list will be interpreted as the printf format. 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.

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/
qx/STRING/
qw/STRING/

Generalized quotes. See *perlop*.

quotemeta EXPR
quotemeta

Returns the value of EXPR with all non−alphanumeric 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 between 0 and the value of EXPR. (EXPR should be positive.) If EXPR is omitted, returns a value between 0 and 1. Automatically calls `srand()` unless `srand()` has already been called. See also `srand()`.

(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 bytes of data into variable SCALAR from the specified FILEHANDLE. Returns the number of bytes actually read, or undef if there was an error. SCALAR will be grown or shrunk to the length actually read. An OFFSET may be specified to place the read data at some other place than the beginning of the string. This call is actually implemented in terms of stdio‘s fread call. To get a true read system call, see `sysread()`.

readdir DIRHANDLE

Returns the next directory entry for a directory opened by `opendir()`. If used in a list context, returns all the rest of the entries in the directory. If there are no more entries, returns an undefined value in a scalar context or a null list in a 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;
```

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 `$!` (errno). If EXPR is omitted, uses `$_`.

recv SOCKET,SCALAR,LEN,FLAGS

Receives a message on a socket. Attempts to receive LENGTH bytes of data into variable SCALAR from the specified SOCKET filehandle. Actually does a C `recvfrom()`, so that it can returns the address of the sender. Returns the undefined value if there‘s an error. SCALAR will be grown or shrunk to the length actually read. Takes the same flags as the system call of the same name. See *UDP: Message Passing in perlipc* for examples.

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;
        }
```

ref EXPR

ref Returns a TRUE value if EXPR is a reference, FALSE 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:

```
REF
SCALAR
ARRAY
HASH
CODE
GLOB
```

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($r) eq "HASH") {
    print "r is a reference to a hash.\n";
}
if (!ref ($r) {
    print "r is not a reference at all.\n";
}
```

See also *perlref*.

rename OLDNAME,NEWNAME

Changes the name of a file. Returns 1 for success, 0 otherwise. Will not work across file system boundaries.

require EXPR

require Demands some semantics specified by EXPR, or by `$_` if EXPR is not supplied. If EXPR is numeric, demands that the current version of Perl (`$]` or `$PERL_VERSION)` be equal or greater than EXPR.

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 {
    local($filename) = @_;
    return 1 if $INC{$filename};
    local($realfilename,$result);
    ITER: {
        foreach $prefix (@INC) {
            $realfilename = "$prefix/$filename";
            if (-f $realfilename) {
                $result = do $realfilename;
                last ITER;
            }
        }
        die "Can’t find $filename in \@INC";
    }
    die $@ if $@;
```

```
            die "$filename did not return true value" unless $result;
            $INC{$filename} = $realfilename;
            $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 bare word, 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.

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 ?? searches
```

Resetting "A−Z" is not recommended because you‘ll wipe out your ARGV and ENV arrays. 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 LIST

Returns from a subroutine, `eval()`, or do FILE with the value specified. (Note that in the absence of a return, a subroutine, eval, or do FILE will automatically return the value of the last expression evaluated.)

reverse LIST

In a list context, returns a list value consisting of the elements of LIST in the opposite order. In a scalar context, concatenates the elements of LIST, and returns a string value consisting of those bytes, but in the opposite order.

```
    print reverse <>;           # line tac, last line first

    undef $/;                   # for efficiency of <>
    print scalar reverse <>;    # byte tac, last line tsrif
```

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.

```
    %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 it is empty. If it succeeds it returns 1, otherwise it returns 0 and sets `$!` (errno). If FILENAME is omitted, uses `$_`.

s/// The substitution operator. See *perlop*.

scalar EXPR

Forces EXPR to be interpreted in a 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 a list context because it‘s in practice never needed. If you really wanted to do so, however, you could use the construction `@{[ (some expression) ]}`, but usually a simple `(some expression)` suffices.

seek FILEHANDLE,POSITION,WHENCE

Randomly positions the file pointer for FILEHANDLE, 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 file pointer to POSITION, 1 to set the it to current plus POSITION, and 2 to set it to EOF plus offset. You may use the values SEEK_SET, SEEK_CUR, and SEEK_END for this from POSIX module. Returns 1 upon success, 0 otherwise.

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 pointer:

```
    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. First the simple trick listed above to clear the filepointer. 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 stdios 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 {
    local(@fhlist) = split(' ',$_[0]);
    local($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 `$timeleft,` so calling `select()` in a scalar context just returns `$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);
```

WARNING: Do not attempt to mix buffered I/O (like `read()` or <FH>) with `select()`. You have to use `sysread()` instead.

semctl ID,SEMNUM,CMD,ARG

Calls the System V IPC function semctl. 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.

semget KEY,NSEMS,FLAGS

Calls the System V IPC function semget. Returns the semaphore id, or the undefined value if there is an error.

semop KEY,OPSTRING

Calls the System V IPC function semop to perform semaphore operations such as signaling and waiting. OPSTRING must be a packed array of semop structures. Each semop structure can be generated with `pack("sss", $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("sss", $semnum, -1, 0);
    die "Semaphore trouble: $!\n" unless semop($semid, $semop);
```

To signal the semaphore, replace "−1" with "1".

send SOCKET,MSG,FLAGS,TO
send SOCKET,MSG,FLAGS

Sends a message on a socket. 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. See *UDP: Message Passing in perlipc* for examples.

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 setpgrp(2). If the arguments are omitted, it defaults to 0,0. Note that the POSIX version of `setpgrp()` does not accept any arguments, so only setpgrp 0,0 is portable.

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 @ARGV array in the main program, and the @_ array in subroutines. (This is determined lexically.) 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. 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.

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.

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 which 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.

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.

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 `POSIX::sin()` 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 by sending the process a 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.

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()`* below.

See also the POSIX module‘s `sigpause()` 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 example in *Sockets: Client/Server Communication in perlipc*.

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.

sort SUBNAME LIST
sort BLOCK LIST
sort LIST

Sorts the LIST and returns the sorted list value. If SUBNAME or BLOCK is omitted, sorts 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 array are to be ordered. (The `<=>` and `cmp` operators are extremely useful in such routines.) SUBNAME may be a scalar variable name, in which case the value provides the name of the subroutine to use. In place of a SUBNAME, you can provide a BLOCK as an anonymous, in−line sort subroutine.

In the interests of efficiency the normal calling code for subroutines is bypassed, with the following effects: the subroutine may not be a recursive subroutine, and the two elements to be compared are passed into the subroutine not via @_ but as the package global variables `$a` and `$b` (see example below). They are passed by reference, so don‘t modify `$a` and `$b.` And don‘t try to declare them as lexicals either.

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*.

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;

    # sort using explicit subroutine name
    sub byage {
        $age{$a} <=> $age{$b};  # presuming numeric
    }
    @sortedclass = sort byage @class;

    # this sorts the %age hash by value instead of key
    # using an in-line function
    @eldest = sort { $age{$b} <=> $age{$a} } keys %age;

    sub backwards { $b cmp $a; }
    @harry = ('dog','cat','x','Cain','Abel');
    @george = ('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 using a Schwartzian Transform (no temps)
    @new = map { $_->[0] }
        sort { $b->[1] <=> $a->[1]
                       ||
               $a->[2] cmp $b->[2]
        } map { [$_, /=(\d+)/, uc($_)] } @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, it‘s

```
    @articles = sort {$main::b <=> $main::a} @files;
```

or just

```
    @articles = sort {$::b <=> $::a} @files;
```

but if you‘re in the `FooPack` package, it‘s

```
    @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 Perl interpreter will probably crash and dump core. This is entirely due to and dependent upon your system‘s qsort(3) library routine; this routine often avoids sanity checks in the interest of speed.

splice ARRAY,OFFSET,LENGTH,LIST
splice ARRAY,OFFSET,LENGTH
splice ARRAY,OFFSET

Removes the elements designated by OFFSET and LENGTH from an array, and replaces them with the elements of LIST, if any. Returns the elements removed from the array. The array grows or shrinks as necessary. If LENGTH is omitted, removes everything from OFFSET onward. The following equivalences hold (assuming `$[ == 0`):

```
    push(@a,$x,$y)      splice(@a,$#a+1,0,$x,$y)
    pop(@a)             splice(@a,-1)
    shift(@a)           splice(@a,0,1)
    unshift(@a,$x,$y)   splice(@a,0,0,$x,$y)
    $a[$x] = $y         splice(@a,$x,1,$y);
```

Example, assuming array lengths are passed before arrays:

```
    sub aeq {   # compare two list values
        local(@a) = splice(@_,0,shift);
        local(@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 a string into an array of strings, and returns it.

If not in a list context, returns the number of fields found and splits into the @_ array. (In a list context, you can force the split into @_ by using `??` as the pattern delimiters, but it still returns the array value.) The use of implicit split to @_ is deprecated, however.

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 is not negative, splits into no more than that many fields (though it may split into fewer). If LIMIT is unspecified, 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.

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'.

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, 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 array 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*/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.

Example:

```
    open(passwd, '/etc/passwd');
    while (<passwd>) {
        ($login, $passwd, $uid, $gid, $gcos,
            $home, $shell) = split(/:/);
        ...
    }
```

(Note that `$shell` above will still have a newline on it. See */chop*, */chomp*, and */join*.)

sprintf FORMAT, LIST

Returns a string formatted by the usual printf conventions of the C language. See *sprintf(3)* or *printf(3)* on your system for details. (The * character for an indirectly specified length is not supported, but you can get the same effect by interpolating a variable into the pattern.) 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*. Some C libraries' implementations of `sprintf()` can dump core when fed ludicrous arguments.

sqrt EXPR

sqrt

Return the square root of EXPR. If EXPR is omitted, returns square root of `$_.`

srand EXPR

srand

Sets the random number seed for the `rand` operator. If EXPR is omitted, uses a semi−random value based on the current time and process ID, among other things. 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.

In fact, it‘s usually not necessary to call `srand()` at all, because if it is not called explicitly, it is called implicitly at the first use of the `rand` operator. However, this was not the case in version of Perl before 5.004, so if your script will run under older Perl versions, it should call `srand()`.

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.

Do *not* call `srand()` multiple times in your program unless you know exactly what you‘re doing and why you‘re doing it. 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. Just do it once at the top of your program, or you *won‘t* get random numbers out of `rand()`!

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 array 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 meaning of the fields:

```
  dev       device number of filesystem
  ino       inode number
  mode      file mode  (type and permissions)
  nlink     number of (hard) links to the file
  uid       numeric user ID of file's owner
  gid       numeric group ID of file's owner
  rdev      the device identifier (special files only)
  size      total size of file, in bytes
  atime     last access time since the epoch
  mtime     last modify time since the epoch
  ctime     inode change time (NOT creation time!) since the epoch
  blksize   preferred block size for file system I/O
  blocks    actual number of blocks allocated
```

(The epoch was at 00:00 January 1, 1970 GMT.)

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 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.)

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 which 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 BLOCK
sub NAME
sub NAME BLOCK

This is subroutine definition, not a real function *per se*. With just a NAME (and possibly prototypes), 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.

substr EXPR,OFFSET,LEN
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, starts that far from the end of the string. If LEN is omitted, returns everything to the end of the string. If LEN 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 be an lvalue. If you assign something shorter than LEN, the string will shrink, and if you assign something longer than LEN, 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()`.

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("","");', $@ eq '');
```

syscall 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. 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.

```
    require 'syscall.ph';               # may need to run h2ph
    syscall(&SYS_write, fileno(STDOUT), "hi there\n", 9);
```

Note that Perl supports passing of up to only 14 arguments to your system call, which in practice should usually suffice.

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`. However, for historical reasons, some values are universal: zero means read−only, one means write−only, and two means read/write.

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 PERMS is omitted, the default value is 0666, which allows read and write for all. This default is reasonable: see `umask`.

The IO::File module provides a more object−oriented approach, if you‘re into that kind of thing.

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 stdio, so mixing this with other kinds of reads may cause confusion. Returns the number of bytes actually read, or undef if there was an error. 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 bytes 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.

system 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. The return value is the exit status of the program as returned by the `wait()` call. To get the actual exit value divide by 256. See also */exec*. This is *NOT* what you want to use to capture the output from a command, for that you should use merely back−ticks or qx//, as described in *‘STRING‘ in perlop*.

Because `system()` and back−ticks block SIGINT and SIGQUIT, killing the program they‘re running doesn‘t actually interrupt your program.

```
    @args = ("command", "arg1", "arg2");
    system(@args) == 0
         or die "system @args failed: $?"
```

Here‘s a more elaborate example of analysing the return value from `system()` on a UNIX system to check for all possibilities, including for signals and coredumps.

```
    $rc = 0xffff & system @args;
    printf "system(%s) returned %#04x: ", "@args", $rc;
    if ($rc == 0) {
        print "ran with normal exit\n";
    }
    elsif ($rc == 0xff00) {
        print "command failed: $!\n";
    }
    elsif ($rc > 0x80) {
        $rc >>= 8;
        print "ran with non−zero exit status $rc\n";
    }
    else {
        print "ran with ";
        if ($rc &   0x80) {
            $rc &= ~0x80;
            print "coredump from ";
        }
        print "signal $rc\n"
    }
    $ok = ($rc != 0);
```

syswrite FILEHANDLE,SCALAR,LENGTH,OFFSET
syswrite FILEHANDLE,SCALAR,LENGTH

Attempts to write LENGTH bytes of data from variable SCALAR to the specified FILEHANDLE, using the system call write(2). It bypasses stdio, so mixing this with prints may cause confusion. Returns the number of bytes actually written, or undef if there was an error. If the length is greater than the available data, 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 from that many bytes counting backwards from the end of the string.

tell FILEHANDLE
tell

Returns the current file position for FILEHANDLE. FILEHANDLE may be an expression whose value gives the name of the actual filehandle. If FILEHANDLE is omitted, assumes the file last read.

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, 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 array values 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
```

```
DESTROY this
FETCH this, key
STORE this, key, value
DELETE this, key
EXISTS this, key
FIRSTKEY this
NEXTKEY this, lastkey
```

A class implementing an ordinary array should have the following methods:

```
TIEARRAY classname, LIST
DESTROY this
FETCH this, key
STORE this, key, value
[others TBD]
```

A class implementing a scalar should have the following methods:

```
TIESCALAR classname, LIST
DESTROY this
FETCH this,
STORE this, value
```

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.

tied VARIABLE

Returns a reference to the object underlying VARIABLE (the same value that was originally returned by the `tie()` call which 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 (that‘s 00:00:00, January 1, 1904 for MacOS, and 00:00:00 UTC, January 1, 1970 for most other systems). Suitable for feeding to `gmtime()` and `localtime()`.

times

Returns a four−element array giving the user and system times, in seconds, for this process and the children of this process.

```
($user,$system,$cuser,$csystem) = times;
```

tr///

The translation operator. 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.

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*.

If EXPR is omitted, uses `$_`.

ucfirst EXPR
ucfirst

Returns the value of EXPR with the first character uppercased. 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*.

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. 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 on only a scalar value, an entire array or hash, or a subroutine name (using "&"). (Using `undef()` will probably not do what you expect on most predefined variables or DBM list values, so don‘t do that.) 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;
return (wantarray ? () : undef) if $they_blew_it;
select undef, undef, undef, 0.25;
($a, $b, undef, $c) = &foo;       # Ignore third value returned
```

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 representing a structure and expands it out into a list value, returning the array value. (In a scalar context, it returns merely the first value produced.) The TEMPLATE has the same format as in the pack function. Here‘s a subroutine that does substring:

```
sub substr {
    local($what,$where,$howmuch) = @_;
    unpack("x$where a$howmuch", $what);
}
```

and then there‘s

```
sub ordinal { unpack("c",$_[0]); } # same as ord()
```

In addition, 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. For example, the following computes the same number as the System V sum program:

```
while (<>) {
    $checksum += unpack("%16C*", $_);
}
$checksum %= 65536;
```

The following efficiently counts the number of set bits in a bit vector:

```
    $setbits = unpack("%32b*", $selectmask);
```

untie VARIABLE

Breaks the binding between a variable and a package. (See `tie()`.)

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 LIST
use Module
use Module VERSION LIST
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 bare word.

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. 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.)

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 error is currently silently ignored. This may change to a fatal error in a future version.

If you don‘t want your namespace altered, explicitly supply an 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.` (Note that there is not a comma after VERSION!)

Because this is a wide−open interface, pragmas (compiler directives) are also implemented this way. Currently implemented pragmas are:

```
    use integer;
    use diagnostics;
    use sigtrap qw(SEGV BUS);
    use strict  qw(subs vars refs);
    use subs    qw(afunc blurfl);
```

These pseudo−modules import semantics into the current block scope, 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’;
```

If no unimport method can be found the call fails with a fatal error.

See *perlmod* for a list of standard modules and pragmas.

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 modification time of each file is set to the current time. Example of a "touch" command:

```
    #!/usr/bin/perl
    $now = time;
    utime $now, $now, @ARGV;
```

values HASH

Returns a normal array 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, but it is the same order as either the `keys()` or `each()` function would produce on the same hash. As a side effect, it resets HASH‘s iterator. See also `keys()`, `each()`, and `sort()`.

vec EXPR,OFFSET,BITS

Treats the string in EXPR as a vector of unsigned integers, and returns the value of the bit field specified by OFFSET. BITS specifies the number of bits that are reserved for each entry in the bit vector. This must be a power of two from 1 to 32. `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;
```

Vectors created with `vec()` can also be manipulated with the logical operators |, `&,` and ^, which will assume a bit vector operation is desired when both operands are strings.

To transform a bit vector into a string or array 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 *.

wait

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 `$?.`

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. The status is returned in `$?.` If you say

```
    use POSIX ":sys_wait_h";
    ...
    waitpid(−1,&WNOHANG);
```

then you can do a non−blocking wait for any process. 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.)

wantarray

Returns TRUE if the context of the currently executing subroutine is looking for a list value. Returns FALSE if the context is looking for a scalar.

```
return wantarray ? () : undef;
```

warn LIST

Produces a message on STDERR just like `die()`, but doesn‘t exit or throw an exception.

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.

write FILEHANDLE
write EXPR
write

Writes a formatted record (possibly multi−line) to the specified file, using the format associated with that file. By default the format for a file is the one having the same name is 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 translation operator. See *perlop*.

NAME

perlvar − Perl predefined variables

DESCRIPTION

Predefined Names

The following names have special meaning to Perl. Most of the punctuation names have reasonable mnemonics, or analogues in one of the shells. Nevertheless, if you wish to use the long variable names, you just need to say

```
use English;
```

at the top of your program. This will alias all the short names to the long names in the current package. Some of them even have medium names, generally borrowed from **awk**.

To go a step further, those variables that depend on the currently selected filehandle may instead be set by calling an object method on the FileHandle object. (Summary lines below for this contain the word HANDLE.) First you must say

```
use FileHandle;
```

after which you may use either

```
method HANDLE EXPR
```

or

```
HANDLE−>method(EXPR)
```

Each of the methods returns the old value of the FileHandle attribute. The methods each take an optional EXPR, which if supplied specifies the new value for the FileHandle attribute in question. If not supplied, most of the methods do nothing to the current value, except for `autoflush()`, which will assume a 1 for you, just to be different.

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.

`$ARG`
`$_` The default input and pattern−searching space. The following pairs are equivalent:

```
while (<>) {...}    # equivalent in only while!
while ($_ = <>) {...}

/^Subject:/
$_ =~ /^Subject:/

tr/a−z/A−Z/
$_ =~ tr/a−z/A−Z/

chop
chop($_)
```

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. Note that outside of a `while` test, this will not happen.

(Mnemonic: underline is understood in certain operations.)

`$<digit>`
: Contains the sub−pattern from the corresponding set of parentheses in the last pattern matched, not counting patterns matched in nested blocks that have been exited already. (Mnemonic: like \digit.) These variables are all read−only.

`$MATCH`
`$&`
: 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.

`$PREMATCH`
`$‘`
: 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.

`$POSTMATCH`
`$’`
: 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:

```
    $_ = 'abcdefghi';
    /def/;
    print "$`:$&:$'\n";         # prints abc:def:ghi
```

This variable is read−only.

`$LAST_PAREN_MATCH`
`$+`
: The last bracket matched by the last search pattern. This is useful if you don‘t know which of a set of alternative patterns matched. For example:

```
    /Version: (.*)|Revision: (.*)/ && ($rev = $+);
```

(Mnemonic: be positive and forward looking.) This variable is read−only.

`$MULTILINE_MATCHING`
`$*`
: Set to 1 to do multi−line matching within a string, 0 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 "`$*`" is 0. Default is 0. (Mnemonic: * matches multiple things.) Note that this variable influences the interpretation of only "`^`" and "`$`". A literal newline can be searched for even when `$* == 0`.

Use of "`$*`" is deprecated in modern perls.

input_line_number HANDLE EXPR
`$INPUT_LINE_NUMBER`
`$NR`
`$.`
: The current input line number for the last file handle from which you read (or performed a `seek` or `tell` on). An explicit close on a filehandle resets the line number. Because "`<>`" never does an explicit close, line numbers increase across ARGV files (but see examples under `eof()`). Localizing `$.` has the effect of also localizing Perl‘s notion of "the last read filehandle". (Mnemonic: many programs use "." to mean the current line number.)

input_record_separator HANDLE EXPR
`$INPUT_RECORD_SEPARATOR`
`$RS`
`$/` The input record separator, newline by default. Works like **awk**‘s RS variable, including treating empty lines as delimiters if set to the null string. (Note: An empty line cannot contain any spaces or tabs.) You may set it to a multicharacter string to match a multi−character delimiter. Note that setting it to `"\n\n"` means something slightly different than setting it to `""`, if the file contains consecutive empty lines. Setting it to `""` will treat two or more consecutive empty lines as a single empty line. Setting it to `"\n\n"` will blindly assume that the next input character belongs to the next paragraph, even if it‘s a newline. (Mnemonic: / is used to delimit line boundaries when quoting poetry.)

```
undef $/;
$_ = <FH>;          # whole file now here
s/\n[ \t]+/ /g;
```

Remember: the value of `$/` is a string, not a regexp. AWK has to be better for something :−)

autoflush HANDLE EXPR
`$OUTPUT_AUTOFLUSH`
`$|` If set to nonzero, forces a flush after every write or print on the currently selected output channel. Default is 0 (regardless of whether the channel is actually buffered by the system or not; `$|` tells you only whether you‘ve asked Perl explicitly to flush after each write). Note that 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, such as when you are running a Perl script under rsh and want to see the output as it‘s happening. This has no effect on input buffering. (Mnemonic: when you want your pipes to be piping hot.)

output_field_separator HANDLE EXPR
`$OUTPUT_FIELD_SEPARATOR`
`$OFS`
`$,` The output field separator for the print operator. Ordinarily the print operator simply prints out the comma−separated fields you specify. 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.)

output_record_separator HANDLE EXPR
`$OUTPUT_RECORD_SEPARATOR`
`$ORS`
`$\` The output record separator for the print operator. Ordinarily the print operator simply prints out the comma−separated fields you specify, with no trailing newline or record separator assumed. To get behavior more like **awk**, set this variable as you would set **awk**‘s ORS variable to specify what is printed at the end of the print. (Mnemonic: you set "`$\`" 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 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 multi−dimensional 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**. Note that 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" multi−dimensional arrays.

`$OFMT`
`$#` 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 is in fact 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.

format_page_number HANDLE EXPR
`$FORMAT_PAGE_NUMBER`
`$%` The current page number of the currently selected output channel. (Mnemonic: % is page number in **nroff**.)

format_lines_per_page HANDLE EXPR
`$FORMAT_LINES_PER_PAGE`
`$=` The current page length (printable lines) of the currently selected output channel. Default is 60. (Mnemonic: = has horizontal lines.)

format_lines_left HANDLE EXPR
`$FORMAT_LINES_LEFT`
`$−` The number of lines left on the page of the currently selected output channel. (Mnemonic: lines_on_page − lines_printed.)

format_name HANDLE EXPR
`$FORMAT_NAME`
`$~` The name of the current report format for the currently selected output channel. Default is name of the filehandle. (Mnemonic: brother to "`$^`".)

format_top_name HANDLE EXPR
`$FORMAT_TOP_NAME`
`$^` The name of the current top−of−page format for the currently selected output channel. Default is name of the filehandle with _TOP appended. (Mnemonic: points to top of page.)

format_line_break_characters HANDLE EXPR
`$FORMAT_LINE_BREAK_CHARACTERS`
`$:` The current set of characters after which a string may be broken to fill continuation fields (starting with ^) in a format. Default is " \n−", to break on whitespace or hyphens. (Mnemonic: a "colon" in poetry is a part of a line.)

format_formfeed HANDLE EXPR
`$FORMAT_FORMFEED`
`$^L` What formats output to perform a form feed. Default is \f.

`$ACCUMULATOR`

`$^A` The current value of the `write()` accumulator for `format()` lines. A format contains `formline()` commands that put their result into `$^A.` After calling its format, `write()` prints out the contents of `$^A` and empties. So you never actually see the contents of `$^A` unless you call `formline()` yourself and then look at it. See *perlform* and *`formline()`*.

`$CHILD_ERROR`
`$?` The status returned by the last pipe close, back−tick (`‘‘`) command, or `system()` operator. Note that this is the 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 actually (`$? >> 8`), and `$? & 255` gives which signal, if any, the process died from, and whether there was a core dump. (Mnemonic: similar to **sh** and **ksh**.)

Note that 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 the script.

Under VMS, the pragma `use vmsish ‘status’` makes `$?` reflect the actual VMS exit status, instead of the default emulation of POSIX status.

`$OS_ERROR`
`$ERRNO`
`$!` If used in a numeric context, yields the current value of errno, with all the usual caveats. (This means that you shouldn‘t depend on the value of "`$!`" to be anything in particular unless you‘ve gotten a specific error return indicating a system error.) If used in a string context, yields the corresponding system error string. You can assign 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?)

`$EXTENDED_OS_ERROR`
`$^E` More specific information about the last system error than that provided by `$!,` if available. (If not, it‘s just `$!` again, except under OS/2.) At the moment, this differs from `$!` under only VMS and OS/2, where it provides the VMS status value from the last system error, and OS/2 error code of the last call to OS/2 API which was not directed via CRT. The caveats mentioned in the description of `$!` apply here, too. (Mnemonic: Extra error explanation.)

Note that under OS/2 `$!` and `$^E` do not track each other, so if an OS/2−specific call is performed, you may need to check both.

`$EVAL_ERROR`
`$@` The Perl syntax error message from the last `eval()` command. If null, 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"?)

Note that warning messages are not collected in this variable. You can, however, set up a routine to process warnings by setting `$SIG{__WARN__}` below.

`$PROCESS_ID`
`$PID`
`$$` The process number of the Perl running this script. (Mnemonic: same as shells.)

`$REAL_USER_ID`
`$UID`
`$<` The real uid of this process. (Mnemonic: it‘s the uid you came *FROM*, if you‘re running setuid.)

`$EFFECTIVE_USER_ID`
`$EUID`

$ The effective uid of this process. Example:

```
$< = $>;            # set real to effective uid
($<,$>) = ($>,$<);  # swap real and effective uid
```

(Mnemonic: it's the uid you went *TO*, if you're running setuid.) Note: "`$<`" and "`$>`" can be swapped on only 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. (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. (Mnemonic: parentheses are used to *GROUP* things. The effective gid is the group that's *RIGHT* for you, if you're running setgid.)

Note: "`$<`", "`$>`", "`$(`" and "`$)`" can be set only on machines that support the corresponding *`set[re][ug]id()`* routine. "`$(`" and "`$)`" can be swapped on only machines supporting `setregid()`. Because Perl doesn't currently use `initgroups()`, you can't set your group vector to multiple groups.

`$PROGRAM_NAME`
`$0` Contains the name of the file containing the Perl script being executed. Assigning to "`$0`" modifies the argument area that the ps(1) program sees. This 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**.)

`$[` The index of the first element in an array, and of the first character in a substring. Default is 0, but you could 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 Perl 5, assignment to "`$[`" is treated as a compiler directive, and cannot influence the behavior of any other file. Its use is discouraged.

`$PERL_VERSION`
`$]` The string printed out when you say `perl -v`. (This is currently *BROKEN*). It can be used to determine at the beginning of a script whether the perl interpreter executing the script is in the right range of versions. If used in a numeric context, returns the version + patchlevel / 1000. Example:

```
# see if getc is available
($version,$patchlevel) =
         $] =~ /(\d+\.\d+).*\nPatch level: (\d+)/;
print STDERR "(No filename completion available.)\n"
         if $version * 1000 + $patchlevel < 2016;
```

or, used numerically,

```
warn "No checksumming!\n" if $] < 3.019;
```

(Mnemonic: Is this version of perl in the right bracket?)

$DEBUGGING

$^D The current value of the debugging flags. (Mnemonic: value of **−D** switch.)

$SYSTEM_FD_MAX

$^F The maximum system file descriptor, ordinarily 2. System file descriptors are passed to `exec()`ed processes, while higher file descriptors are not. Also, during an `open()`, system file descriptors are preserved even if the `open()` fails. (Ordinary file descriptors are closed before the `open()` is attempted.) Note that the close−on−exec status of a file descriptor will be decided according to the value of `$^F` at the time of the open, not the time of the exec.

$^H The current set of syntax checks enabled by `use strict`. See the documentation of `strict` for more details.

$INPLACE_EDIT

$^I The current value of the inplace−edit extension. Use `undef` to disable inplace editing. (Mnemonic: value of **−i** switch.)

$OSNAME

$^O The name of the operating system under which this copy of Perl was built, as determined during the configuration process. The value is identical to `$Config{‘osname‘}.`

$PERLDB

$^P The internal flag that the debugger clears so that it doesn‘t debug itself. You could conceivably disable debugging yourself by clearing it.

$BASETIME

$^T The time at which the script 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.

$WARNING

$^W The current value of the warning switch, either TRUE or FALSE. (Mnemonic: related to the **−w** switch.)

$EXECUTABLE_NAME

$^X The name that the Perl binary itself was executed as, from C‘s `argv[0]`.

$ARGV contains the name of the current file when reading from <>.

@ARGV The array @ARGV contains the command line arguments intended for the script. Note that $#ARGV is the generally number of arguments minus one, because `$ARGV[0]` is the first argument, *NOT* the command name. See "`$0`" for the command name.

@INC The array @INC contains the list of places to look for Perl scripts to be evaluated by the `do EXPR`, `require`, or `use` constructs. 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. 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;
```

%INC The hash %INC contains entries for each filename that has been included via `do` or `require`. The key is the filename you specified, and the value is the location of the file actually found. The `require` command uses this array to determine whether a given file has already been included.

$ENV{expr}

The hash %ENV contains your current environment. Setting a value in `ENV` changes the environment for child processes.

$SIG{expr}

The hash %SIG is used to set signal handlers for various signals. Example:

```
sub handler {       # 1st argument is signal name
    local($sig) = @_;
    print "Caught a SIG$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
```

The %SIG array contains values for only the signals actually set within the Perl script. Here are some other examples:

```
$SIG{PIPE} = Plumber;       # SCARY!!
$SIG{"PIPE"} = "Plumber";   # just fine, assumes main::Plumber
$SIG{"PIPE"} = \&Plumber;   # just fine; assume current Plumber
$SIG{"PIPE"} = Plumber();   # oops, what did Plumber() return??
```

The one marked scary is problematic because it's a bareword, which means sometimes it's a string representing the function, and sometimes it's going to call the subroutine call right then and there! Best to be sure and quote it or take a reference to it. *Plumber works too. See *perlsub*.

If your system has the `sigaction()` function then signal handlers are installed using it. This means you get reliable signal handling. If your system has the SA_RESTART flag it is used when signals handlers are installed. This means that system calls for which it is supported continue rather than returning when a signal arrives. If you want your system calls to be interrupted by signal delivery then do something like this:

```
use POSIX ':signal_h';

my $alarm = 0;
sigaction SIGALRM, new POSIX::SigAction sub { $alarm = 1 }
    or die "Error setting SIGALRM handler: $!\n";
```

See *POSIX*.

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__`. See *die*, *warn* and *eval*.

`$^M` By default, running out of memory it is not trappable. However, if compiled for this, Perl may use the contents of `$^M` as an emergency pool after `die()`ing with this message. Suppose that your Perl were compiled with −DEMERGENCY_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 *English* long name for this variable.

NAME

perlsub − Perl subroutines

SYNOPSIS

To declare subroutines:

```
sub NAME;             # A "forward" declaration.
sub NAME(PROTO);      #  ditto, but with prototypes

sub NAME BLOCK        # A declaration and a definition.
sub NAME(PROTO) BLOCK #  ditto, but with prototypes
```

To define an anonymous subroutine at runtime:

```
$subref = sub BLOCK;
```

To import subroutines:

```
use PACKAGE qw(NAME1 NAME2 NAME3);
```

To call subroutines:

```
NAME(LIST);    # & is optional with parentheses.
NAME LIST;     # Parentheses optional if pre−declared/imported.
&NAME;         # Passes current @_ to subroutine.
```

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 even generated on the fly using `eval` or anonymous subroutines (closures). You can even call a function indirectly using a variable containing its name or a CODE reference to it, as in `$var = \&function.`

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 the language‘s perspective.)

Any arguments passed to the routine come in as the array @_. Thus if you called a function with two arguments, those would be stored in `$_[0]` and `$_[1].` The array @_ is a local array, but its values are implicit references (predating *perlref*) to the actual scalar parameters. The return value of the subroutine is the value of the last expression evaluated. Alternatively, a return statement may be used to specify the returned value and exit the subroutine. If you return one or more arrays and/or hashes, these will be flattened together into one large indistinguishable list.

Perl does not have named formal parameters, but in practice all you do is assign to a `my()` list of these. Any variables you use in the function that aren‘t declared private are global variables. For the gory details on creating private variables, see *"Private Variables via `my()`"* and *"Temporary Values via `local()`"*. 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 = $lookahead;  # GLOBAL VARIABLES!!
        LINE: while ($lookahead = <STDIN>) {
            if ($lookahead =~ /^[ \t]/) {
                $thisline .= $lookahead;
            }
            else {
                last LINE;
            }
        }
        $thisline;
    }

    $lookahead = <STDIN>;       # get first line
    while ($_ = get_line()) {
        ...
    }
```

Use array assignment to a local list to name your formal arguments:

```
    sub maybeset {
        my($key, $value) = @_;
        $Foo{$key} = $value unless $Foo{$key};
    }
```

This also has the effect of turning call−by−reference into call−by−value, because the assignment copies the values. 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
    sub upcase {
        my @parms = @_;
        for (@parms) { tr/a−z/A−Z/ }
        # wantarray checks if we were called in list context
        return wantarray ? @parms : $parms[0];
    }
```

Notice how this (unprototyped) function doesn‘t care whether it was passed real scalars or arrays. Perl will see everything as one big long flat @_ parameter list. This is one of the ways 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);
```

Because like its flat incoming parameter list, the return list is also flat. So all you have managed to do here is stored everything in @a and made @b an empty list. See for alternatives.

A subroutine may be called using the "&" prefix. The "&" is optional in modern Perls, and so are the parentheses if the subroutine has been pre−declared. (Note, however, that the "&" is *NOT* optional when you're 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. See *perlref* for more on 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 pre-declared, else "foo"
```

Not only does the "&" form make the argument list optional, but it also disables any prototype checking on the 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 the section on Prototypes below.

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
```

A "my" declares the listed variables to be confined (lexically) 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 builtins like `$/` must currently be localized with "local" instead.

Unlike dynamic variables created by the "local" statement, lexical variables declared with "my" are totally hidden from the outside world, including any called subroutines (even if it's the same subroutine called from itself or elsewhere—every call gets its own copy).

(An `eval()`, however, can see the 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 the 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 the variables in its argument list, the "my" doesn‘t change whether those variables is viewed as a scalar or an array. So

```
    my ($foo) = <STDIN>;
    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;
```

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 the 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 the 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 throughout the rest of the conditional (including `elsif` and `else` clauses, if any), but not beyond it.

(None of the foregoing applies to `if/unless` or `while/until` modifiers appended to simple statements. Such modifiers are not control structures and have no effect on scoping.)

The `foreach` loop defaults to scoping its index variable dynamically (in the manner of `local`; see below). However, if the index variable is prefixed with the keyword "my", then it is lexically scoped instead. Thus in the loop

```
    for my $i (1, 2, 3) {
        some_function();
    }
```

the scope of `$i` extends to the end of the loop, but not beyond it, and so the value of `$i` is unavailable in `some_function()`.

Some users may wish to encourage the use of lexically scoped variables. As an aid to catching implicit references to package variables, if you say

```
    use strict 'vars';
```

then any variable reference from there to the end of the enclosing block must either refer to a lexical variable, or 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 principle usefulness of this is to quiet `use strict ‘vars’`. The actual initialization doesn‘t happen until run time, so gets executed every time through a loop.

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 totally from the outside world. This is similar to C‘s static variables at the file level. To do this with a subroutine requires the use of a closure (anonymous function). If a block (such as an `eval()`, function, or `package`) wants 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.

Just because the lexical variable is lexically (also called statically) scoped doesn‘t mean that within a function it works like a C static. It normally works more like a C auto. But 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 pain program, or more likely, placing merely a BEGIN sub around it to make sure it gets executed before your program starts to run:

```
sub BEGIN {
    my $secret_val = 0;
    sub gimme_another {
        return ++$secret_val;
    }
}
```

See *perlrun* about the BEGIN function.

Temporary Values via `local()`

NOTE: In general, you should be using "my" instead of "local", because it's faster and safer. Exceptions to this include the global punctuation variables, filehandles and formats, and direct manipulation of the Perl symbol table itself. Format variables often use "local" though, as do other variables whose current value must be visible to called subroutines.

Synopsis:

```
local $foo;                 # declare $foo dynamically local
local (@wid, %get);         # declare list of variables local
local $foo = "flurp";       # declare $foo dynamic, and init it
local @oof = @bar;          # declare @oof dynamic, and init it

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, (or subroutine, `eval{}`, or `do`) and *any called from within that block*. A `local()` just gives temporary values to global (meaning package) variables. This is known as dynamic scoping. Lexical scoping is done with "my", which works more like C's auto declarations.

If more than one variable is given to `local()`, they must be placed in parentheses. All listed elements must be legal lvalues. 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.) Commonly this is used to name the parameters to a subroutine. Examples:

```
for $i ( 0 .. 9 ) {
    $digits{$i} = $i;
}
# assume this function uses global %digits hash
```

```
    parse_num();

    # now temporarily add to %digits hash
    if ($base12) {
        # (NOTE: not claiming this is efficient!)
        local %digits  = (%digits, 't' => 10, 'e' => 11);
        parse_num();  # parse_num gets this new %digits!
    }
    # old %digits restored here
```

Because `local()` is a run−time command, it gets executed every time through a loop. In releases of Perl previous to 5.0, this used more stack storage each time until the loop was exited. Perl now reclaims the space each time through, but it‘s still more efficient to declare your variables outside the loop.

A local is simply a modifier on an lvalue expression. When you assign to a localized 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.

Passing Symbol Table Entries (typeglobs)

[Note: 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);
```

Note that 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*.

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, return 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 (well, and `local()`s) are in the symbol table.

If you‘re passing around filehandles, you could usually just use the bare typeglob, like *STDOUT, but typeglobs references would be better because they‘ll still work properly under `use strict 'refs'`. 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>;
    }
```

Another way to do this is using *HANDLE{IO}, see *perlref* for usage and caveats.

If you‘re planning on generating new filehandles, you could do this:

```
    sub openit {
        my $name = shift;
        local *FH;
        return open (FH, $path) ? *FH : undef;
    }
```

Although that will actually produce a small memory leak. See the bottom of *`open()`* for a somewhat cleaner way using the IO::Handle package.

Prototypes

As of the 5.002 release of perl, if you declare

```
    sub mypush (\@@)
```

then `mypush()` takes arguments exactly like `push()` does. The declaration of the function to be called must be visible at compile time. The prototype affects only the 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 builtin function, then it behaves like a builtin 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}`.

Method calls are not influenced by prototypes either, because the function to be called is indeterminate at compile time, because it depends on inheritance.

Because the intent is primarily to let you define subroutines that work like builtin commands, here are the prototypes for some other functions that parse almost exactly like the corresponding builtins.

```
    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 to the subroutine (as part of `@_`) will be a reference to the actual argument given in the subroutine call, obtained by applying `\` to that argument.

Unbackslashed prototype characters have special meanings. Any unbackslashed @ or % eats all the rest of the 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 * does whatever it has to do to turn the argument into a reference to a symbol table entry.

A semicolon separates mandatory arguments from optional arguments. (It is redundant before @ or %.)

Note how the last three examples 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 the prototype.

The interesting thing about & is that you can generate new syntax with it:

```
    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 the 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 grep:

```
    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.

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, the `func()` now gets passed in 1, that is, the number of elements in @foo. And the `split()` gets called in a scalar context and starts scribbling on your @_ parameter list.

This is all very powerful, of course, and should be used only in moderation to make the world a better place.

Constant Functions

Functions with a prototype of `()` are potential candidates for inlining. If the result after optimization and constant folding is a constant then it will be used in place of new−style calls to the function. Old−style calls (that is, calls made using `&`) are not affected.

All of the following functions would 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 ()      { 1 }
    sub BAZ_VAL () {
        if (OPT_BAZ) {
            return 23;
        }
        else {
            return 42;
        }
    }
```

If you redefine a subroutine which 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 the calling semantics, so beware) or by thwarting the inlining mechanism in some other way, such as

```
    my $dummy;
    sub not_inlined () {
        $dummy || 23
    }
```

Overriding Builtin Functions

Many builtin 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 builtin functionality on a non−Unix system.

Overriding may be done only by importing the name from a module—ordinary predeclaration isn‘t good enough. However, the `subs` pragma (compiler directive) lets you, in effect, pre−declare subs via the import syntax, and these names may then override the builtin ones:

```
    use subs 'chdir', 'chroot', 'chmod', 'chown';
    chdir $somewhere;
    sub chdir { ... }
```

Library modules should not in general export builtin 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 the name to the @EXPORT_OK list, 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 the overrides.

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 of the base classes of the class package.) If, however, there is an `AUTOLOAD` subroutine defined in the package or packages that were searched for 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 `$AUTOLOAD` variable in the same package as the `AUTOLOAD` routine. The name is not passed as an ordinary argument because, er, well, just because, that‘s why...

Most `AUTOLOAD` routines will load in a definition for the subroutine in question using eval, and then execute that subroutine using a special form of "goto" that erases the stack frame of the `AUTOLOAD` routine without a trace. (See the standard `AutoLoader` module, 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 call `system()` with those arguments. All you‘d do is this:

```
sub AUTOLOAD {
    my $program = $AUTOLOAD;
    $program =~ s/.*:://;
    system($program, @_);
}
date();
who('am', 'i');
ls('-l');
```

In fact, if you pre−declare the functions you want to call that way, you don‘t even need the 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 Unix programs.

Mechanisms are available for modules writers to help split the modules up 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*.

SEE ALSO

See *perlref* for more on references. See *perlxs* if you‘d like to learn about calling C subroutines from perl. See *perlmod* to learn about bundling up your functions in separate files.

NAME

perlmod − Perl modules (packages)

DESCRIPTION

Packages

Perl provides a mechanism for alternative namespaces to protect packages from stomping on each other‘s variables. In fact, apart from certain magical variables, 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 (the same scope as the `local()` 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 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 influences merely 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.)

Packages may be nested inside other packages: `$OUTER::INNER::var.` This implies nothing about the order of name lookups, however. All symbols are either local to the current package, or must be fully qualified from the outer package name down. For instance, there is nowhere within package `OUTER` that `$INNER::var` refers to `$OUTER::INNER::var.` It would treat package `INNER` as 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 of the punctuation variables like `$_.` In addition, 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 one. Note also that, if you have a package called `m`, `s`, or `y`, then you can‘t use the qualified form of an identifier because it will be interpreted instead as a pattern match, a substitution, or a translation.

(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. `$_` is still global though.)

`Eval()`ed 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 script 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*.

See *perlsub* for other scoping issues related to `my()` and `local()`, or *perlref* regarding closures.

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 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'};
```

You can use this to print out all the variables in a package, for instance. Here is ***dumpvar.pl*** from the Perl library:

```
   package dumpvar;
   sub main::dumpvar {
       ($package) = @_;
       local(*stab) = eval("*${package}::");
       while (($key,$val) = each(%stab)) {
           local(*entry) = $val;
           if (defined $entry) {
               print "\$$key = '$entry'\n";
           }

           if (defined @entry) {
               print "\@$key = (\n";
               foreach $num ($[ .. $#entry) {
                   print "  $num\t'",$entry[$num],"'\n";
               }
               print ")\n";
           }

           if ($key ne "${package}::" && defined %entry) {
               print "\%$key = (\n";
               foreach $key (sort keys(%entry)) {
                   print "  $key\t'",$entry{$key},"'\n";
               }
               print ")\n";
           }
       }
   }
```

Note that even though the subroutine is compiled in package `dumpvar`, the name of the subroutine is qualified so that its name is inserted into package `main`.

Assignment to a typeglob performs an aliasing operation, i.e.,

```
    *dick = *richard;
```

causes variables, subroutines, and file handles accessible via the identifier `richard` to also be accessible via the identifier `dick`. If you want to alias only a particular variable or subroutine, you can assign a reference instead:

```
    *dick = \$richard;
```

makes `$richard` and `$dick` the same variable, but leaves @richard and @dick as separate arrays. Tricky, eh?

This mechanism may be used to pass and return cheap references into or from subroutines if you won't want to copy the whole thing.

```
    %some_hash = ();
    *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 won‘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.

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 which is 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*.

Package Constructors and Destructors

There are two special subroutine definitions that function as package constructors and destructors. These are the `BEGIN` and `END` routines. The `sub` is optional for these routines.

A `BEGIN` subroutine is executed as soon as possible, that is, the moment it is completely defined, even before the rest of the containing file is parsed. You may have multiple `BEGIN` blocks within a file—they will execute in order of definition. Because a `BEGIN` 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 file.

An `END` subroutine is executed as late as possible, that is, when the interpreter is being exited, even if it is exiting as a result of a `die()` function. (But not if it‘s is 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).

Inside an `END` subroutine `$?` contains the value that the script is going to pass to `exit()`. You can modify `$?` to change the exit value of the script. Beware of changing `$?` by accident (e.g.,, by running something via `system`).

Note that when you use the **−n** and **−p** switches to Perl, `BEGIN` and `END` work just as they do in **awk**, as a degenerate case.

Perl Classes

There is no special class syntax in Perl, but a package may function as a class if it provides subroutines that function as methods. Such a package may also derive some of its methods from another class package by listing the other package name in its @ISA array.

For more on this, see *perlobj*.

Perl Modules

A module is just a package that is defined in a library file of the same name, and is designed to be reusable. 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 explicit exportation of any symbols. Or it can do a little of both.

For example, to start a normal 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;

BEGIN {
    use Exporter   ();
    use vars       qw($VERSION @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS);

    # set the version for version checking
    $VERSION     = 1.00;
    # if using RCS/CVS, this may be preferred
    $VERSION = do { my @r = (q$Revision: 2.21 $ =~ /\d+/g); sprintf "%d."."%02d"

    @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);
}
use vars      @EXPORT_OK;

# non-exported package globals go here
use vars      qw(@more $stuff);

# initalize 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)
```

Then go on to declare and use your variables in functions without any qualifications. See *Exporter* and the *Perl Modules File* for details on mechanics and style issues in module creation.

Perl modules are included into your program by saying

```
    use Module;
```

or

```
    use Module LIST;
```

This is exactly equivalent to

```
    BEGIN { require "Module.pm"; import Module; }
```

or

```
    BEGIN { require "Module.pm"; import Module LIST; }
```

As a special case

```
    use Module ();
```

is exactly equivalent to

```
    BEGIN { require "Module.pm"; }
```

All Perl module files have the extension ***.pm***. `use` assumes this so that 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).

Because the `use` statement implies a `BEGIN` block, the importation of semantics happens at the moment 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 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;`.

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 makes 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 or autoloaded subroutine definitions 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. The POSIX module happens to do both dynamic loading and autoloading, but the user can say just `use POSIX` to get it all.

For more information on writing extension modules, see *perlxs* and *perlguts*.

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.

THE PERL MODULE LIBRARY

A number of modules are included the Perl distribution. These are described below, and all end in ***.pm***. 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 module.) The **pl2pm** file in the distribution may help in your conversion, but it‘s just a mechanical process and therefore far from bulletproof.

Pragmatic Modules

They work somewhat like pragmas 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 locally scoped, so an inner BLOCK may countermand any of these by saying:

```
    no integer;
    no strict 'refs';
```

which lasts until the end of that BLOCK.

Unlike the pragmas that effect the `$^H` hints variable, the `use vars` and `use subs` declarations are not BLOCK−scoped. They allow you to pre−declare 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).

blib	manipulate @INC at compile time to use MakeMaker‘s uninstalled version of a package
diagnostics	force verbose warning diagnostics
integer	compute arithmetic in integer instead of double
less	request less of something from the compiler
lib	manipulate @INC at compile time
locale	use or ignore current locale for built−in operations (see *perllocale*)
ops	restrict named opcodes when compiling or running Perl code
overload	overload basic Perl operations
sigtrap	enable simple signal handling
strict	restrict unsafe constructs
subs	pre−declare sub names
vmsish	adopt certain VMS−specific behaviors
vars	pre−declare global variable names

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.

AnyDBM_File	provide framework for multiple DBMs
AutoLoader	load functions only on demand
AutoSplit	split a package for autoloading
Benchmark	benchmark running times of code
CPAN	interface to Comprehensive Perl Archive Network
CPAN::FirstTime	create a CPAN configuration file
CPAN::Nox	run CPAN while avoiding compiled extensions
Carp	warn of errors (from perspective of caller)
Class::Template	struct/member template builder
Config	access Perl configuration information
Cwd	get pathname of current working directory
DB_File	access to Berkeley DB
Devel::SelfStubber	generate stubs for a SelfLoading module
DirHandle	supply object methods for directory handles
DynaLoader	dynamically load C libraries into Perl code
English	use nice English (or awk) names for ugly punctuation variables
Env	import environment variables
Exporter	implements default import method for modules
ExtUtils::Embed	utilities for embedding Perl in C/C++ applications
ExtUtils::Install	install files from here to there
ExtUtils::Liblist	determine libraries to use and how to use them
ExtUtils::MM_OS2	methods to override UN*X behaviour in ExtUtils::MakeMaker
ExtUtils::MM_Unix	methods used by ExtUtils::MakeMaker
ExtUtils::MM_VMS	methods to override UN*X behaviour in ExtUtils::MakeMaker
ExtUtils::MakeMaker	create an extension Makefile
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::testlib	add blib/* directories to @INC
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::Find	traverse a file tree
File::Path	create or remove a series of directories
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
FindBin	locate directory of original perl script
GDBM_File	access to the gdbm library
Getopt::Long	extended processing of command line options
Getopt::Std	process single−character switches with switch clustering
I18N::Collate	compare 8−bit scalar data according to the current locale
IO	load various IO modules
IO::File	supply object methods for filehandles
IO::Handle	supply object methods for I/O handles
IO::Pipe	supply object methods for pipes
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
IPC::Open2	open a process for both reading and writing
IPC::Open3	open a process for reading, writing, and error handling
Math::BigFloat	arbitrary length float math package
Math::BigInt	arbitrary size integer math package
Math::Complex	complex numbers and associated mathematical functions
NDBM_File	tied access to ndbm files

Net::Ping	Hello, anybody home?
Net::hostent	by−name interface to Perl‘s built−in `gethost*()` functions
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
Opcode	disable named opcodes when compiling or running perl code
Pod::Text	convert POD data to formatted ASCII text
POSIX	interface to IEEE Standard 1003.1
SDBM_File	tied access to sdbm files
Safe	compile and execute code in restricted compartments
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
Symbol	manipulate Perl symbols and their names
Sys::Hostname	try every conceivable way to get hostname
Sys::Syslog	interface to the UNIX syslog(3) calls
Term::Cap	termcap interface
Term::Complete	word completion module
Term::ReadLine	interface to various `readline` packages
Test::Harness	run perl standard test scripts with statistics
Text::Abbrev	create an abbreviation table from a list
Text::ParseWords	parse text into an array of tokens
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
Tie::Hash	base class definitions for tied hashes
Tie::RefHash	base class definitions for tied hashes with references as keys
Tie::Scalar	base class definitions for tied scalars
Tie::SubstrHash	fixed−table−size, fixed−key−length hashing

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)
User::grent	by−name interface to Perl's built−in `getgr*()` functions
User::pwent	by−name interface to Perl's built−in `getpw*()` functions

To find out *all* the modules installed on your system, including those without documentation or outside the standard release, do this:

```
    find `perl −e 'print "@INC"'` −name '*.pm' −print
```

They should all have their own documentation installed and accessible via your system man(1) command. If that fails, try the *perldoc* program.

Extension Modules

Extension modules are written in C (or a mix of Perl and C) and get dynamically loaded into Perl if and when you need them. Supported extension modules include the Socket, Fcntl, and POSIX modules.

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 in archie(1L), the Perl FAQ or Meta−FAQ, the WWW page, and even with their authors before randomly posting asking for their present condition and disposition.

CPAN

CPAN stands for the Comprehensive Perl Archive Network. This is a globally replicated collection of all known Perl materials, including hundreds of unbundled modules. Here are the major categories of modules:

- 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 registered CPAN sites as of this writing include the following. You should try to choose one close to you:

- Africa

```
    South Africa    ftp://ftp.is.co.za/programming/perl/CPAN/
```

- Asia

```
    Hong Kong       ftp://ftp.hkstar.com/pub/CPAN/
    Japan           ftp://ftp.jaist.ac.jp/pub/lang/perl/CPAN/
                    ftp://ftp.lab.kdd.co.jp/lang/perl/CPAN/
    South Korea     ftp://ftp.nuri.net/pub/CPAN/
    Taiwan          ftp://dongpo.math.ncu.edu.tw/perl/CPAN/
                    ftp://ftp.wownet.net/pub2/PERL/
```

- Australasia

```
    Australia       ftp://ftp.netinfo.com.au/pub/perl/CPAN/
    New Zealand     ftp://ftp.tekotago.ac.nz/pub/perl/CPAN/
```

- Europe

```
    Austria         ftp://ftp.tuwien.ac.at/pub/languages/perl/CPAN/
    Belgium         ftp://ftp.kulnet.kuleuven.ac.be/pub/mirror/CPAN/
    Czech Republic  ftp://sunsite.mff.cuni.cz/Languages/Perl/CPAN/
    Denmark         ftp://sunsite.auc.dk/pub/languages/perl/CPAN/
    Finland         ftp://ftp.funet.fi/pub/languages/perl/CPAN/
    France          ftp://ftp.ibp.fr/pub/perl/CPAN/
                    ftp://ftp.pasteur.fr/pub/computing/unix/perl/CPAN/
    Germany         ftp://ftp.gmd.de/packages/CPAN/
                    ftp://ftp.leo.org/pub/comp/programming/languages/perl/CPAN/
                    ftp://ftp.mpi-sb.mpg.de/pub/perl/CPAN/
                    ftp://ftp.rz.ruhr-uni-bochum.de/pub/CPAN/
                    ftp://ftp.uni-erlangen.de/pub/source/Perl/CPAN/
                    ftp://ftp.uni-hamburg.de/pub/soft/lang/perl/CPAN/
    Greece          ftp://ftp.ntua.gr/pub/lang/perl/
    Hungary         ftp://ftp.kfki.hu/pub/packages/perl/CPAN/
    Italy           ftp://cis.utovrm.it/CPAN/
    the Netherlands ftp://ftp.cs.ruu.nl/pub/PERL/CPAN/
                    ftp://ftp.EU.net/packages/cpan/
    Norway          ftp://ftp.uit.no/pub/languages/perl/cpan/
    Poland          ftp://ftp.pk.edu.pl/pub/lang/perl/CPAN/
                    ftp://sunsite.icm.edu.pl/pub/CPAN/
    Portugal        ftp://ftp.ci.uminho.pt/pub/lang/perl/
                    ftp://ftp.telepac.pt/pub/CPAN/
    Russia          ftp://ftp.sai.msu.su/pub/lang/perl/CPAN/
```

Slovenia	ftp://ftp.arnes.si/software/perl/CPAN/
Spain	ftp://ftp.etse.urv.es/pub/mirror/perl/
	ftp://ftp.rediris.es/mirror/CPAN/
Sweden	ftp://ftp.sunet.se/pub/lang/perl/CPAN/
UK	ftp://ftp.demon.co.uk/pub/mirrors/perl/CPAN/
	ftp://sunsite.doc.ic.ac.uk/packages/CPAN/
	ftp://unix.hensa.ac.uk/mirrors/perl−CPAN/

- North America

Ontario	ftp://ftp.utilis.com/public/CPAN/
	ftp://enterprise.ic.gc.ca/pub/perl/CPAN/
Manitoba	ftp://theory.uwinnipeg.ca/pub/CPAN/
California	ftp://ftp.digital.com/pub/plan/perl/CPAN/
	ftp://ftp.cdrom.com/pub/perl/CPAN/
Colorado	ftp://ftp.cs.colorado.edu/pub/perl/CPAN/
Florida	ftp://ftp.cis.ufl.edu/pub/perl/CPAN/
Illinois	ftp://uiarchive.uiuc.edu/pub/lang/perl/CPAN/
Massachusetts	ftp://ftp.iguide.com/pub/mirrors/packages/perl/CPAN/
New York	ftp://ftp.rge.com/pub/languages/perl/
North Carolina	ftp://ftp.duke.edu/pub/perl/
Oklahoma	ftp://ftp.ou.edu/mirrors/CPAN/
Oregon	http://www.perl.org/CPAN/
	ftp://ftp.orst.edu/pub/packages/CPAN/
Pennsylvania	ftp://ftp.epix.net/pub/languages/perl/
Texas	ftp://ftp.sedl.org/pub/mirrors/CPAN/
	ftp://ftp.metronet.com/pub/perl/

- South America

Chile	ftp://sunsite.dcc.uchile.cl/pub/Lang/perl/CPAN/

For an up−to−date listing of CPAN sites, see ***http://www.perl.com/perl/CPAN*** or ***ftp://ftp.perl.com/perl/***.

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.

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.

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 hardwired 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 which 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 which need less strictness. Always use **−w**. Always use **−w**! Follow the guidelines in the perlstyle(1) manual.

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.

To be portable each component of a module name should be limited to 11 characters. If it might be used on 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 license: The GNU GPL and The Artistic License (see the files README, Copying, and Artistic). 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.pm in Perl5.001m or later 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 you should place the module into a major ftp archive and include details of its location in your announcement.

Some notes about ftp archives: Please use a long descriptive file name which 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://franz.ww.tu−berlin.de/modulelist
```

or upload to one of these sites:

```
ftp://franz.ww.tu−berlin.de/incoming
ftp://ftp.cis.ufl.edu/incoming
```

and notify <***upload@franz.ww.tu−berlin.de***.

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 (see 2.2 above) Otherwise try to add a mechanism to revert to the old behaviour if people rely on it. Document incompatible changes.

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 the perl applications which 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!

Guidelines for Reusing Application Code

Complete applications rarely belong in the Perl Module Library.

Many applications contain some perl code which 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)
```

NAME

perlform − Perl formats

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 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 name "STDOUT", and the default format for filehandle TEMP is name "TEMP". They just look the same. They aren‘t.

Output record formats are declared as follows:

```
format NAME =
FORMLIST
.
```

If name is omitted, format "STDOUT" is defined. FORMLIST consists of a sequence of lines, each of which may be of 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 are printed exactly as they look, except for certain fields that substitute values into the line. Each field in a picture line starts with either "@" (at) or "^" (caret). These lines do not undergo any kind of variable interpolation. The at field (not to be confused with the array marker @) is the normal kind of field; the other kind, caret fields, are used to do rudimentary multi−line text block filling. The length of the field is supplied by padding out the field with multiple "<", ">", or "|" characters to specify, respectively, left justification, right justification, or centering. If the variable would exceed the width specified, it is truncated.

As an alternate form of right justification, you may also use "#" characters (with an optional ".") to specify a numeric field. This way you can line up the decimal points. If any value supplied for these fields contains a newline, only the text up to the newline is printed. Finally, the special field "@*" can be used for printing multi−line, non−truncated values; it should appear by itself on a line.

The values are specified on the following line in the same order as the picture fields. The expressions providing the values should be separated by commas. The expressions 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 *"WARNINGS"* for more information.

Picture fields that begin with ^ rather than @ are treated specially. With a # field, the field is blanked out if the value is undefined. For other field types, the caret enables a kind of fill mode. Instead of an arbitrary expression, the value supplied must be a scalar variable name that contains a text string. Perl puts as much text as it can 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 returned.) Normally you would use a sequence of fields in a vertical stack 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. 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.

Using caret fields can produce variable length records. If the text to be formatted is short, you can suppress blank lines by putting a "~" (tilde) character anywhere in the line. The tilde will be translated to a space upon output. If you put a second tilde contiguous to the first, the line will be repeated until all the fields on the line are exhausted. (If you use a field of the at variety, the expression you supply had better not give the same value every time forever!)

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*.

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.

Format Variables

The current format name is stored in the variable `$~` (`$FORMAT_NAME`), and the current top of form format name is in `$^` (`$FORMAT_TOP_NAME`). The current output page number is stored in `$%` (`$FORMAT_PAGE_NUMBER`), and the number of lines on the page is in `$=` (`$FORMAT_LINES_PER_PAGE`). Whether to autoflush output on this handle is stored in `$|` (`$OUTPUT_AUTOFLUSH`). The string output before each top of page (except the first) is stored in `$^L` (`$FORMAT_FORMFEED`). These variables are set on a per−filehandle basis, so you‘ll need to `select()` into a different one to affect them:

```
    select((select(OUTF),
            $~ = "My_Other_Format",
            $^ = "My_Top_Format"
           )[0]);
```

Pretty ugly, eh? It‘s a common idiom though, so don‘t be too surprised when you see it. You can at least use a temporary variable to hold the previous filehandle: (this is a much better approach in general, because not only does legibility improve, you now have intermediary stage in the expression to single−step the debugger through):

```
    $ofh = select(OUTF);
    $~ = "My_Other_Format";
    $^ = "My_Top_Format";
    select($ofh);
```

If you use the English module, you can even read the variable names:

```
    use English;
    $ofh = select(OUTF);
    $FORMAT_NAME     = "My_Other_Format";
    $FORMAT_TOP_NAME = "My_Top_Format";
    select($ofh);
```

But you still have those funny `select()`s. So just use the FileHandle module. Now, you can access these special variables using lowercase method names instead:

```
    use FileHandle;
    format_name     OUTF "My_Other_Format";
    format_top_name OUTF "My_Top_Format";
```

Much better!

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;
 }
```

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(MESELF, "|-")` (see *open()*) and always `write()` to MESELF instead of STDOUT. Have your child process massage its STDIN to rearrange headers and footers however you like. Not very convenient, but doable.

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;
```

WARNINGS

The lone dot that ends a format can also prematurely end an email 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 email, you should indent it so that the format−ending dot is not on the left margin; this will prevent email 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 which 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.

NAME

perllocale − Perl locale handling (internationalization and localization)

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 very naïve to think that `A−Za−z` defines all the "letters". 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 *"NOTES"*.

PREPARING TO USE LOCALES

If Perl applications are to be able 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 the locales which 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, 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 which 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 L<The use locale pragma) where appropriate, and **at least one** of the following must be true:

- **The locale−determining environment variables (see *"ENVIRONMENT"*) must be correctly set up**, either by yourself, or by the person who set up your system account, at the time the application is started.
- **The application must set its own locale** using the method described in *The setlocale function*.

USING LOCALES

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 it is used without an explicit comparison function because it uses `cmp` by default.

  Note: `eq` and `ne` are unaffected by the locale: they always perform a byte−by−byte 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 byte−by−byte comparison, and only returns (equal) if the operands are bit−for−bit 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 returns with `no locale` or on reaching the end of the enclosing block.

Note that the string result of any operation that uses locale information is tainted, as it is possible for a locale to be untrustworthy. See *"SECURITY"*.

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 *"ENVIRONMENT"*. 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, 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 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 use this in a subsequent 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 that 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 which was in force when Perl started up: changes to the environment made by the application after start−up may or may not be noticed, depending on the implementation of 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)*. For the locales available in your system, also consult *setlocale(3)* and see whether it leads you to the list of the 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
```

and see whether they list something resembling these

```
en_US.ISO8859-1     de_DE.ISO8859-1     ru_RU.ISO8859-5
en_US               de_DE               ru_RU
en                  de                  ru
english             german              russian
english.iso88591    german.iso88591     russian.iso88595
```

Sadly, even though the calling interface for `setlocale()` has been standardized, the names of the locales and the directories where the configuration is, have not. The basic form of the name is *language_country/territory***.***codeset*, but the latter parts are not always present.

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 and the second by the POSIX standard. What they define is 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.

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 formatting variable names such as `decimal_point` and `thousands_sep`; the values are the corresponding values. See *localeconv* for a longer example, which lists all the categories an implementation might be expected to provide; some provide more and others fewer, however. Note that you don‘t need `use locale`: as a function with the job of querying the locale, `localeconv()` always observes the current locale.

Here‘s a simple−minded example program which rewrites its command line parameters as integers formatted correctly 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 = 3 unless $grouping;

        # Format command line params for current locale
        for (@ARGV) {
            $_ = int;    # Chop non−integer part
            1 while
            s/(\d)(\d{$grouping}($|$thousands_sep))/$1$thousands_sep$2/;
            print "$_";
        }
        print "\n";
```

LOCALE CATEGORIES

The subsections which follow describe basic locale categories. As well as these, there are some combination categories which allow the manipulation of more than one basic category at a time. See *"ENVIRONMENT"* for a discussion of these.

Category LC_COLLATE: Collation

When in the scope of `use locale`, Perl looks to the `LC_COLLATE environment variable to determine the application‘s notions on the collation (ordering) of characters. (‘b’ follows ‘a’ in Latin alphabets, but where do ‘á’ and ‘å’ belong?)`

Here is a code snippet that will tell you what are the alphanumeric characters in the current locale, in the locale 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 byte−by−byte comparison for strings which 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 which ignores space characters completely, and which folds case.

If you have a single string which 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 byte−by−byte comparisons against other transformed strings during collation. "Under the hood", locale−affected Perl comparison operators call `strxfrm()` for both their operands, then do a byte−by−byte 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. In fact, it doesn‘t save anything: Perl magic (see *Magic Variables*) creates the transformed version of a string the first time it‘s needed in a comparison, then keeps it 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. And 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, as it isn‘t needed: `strcoll()` and `strxfrm()` exist only to generate locale−dependent results, and so always obey the current `LC_COLLATE` locale.

Category LC_CTYPE: Character Types

When 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 and numeric characters. (Consult` *`perlre`* `for more information about regular expressions.) Thanks to LC_CTYPE, depending on your locale setting, characters like ‘æ‘, ‘ð‘, ‘‘, and ‘ø’ may be understood as \w characters.`

The `LC_CTYPE` locale also provides the map used in translating 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 in `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 (unaccented) letters and digits − for example, in command strings − locale−aware applications should use `\w` inside a `no locale` block. See *"SECURITY"*.

Category LC_NUMERIC: Numeric Formatting

When in the scope of `use locale`, Perl obeys the `LC_NUMERIC locale information, which controls 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.)`

Note that output produced by `print()` is **never** affected by the current locale: it is independent of 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-independent conversion to string

        print "half five is $n\n";       # Locale-independent 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
```

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, though voluminous and complex, does not quite meet your requirements: currency formatting is a hard nut to crack.

LC_TIME

The 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 the 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 which exists only to generate locale−dependent results, `strftime()` always obeys the current `LC_TIME` locale.

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 which are not part of the standard Perl distribution.

SECURITY

While 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 which 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 which 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.
- If the decimal point character in the `LC_NUMERIC` locale is surreptitiously changed from a dot to a comma, `sprintf("%g", 0.123456e3)` produces a string result of "123,456". Many people would interpret this as one hundred and twenty−three thousand, four hundred and fifty−six.

- A sneaky `LC_COLLATE` locale could result in the names of students with "D" grades appearing ahead of those with "A"s.
- An application which takes the trouble to use the information in `LC_MONETARY` may format debits as if they were credits and vice versa if that locale has been subverted. Or it make may 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 maliciously be modified presents similar challenges. Similarly, they are not specific to Perl: any programming language which allows you to write programs which take account of their environment exposes you to these issues.

Perl cannot protect you from all of the 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 which become locale−dependent, and which may be untrustworthy in consequence. Here is a summary of the tainting behavior of operators and functions which 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 an array−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 it is 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.

In−memory formatting function (`sprintf()`):
: Result is tainted if "use locale" is in effect.

Output formatting functions (`printf()` and `write()`):
: Success/failure result is never tainted.

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 very similar program which is locale−aware:

```
        #/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` when `use locale` is in effect.

ENVIRONMENT

PERL_BADLANG

A string that can suppress Perl‘s warning about failed locale settings at start−up. Failure can occur if the locale support in the operating system is lacking (broken) is 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 which 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 it is set, it overrides all the rest of the locale environment variables.

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_...`.

NOTES

Backward compatibility

Versions of Perl prior to 5.004 **mostly** ignored locale information, generally behaving as if something similar to the `"C"` locale (see *The setlocale function*) was always in force, even if the program environment suggested otherwise. By default, Perl still behaves this way so as to maintain backward compatibility. If you want a Perl application to pay attention to locale information, you **must** use the
`use locale` pragma (see L<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 that was available, that is, `\w` did understand what are 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.

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`.

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.

`write()` and LC_NUMERIC

Formats are the only part of Perl which 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.

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 the installation of arbitrary locales, you may find the definitions useful as they are, or as a basis for the development of your own locales.

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**.

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.

BUGS

Broken systems

In certain system environments 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.com***, and complain to your vendor: maybe some bug fixes exist for these problems in your operating system. Sometimes such bug fixes are called an operating system upgrade.

SEE ALSO

isalnum, *isalpha*, *isdigit*, *isgraph*, *islower*, *isprint*, *ispunct*, *isspace*, *isupper*, *isxdigit*, *localeconv*, *setlocale*, *strcoll*, *strftime*, *strtod*, *strxfrm*

HISTORY

Jarkko Hietaniemi‘s original ***perli18n.pod*** heavily hacked by Dominic Dunlop, assisted by the perl5−porters.

Last update: Wed Jan 22 11:04:58 EST 1997

NAME

perlref − Perl references and nested data structures

DESCRIPTION

Before release 5 of Perl it was difficult to represent complex data structures, because all references had to be symbolic, and even that was difficult to do when you wanted to refer to a variable rather than a symbol table entry. Perl not only makes it easier to use symbolic references to variables, but lets you have "hard" references to any piece of data. 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. (Note: The 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.)

A symbolic reference contains the name of a variable, just as a symbolic link in the filesystem contains merely the name of a file. The `*glob` notation is a kind of symbolic reference. Hard references are more like hard links in the file system: merely another way at getting at the same underlying object, irrespective of its name.

"Hard" 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 scalar. It doesn‘t magically start being an array or a hash unless you tell it so explicitly by dereferencing it.

References can be constructed several ways.

1. By using the backslash operator on a variable, subroutine, or value. (This works much like the & (address−of) operator works in C.) Note that 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. See the explanation of the *foo{THING} syntax below. (However, you‘re apt to find Perl code out there using globrefs as though they were IO handles, which is grandfathered into continued functioning.)

2. A reference to an anonymous array can be constructed using square brackets:

```
    $arrayref = [1, 2, ['a', 'b', 'c']];
```

   Here we‘ve constructed a reference to an anonymous array of three elements whose final element is itself 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".)

   Note that 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`.

3. A reference to an anonymous hash can be constructed using curly brackets:

```
    $hashref = {
        'Adam'  => 'Eve',
        'Clyde' => 'Bonnie',
    };
```

Anonymous hash and array constructors 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
```

4. A reference to an anonymous subroutine can be constructed by using `sub` without a subname:

```
    $coderef = sub { print "Boink!\n" };
```

Note the presence of the semicolon. Except for the fact that the code inside isn't executed immediately, 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 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 visible lexically 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 of 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 provides a different mechanism to do that already—see *perlobj*.

You can also think of closure as a way to write a subroutine template without using eval. (In fact, in version 5.000, eval was the *only* way to get closures. You may wish to use "require 5.001" if you use closures.)

Here's a small example of how closures works:

```
    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 `$x` continues to refer to the value passed into `newprint()` *despite* the fact that the "my `$x`" has seemingly gone out of scope by the time the anonymous subroutine runs. That's what closure is all about.

This applies to only 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 kind 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 customarily named `new()`, but don't have to be:

```
$objref = new Doggie (Tail => 'short', Ears => 'long');
```

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*), sockets (*socket* and *socketpair*), and directory handles (*opendir*). For compatibility with previous versions of Perl, *foo{FILEHANDLE} is a synonym for *foo{IO}.

*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.

The use of *foo{IO} is the best way to pass bareword filehandles into or out of subroutines, or to store them in larger data structures.

```
splutter(*STDOUT{IO});
sub splutter {
    my $fh = shift;
    print $fh "her um well a hmmm\n";
}

$rec = get_rec(*STDIN{IO});
sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

Beware, though, that you can‘t do this with a routine which is going to open the filehandle for you, because *HANDLE{IO} will be undef if HANDLE hasn‘t been used yet. Use *HANDLE for that sort of thing instead.

Using *HANDLE (or *HANDLE) is another way to use and store non−bareword filehandles (before perl version 5.002 it was the only way). The two methods are largely interchangeable, you can do

```
splutter(\*STDOUT);
$rec = get_rec(\*STDIN);
```

with the above subroutine definitions.

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. The case of individual array elements arises often enough that it gets cumbersome to use method 2. As a form of syntactic sugar, the two lines like that above can be written:

   ```
   $arrayref->[0] = "January";
   $hashref->{"KEY"} = "VALUE";
   ```

   The left side of the array 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.

   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.

The `ref()` operator may be used to determine what type of thing the reference is pointing to. See *perlfunc*.

The `bless()` operator may be used to associate a reference 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 kind 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";
```

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 in this case, 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"} # Sets $THAT::foo without eval
```

This is very 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 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.

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", despite the fact that 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.) Note that 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 **−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.

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.

SEE ALSO

Besides the obvious documents, source code can be instructive. Some rather 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 *perlobj* for how to use them to create objects.

NAME

perldsc − Perl Data Structures Cookbook

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 `$m{$LoL,$b}` notation borrowed from *awk* in which the keys are actually more like a single concatenated string `"$LoL$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 a array with three dimensions!

```
for $x (1 .. 10) {
    for $y (1 .. 10) {
        for $z (1 .. 10) {
            $LoL[$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 @LoL`? How do you sort it? How can you pass it to a function or get one of these back from a function? Is is 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 some of the general issues common to all of these types of data structures.

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 a 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.

```
    $list[7][12]                        # array of arrays
    $list[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:

```
    @LoL = ( [2, 3], [4, 5, 7], [0] );
    print $LoL[1][2];
  7
    print @LoL;
  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]`.

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 $i (1..10) {
        @list = somefunc($i);
        $LoL[$i] = @list;       # WRONG!
    }
```

That‘s just the simple case of assigning a list 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 $i (1..10) {
        @list = somefunc($i);
        $counts[$i] = scalar @list;
    }
```

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

```
    for $i (1..10) {
        @list = somefunc($i);
        $LoL[$i] = \@list;      # 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 @LoL refer to the *very same place*, and they will therefore all hold whatever was last in @list! 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 $i (1..10) {
        @list = somefunc($i);
        $LoL[$i] = [ @list ];
    }
```

The square brackets make a reference to a new array with a *copy* of what's in @list 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) {
        @list = 0 .. $i;
        @{$LoL[$i]} = @list;
    }
```

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 `@{$LoL[$i]}}` dereference on the left−hand−side of the assignment. It all depends on whether `$LoL[$i]` had been undefined to start with, or whether it already contained a reference. If you had already populated @LoL with references, as in

```
    $LoL[3] = \@another_list;
```

Then the assignment with the indirection on the left−hand−side would use the existing reference that was already there:

```
    @{$LoL[3]} = @list;
```

Of course, this *would* have the "interesting" effect of clobbering @another_list. (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 $i (1..10) {
        my @list = somefunc($i);
        $LoL[$i] = \@list;
    }
```

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:

```
    $LoL[$i] = [ @list ];       # usually best
    $LoL[$i] = \@list;          # perilous; just how my() was that list?
    @{ $LoL[$i] } = @list;      # way too tricky for most programmers
```

CAVEAT ON PRECEDENCE

Speaking of things like `@{$LoL[$i]},` the following are actually the same thing:

```
    $listref−>[2][2]    # clear
    $$listref[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, `$$listref[$i]` first does the deref of `$listref,` making it take `$listref` as a reference to an array, and then dereference that, and finally tell you the *i‘th* value of the array pointed to by `$LoL.` If you wanted the C notion, you‘d have to write `${$LoL[$i]}` to force the `$LoL[$i]` to get evaluated first before the leading `$` dereferencer.

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 $listref = [
        [ "fred", "barney", "pebbles", "bambam", "dino", ],
        [ "homer", "bart", "marge", "maggie", ],
        [ "george", "jane", "elroy", "judy", ],
    ];

    print $listref[2][2];
```

The compiler would immediately flag that as an error *at compile time*, because you were accidentally accessing `@listref`, an undeclared variable, and it would thereby remind you to write instead:

```
    print $listref−>[2][2]
```

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 `$LoL` above, here‘s the debugger output:

```
    DB<1> X $LoL
    $LoL = 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'
```

There's also a lowercase **x** command which is nearly the same.

CODE EXAMPLES

Presented with little comment (these will get their own man pages someday) here are short code examples illustrating access of various types of data structures.

LISTS OF LISTS

Declaration of a LIST OF LISTS

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

Generation of a LIST OF LISTS

```
    # reading from file
    while ( <> ) {
        push @LoL, [ split ];
    }

    # calling a function
    for $i ( 1 .. 10 ) {
        $LoL[$i] = [ somefunc($i) ];
    }

    # using temp vars
    for $i ( 1 .. 10 ) {
        @tmp = somefunc($i);
        $LoL[$i] = [ @tmp ];
    }

    # add to an existing row
    push @{ $LoL[0] }, "wilma", "betty";
```

Access and Printing of a LIST OF LISTS

```
    # one element
    $LoL[0][0] = "Fred";

    # another element
    $LoL[1][1] =~ s/(\w)/\u$1/;

    # print the whole thing with refs
    for $aref ( @LoL ) {
        print "\t [ @$aref ],\n";
    }

    # print the whole thing with indices
    for $i ( 0 .. $#LoL ) {
        print "\t [ @{$LoL[$i]} ],\n";
    }

    # print the whole thing one at a time
    for $i ( 0 .. $#LoL ) {
        for $j ( 0 .. $#{ $LoL[$i] } ) {
            print "elt $i $j is $LoL[$i][$j]\n";
        }
    }
```

HASHES OF LISTS

Declaration of a HASH OF LISTS

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

Generation of a HASH OF LISTS

```
    # reading from file
    # flintstones: fred barney wilma dino
    while ( <> ) {
        next unless s/^(.*?):\s*//;
        $HoL{$1} = [ split ];
    }

    # reading from file; more temps
    # flintstones: fred barney wilma dino
    while ( $line = <> ) {
        ($who, $rest) = split /:\s*/, $line, 2;
        @fields = split ' ', $rest;
        $HoL{$who} = [ @fields ];
    }

    # calling a function that returns a list
    for $group ( "simpsons", "jetsons", "flintstones" ) {
        $HoL{$group} = [ get_family($group) ];
    }

    # likewise, but using temps
    for $group ( "simpsons", "jetsons", "flintstones" ) {
        @members = get_family($group);
        $HoL{$group} = [ @members ];
    }

    # append new members to an existing family
    push @{ $HoL{"flintstones"} }, "wilma", "betty";
```

Access and Printing of a HASH OF LISTS

```
    # one element
    $HoL{flintstones}[0] = "Fred";

    # another element
```

```
$HoL{simpsons}[1] =~ s/(\w)/\u$1/;

# print the whole thing
foreach $family ( keys %HoL ) {
    print "$family: @{ $HoL{$family} }\n"
}

# print the whole thing with indices
foreach $family ( keys %HoL ) {
    print "family: ";
    foreach $i ( 0 .. $#{ $HoL{$family} } ) {
        print " $i = $HoL{$family}[$i]";
    }
    print "\n";
}

# print the whole thing sorted by number of members
foreach $family ( sort { @{$HoL{$b}} <=> @{$HoL{$a}} } keys %HoL ) {
    print "$family: @{ $HoL{$family} }\n"
}

# print the whole thing sorted by number of members and name
foreach $family ( sort {
                           @{$HoL{$b}} <=> @{$HoL{$a}}
                                       ||
                                   $a cmp $b
           } keys %HoL )
{
    print "$family: ", join(", ", sort @{ $HoL{$family}}), "\n";
}
```

LISTS OF HASHES

Declaration of a LIST OF HASHES

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

Generation of a LIST OF HASHES

```
# reading from file
# format: LEAD=fred FRIEND=barney
while ( <> ) {
    $rec = {};
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
```

```
            $rec->{$key} = $value;
        }
        push @LoH, $rec;
    }

    # reading from file
    # format: LEAD=fred FRIEND=barney
    # no temp
    while ( <> ) {
        push @LoH, { split /[\s+=]/ };
    }

    # calling a function  that returns a key,value list, like
    # "lead","fred","daughter","pebbles"
    while ( %fields = getnextpairset() ) {
        push @LoH, { %fields };
    }

    # likewise, but using no temp vars
    while (<>) {
        push @LoH, { parsepairs($_) };
    }

    # add key/value to an element
    $LoH[0]{pet} = "dino";
    $LoH[2]{pet} = "santa's little helper";
```

Access and Printing of a LIST OF HASHES

```
    # one element
    $LoH[0]{lead} = "fred";

    # another element
    $LoH[1]{lead} =~ s/(\w)/\u$1/;

    # print the whole thing with refs
    for $href ( @LoH ) {
        print "{ ";
        for $role ( keys %$href ) {
            print "$role=$href->{$role} ";
        }
        print "}\n";
    }

    # print the whole thing with indices
    for $i ( 0 .. $#LoH ) {
        print "$i is { ";
        for $role ( keys %{ $LoH[$i] } ) {
            print "$role=$LoH[$i]{$role} ";
        }
        print "}\n";
    }

    # print the whole thing one at a time
    for $i ( 0 .. $#LoH ) {
        for $role ( keys %{ $LoH[$i] } ) {
            print "elt $i $role is $LoH[$i]{$role}\n";
        }
    }
```

HASHES OF HASHES

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",
       },
);
```

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};
}
```

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";
    }
```

MORE ELABORATE RECORDS

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->{LIST}[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");
```

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

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 examples
# if you wanted a {kids} field that was an array reference
# to a list 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 list, but the list 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";
        }
```

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 *perlmod* for source code to MLDBM.

SEE ALSO

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

AUTHOR

Tom Christiansen <***tchrist@perl.com***

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

NAME

perlLoL − Manipulating Lists of Lists in Perl

DESCRIPTION

Declaration and Access of Lists of Lists

The simplest thing to build is a list of lists (sometimes called an array of arrays). It‘s reasonably easy to understand, and almost everything that applies here will also be applicable later on with the fancier data structures.

A list of lists, or an array of an array if you would, is just a regular old array @LoL that you can get at with two subscripts, like `$LoL[3][2].` Here‘s a declaration of the array:

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

    print $LoL[2][2];
  bart
```

Now you should be very careful that the outer bracket type is a round one, that is, parentheses. That‘s because you‘re assigning to an @list, so you need parentheses. If you wanted there *not* to be an @LoL, but rather just a reference to it, you could do something more like this:

```
    # assign a reference to list of list references
    $ref_to_LoL = [
        [ "fred", "barney", "pebbles", "bambam", "dino", ],
        [ "homer", "bart", "marge", "maggie", ],
        [ "george", "jane", "alroy", "judy", ],
    ];

    print $ref_to_LoL->[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. `$ref_to_LoL` is a reference to an array, whereas @LoL is an array proper. Likewise, `$LoL[2]` is not an array, but an array ref. So how come you can write these:

```
    $LoL[2][2]
    $ref_to_LoL->[2][2]
```

instead of having to write these:

```
    $LoL[2]->[2]
    $ref_to_LoL->[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_LoL` always needs it.

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 @LoL list containing all these, here‘s the right way to do that:

```
    while (<>) {
        @tmp = split;
        push @LoL, [ @tmp ];
    }
```

You might also have loaded that from a function:

```
    for $i ( 1 .. 10 ) {
        $LoL[$i] = [ somefunc($i) ];
    }
```

Or you might have had a temporary variable sitting around with the list in it.

```
    for $i ( 1 .. 10 ) {
        @tmp = somefunc($i);
        $LoL[$i] = [ @tmp ];
    }
```

It‘s very important that you make sure to use the `[]` list reference constructor. That‘s because this will be very wrong:

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

You see, assigning a named list 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(@LoL, @tmp);
    while (<>) {
        @tmp = split;
        push @LoL, [ @tmp ];
    }
```

Of course, you don‘t need the temporary array to have a name at all:

```
    while (<>) {
        push @LoL, [ 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 (@LoL, $i, $line);
    for $i ( 0 .. 10 ) {
        $line = <>;
        $LoL[$i] = [ split ’ ’, $line ];
    }
```

or even just

```
    my (@LoL, $i);
    for $i ( 0 .. 10 ) {
        $LoL[$i] = [ split ’ ’, <> ];
    }
```

You should in general be leery of using potential list functions in a scalar context without explicitly stating such. This would be clearer to the casual reader:

```
    my (@LoL, $i);
    for $i ( 0 .. 10 ) {
        $LoL[$i] = [ split ’ ’, scalar(<>) ];
    }
```

If you wanted to have a `$ref_to_LoL` variable as a reference to an array, you‘d have to do something like this:

```
    while (<>) {
        push @$ref_to_LoL, [ split ];
    }
```

Actually, if you were using strict, you‘d have to declare not only `$ref_to_LoL` as you had to declare @LoL, but you‘d *also* having to initialize it to a reference to an empty list. (This was a bug in perl version 5.001m that‘s been fixed for the 5.002 release.)

```
    my $ref_to_LoL = [];
    while (<>) {
        push @$ref_to_LoL, [ split ];
    }
```

Ok, 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) {
            $LoL[$x][$y] = func($x, $y);
        }
    }

    for $x ( 3, 7, 9 ) {
        $LoL[$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 @{ $LoL[0] }, "wilma", "betty";
```

Notice that I *couldn‘t* say just:

```
    push $LoL[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.

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 $LoL[0][0];
```

If you want to print the whole thing, though, you can‘t say

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

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

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

or maybe even this. Notice the inner loop.

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

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

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

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 @LoL variable as before.

```
    @part = ();
    $x = 4;
    for ($y = 7; $y < 13; $y++) {
        push @part, $LoL[$x][$y];
    }
```

That same loop could be replaced with a slice operation:

```
    @part = @{ $LoL[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:

```
    @newLoL = ();
    for ($startx = $x = 4; $x <= 8; $x++) {
        for ($starty = $y = 7; $x <= 12; $y++) {
            $newLoL[$x − $startx][$y − $starty] = $LoL[$x][$y];
        }
    }
```

We can reduce some of the looping through slices

```
    for ($x = 4; $x <= 8; $x++) {
        push @newLoL, [ @{ $LoL[$x] } [ 7..12 ] ];
    }
```

If you were into Schwartzian Transforms, you would probably have selected map for that

```
    @newLoL = map { [ @{ $LoL[$_] } [ 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:

```
    @newLoL = splice_2D( \@LoL, 4 => 8, 7 => 12 );
    sub splice_2D {
        my $lrr = shift;        # ref to list of list refs!
        my ($x_lo, $x_hi,
            $y_lo, $y_hi) = @_;

        return map {
            [ @{ $lrr->[$_] } [ $y_lo .. $y_hi ] ]
        } $x_lo .. $x_hi;
    }
```

SEE ALSO

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

AUTHOR

Tom Christiansen <***tchrist@perl.com***

Last udpate: Sat Oct 7 19:35:26 MDT 1995

NAME

perlobj − Perl objects

DESCRIPTION

First of all, 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*.

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.

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 {} }
```

The `{}` constructs a reference to an anonymous hash containing no key/value pairs. 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 its reference to it could have been returned directly, like this:

```
sub new {
    my $self = {};
    bless $self;
    return $self;
}
```

In fact, 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 perlmod*), 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 `$obj−>new()`, then use something like this. The `initialize()` method used will be of whatever `$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.

A constructor may re−bless a referenced object currently belonging to another class, but then 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.)

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.

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.

If a missing method is found in one of the base classes, 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 a method isn't found, but an AUTOLOAD routine is found, then that is called on behalf of the missing method.

If neither a method nor an AUTOLOAD routine is found in @ISA, then one last try is made for the method (or an AUTOLOAD routine) in a class called UNIVERSAL. (Several commonly used methods are automatically supplied in the UNIVERSAL class; see *"Default UNIVERSAL methods"* for more details.) If that doesn't work, Perl finally gives up and complains.

Perl classes do only method inheritance. 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.

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 or package it is being invoked on. There are just two types of methods, which we'll call class and instance. (Sometimes you'll hear these called static and virtual, in honor of the two C++ method types they most closely resemble.)

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 typically class methods. 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";
        }
    }
```

Method Invocation

There are two ways to invoke a method, one of which you're already familiar with, and the other of which will look familiar. Perl 4 already had an "indirect object" syntax that you use when you say

```
    print STDERR "help!!!\n";
```

This same syntax can be used to call either class or instance methods. We'll use the two methods defined above, the class method to lookup an object reference and the instance method to print out its attributes.

```
    $fred = find Critter "Fred";
    display $fred 'Height', 'Weight';
```

These could be combined into one statement by using a BLOCK in the indirect object slot:

```
    display {find Critter "Fred"} 'Height', 'Weight';
```

For C++ fans, there's also a syntax using −> notation that does exactly the same thing. The parentheses are required if there are any arguments.

```
    $fred = Critter->find("Fred");
    $fred->display('Height', 'Weight');
```

or in one statement,

```
    Critter->find("Fred")->display('Height', 'Weight');
```

There are times when one syntax is more readable, and times when the other syntax is more readable. The indirect object syntax is less cluttered, but it has the same ambiguity as ordinary list operators. Indirect object method calls are parsed using the same rule as list operators: "If it looks like a function, it is a function". (Presuming for the moment that you think two words in a row can look like a function name. C++ programmers seem to think so with some regularity, especially when the first word is "new".) Thus, the parentheses of

```
    new Critter ('Barney', 1.5, 70)
```

are assumed to surround ALL the arguments of the method call, regardless of what comes after. Saying

```
    new Critter ('Bam' x 2), 1.4, 45
```

would be equivalent to

```
    Critter−>new('Bam' x 2), 1.4, 45
```

which is unlikely to do what you want.

There are times when you wish to specify which class‘s method to use. In this case, you can call your method as an ordinary subroutine call, being sure to pass the requisite first argument explicitly:

```
    $fred =  MyCritter::find("Critter", "Fred");
    MyCritter::display($fred, 'Height', 'Weight');
```

Note however, that this does not do any inheritance. If you wish merely to specify that Perl should *START* looking for a method in a particular package, use an ordinary method call, but qualify the method name with the package like this:

```
    $fred = Critter−>MyCritter::find("Fred");
    $fred−>MyCritter::display('Height', 'Weight');
```

If you‘re trying to control where the method search begins *and* you‘re executing in the class itself, then you may use the SUPER pseudo class, which says to start looking in your base class‘s @ISA list without having to name it explicitly:

```
    $self−>SUPER::display('Height', 'Weight');
```

Please note that the `SUPER::` construct is meaningful *only* within the class.

Sometimes you want to call a method when you don‘t know the method name ahead of time. You can use the arrow form, replacing the method name with a simple scalar variable containing the method name:

```
    $method = $fast ? "findfirst" : "findbest";
    $fred−>$method(@args);
```

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`

`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 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` 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.

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 doesn‘t do nested destruction for you. If your constructor re−blessed a reference from one of your base classes, your DESTROY may need to call DESTROY for any base classes that need it. But this applies to only re−blessed objects—an object reference that is merely *CONTAINED* in the current object will be freed and destroyed automatically when the current object is freed.

WARNING

An 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. The left side of −> is not so limited, because it‘s an infix operator, not a postfix operator.

That means that below, A and B are equivalent to each other, and C and D are equivalent, but AB and CD are different:

```
A: method $obref−>{"fieldname"}
B: (method $obref)−>{"fieldname"}
C: $obref−>{"fieldname"}−>method()
D: method {$obref−>{"fieldname"}}
```

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.

Two−Phased Garbage Collection

For most purposes, Perl uses a fast and simple reference−based garbage collection system. For this reason, 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 $self = shift;
    my $class = ref($self) || $self;
    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 multi−threadable 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 ***/tmp/test***, the following output is produced:

```
starting program at /tmp/test line 18.
CREATING SCALAR(0x8e5b8) at /tmp/test line 7.
CREATING SCALAR(0x8e57c) at /tmp/test line 7.
leaving block at /tmp/test line 23.
DESTROYING Subtle=SCALAR(0x8e5b8) at /tmp/test line 13.
just exited block at /tmp/test line 26.
time to die... at /tmp/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 and in fact are destructed in a separate pass before ordinary refs just to try 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.

A more complete garbage collection strategy will be implemented at a future date.

SEE ALSO

A kinder, gentler tutorial on object−oriented programming in Perl can be found in *perltoot*. You should also check out *perlbot* for other object tricks, traps, and tips, as well as *perlmod* for some style guides on constructing both modules and classes.

NAME

perltie − how to hide an object class in a simple variable

SYNOPSIS

```
 tie VARIABLE, CLASSNAME, LIST

 $object = tied VARIABLE

 untie VARIABLE
```

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. All of 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.

Tying Scalars

A class implementing a tied scalar should define the following methods: TIESCALAR, FETCH, STORE, and possibly 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 <***Jarkko.Hietaniemi@hut.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.

```
    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: $!";
        }
        return $new_nicety;
    }
```

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.

Tying Arrays

A class implementing a tied ordinary array should define the following methods: TIEARRAY, FETCH, STORE, and perhaps DESTROY.

WARNING: Tied arrays are *incomplete*. They are also distinctly lacking something for the `$#ARRAY` access (which is hard, as it‘s an lvalue), as well as the other obvious array functions, like `push()`, `pop()`, `shift()`, `unshift()`, and `splice()`.

For this discussion, we‘ll implement an array whose indices are fixed at its creation. If you try to access anything beyond those bounds, you‘ll take an exception. (Well, if you access an individual element; an aggregate assignment would be missed.) For example:

```
    require Bounded_Array;
    tie @ary, 'Bounded_Array', 2;
    $| = 1;
    for $i (0 .. 10) {
        print "setting index $i: ";
        $ary[$i] = 10 * $i;
        $ary[$i] = 10 * $i;
        print "value of elt $i now $ary[$i]\n";
    }
```

The preamble code for the class is as follows:

```
    package Bounded_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 `{BOUND}` field will store the maximum bound 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 $bound = shift;
    confess "usage: tie(\@ary, ’Bounded_Array’, max_subscript)"
        if @_ || $bound =~ /\D/;
    return bless {
        BOUND => $bound,
        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($self,$idx) = @_;
  if ($idx > $self->{BOUND}) {
    confess "Array OOB: $idx > $self->{BOUND}";
  }
  return $self->{ARRAY}[$idx];
}
```

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. For example:

```
sub STORE {
  my($self, $idx, $value) = @_;
  print "[STORE $value at $idx]\n" if _debug;
  if ($idx > $self->{BOUND} ) {
    confess "Array OOB: $idx > $self->{BOUND}";
  }
  return $self->{ARRAY}[$idx] = $value;
}
```

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.

The code we presented at the top of the tied array class accesses many elements of the array, far more than we‘ve set the bounds to. Therefore, it will blow up once they try to access beyond the 2nd element of @ary, as the following output demonstrates:

```
    setting index 0: value of elt 0 now 0
    setting index 1: value of elt 1 now 10
    setting index 2: value of elt 2 now 20
    setting index 3: Array OOB: 3 > 2 at Bounded_Array.pm line 39
            Bounded_Array::FETCH called at testba line 12
```

Tying Hashes

As the first Perl data type to be tied (see `dbmopen()`), hashes have the most complete and useful `tie()` implementation. 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. 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::Hash 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} }
        }
```

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 array values 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);
```

Tying FileHandles

This is partially implemented now.

A class implementing a tied filehandle should define the following methods: TIEHANDLE, at least one of PRINT, READLINE, GETC, or READ, and possibly DESTROY.

It 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 }
```

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++; print join($,,map(uc($_),@_)),$\ }
```

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 via <HANDLE. The method should return undef when there is no more data.

```
    sub READLINE { $r = shift; "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 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>;
```

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 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? Well, the good old `−w` flag will spot any instances where you call `untie()` and there are still valid references to the tied object. If the second script above is 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;
```

SEE ALSO

See *DB_File* or *Config* for some interesting `tie()` implementations.

BUGS

Tied arrays are *incomplete*. They are also distinctly lacking something for the `$#ARRAY` access (which is hard, as it‘s an lvalue), as well as the other obvious array functions, like `push()`, `pop()`, `shift()`, `unshift()`, and `splice()`.

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 *perlmod* for source code to MLDBM.

AUTHOR

Tom Christiansen

TIEHANDLE by Sven Verdoolaege <***skimo@dns.ufsia.ac.be***

NAME

perlbot − Bag‘o Object Tricks (the BOT)

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.

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

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 *"DELEGATION"*.

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*.

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";
```

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";
```

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";
```

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";
```

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;
```

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";
```

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;
```

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;
```

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;
```

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";
```

NAME

perltoot − Tom‘s object−oriented tutorial for perl

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.

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 MacOS 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.

Object Representation

By far the most common mechanism used in Perl to represent a Pascal record, a C struct, or a C++ class 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.

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.

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.

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.

While we‘re at it, let‘s make our constructor a bit more flexible. Rather than being uniquely a class method, we‘ll set it up so that it can be called as either a class method *or* an object method. That way you can say:

```
$me  = Person−>new();
$him = $me−>new();
```

To do this, all we have to do is check whether what was passed in was a reference or not. If so, we were invoked as an object method, and we need to extract the package (class) using the `ref()` function. If not, we just use the string passed in as the package name for blessing our referent.

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    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.

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.

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.)

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 .) 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 trade−offs 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.

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 $proto = shift;
    my $class = ref($proto) || $proto;
    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.

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 $proto = shift;
        my $class = ref($proto) || $proto;
        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"} };
    }
```

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.`

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).

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.

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 $proto = shift;
        my $class = ref($proto) || $proto;
        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 $proto = shift;
        my $class = ref($proto) || $proto;
        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;
    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();
```

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};
    }
```

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.

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, $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 $proto = shift;
        my $class = ref($proto) || $proto;
        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.

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.

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 built−in 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:

```
    use vars qw($VERSION);
    $VERSION = ’1.1’;
```

and then in Employee.pm could you can say

```
    use Employee 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.

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.

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.

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 $that  = shift;
     my $class = ref($that) || $that;
     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.)

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.

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;
    use vars qw($AUTOLOAD);  # it’s a package global

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

    sub new {
        my $that  = shift;
        my $class = ref($that) || $that;
        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.

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;
use vars qw(@ISA);
@ISA = qw(Person);

my %fields = (
    id          => undef,
    salary      => undef,
);

sub new {
    my $that  = shift;
    my $class = ref($that) || $that;
    my $self = bless $that->SUPER::new(), $class;
    my($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.

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::Template and Alias. These and other related metaclasses can be found in the modules directory on CPAN.

Class::Template

One of the older ones is Class::Template. 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::Template::struct()` function alleviates this tedium.

Here‘s a simple example of using it:

```
use Class::Template qw(struct);
use Jobbie;  # user−defined; see below

struct 'Fred' => {
    one        => '$',
    many       => '@',
    profession => Jobbie,  # calls Jobbie->new()
};

$ob = Fred->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.

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::Template 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::Template 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   ();
        use vars       qw(@ISA @EXPORT @EXPORT_OK %EXPORT_TAGS);
```

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

        use Class::Template 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::Template 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, and short−cut function call via `&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 built−in function. They also have the type names that a C programmer would most expect.

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 $that  = shift;
         my $class = ref($that) || $that;
         my $self = {
            NAME  => undef,
            AGE   => undef,
            PEERS => [],
        };
        bless($self, $class);
        return $self;
    }

    use Alias qw(attr);
    use vars qw($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 `use vars` 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 pre−declare 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::Template or Class::MethodMaker.

NOTES

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 built−in 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.

SEE ALSO

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

COPYRIGHT

I *really* hate to have to say this, but recent unpleasant experiences have mandated its inclusion:

```
    Copyright 1996 Tom Christiansen.  All Rights Reserved.
```

This work derives in part from the second edition of *Programming Perl*. Although destined for release as a man page with the standard Perl distribution, it is not public domain (nor is any of Perl and its docset: publishers beware). It‘s expected to someday make its way into a revision of the Camel Book. While it is copyright by me with all rights reserved, permission is granted to freely distribute verbatim copies of this document provided that no modifications outside of formatting be made, and that this notice remain intact. You are permitted and encouraged to use its code and derivatives thereof in your own source code for fun or for profit as you see fit. But so help me, if in six months I find some book out there with a hacked−up version of this material in it claiming to be written by someone else, I‘ll tell all the world that you‘re a jerk. Furthermore, your lawyer will meet my lawyer (or O‘Reilly‘s) over lunch to arrange for you to receive your just deserts. Count on it.

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.

NAME

perlipc − Perl interprocess communication (signals, fifos, pipes, safe subprocesses, sockets, and semaphores)

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.

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. 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, so calling any `print()` functions (or even anything that needs to malloc(3) more memory) could in theory trigger a memory fault and subsequent core dump.

```
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
```

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. 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 send a hang−up signal to all processes in the current process group *except for* the current process 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 another 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";
    }
```

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 re−install 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 even the more elaborate:

```
    use POSIX ":sys_wait_h";
    sub REAPER {
        my $child;
        while ($child = waitpid(−1,WNOHANG)) {
            $Kid_Status{$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 }
```

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.

Named Pipes

A named pipe (often referred to as a FIFO) is an old Unix IPC mechanism for processes communicating on the same machine. It works just like 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
    }
```

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 back−ticks 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");
    print FH "bang\n";
    close FH;
```

Filehandles

Both the main process and the child process share the same STDIN, STDOUT and STDERR filehandles. If both processes try to access them at once, strange things can happen. You may 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 complete dissociate the child process from the parent. 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 the TIOCNOTTY ioctl on it. See *tty(4)* for details.
- Change directory to /
- Reopen STDIN, STDOUT, and STDERR so they‘re not connected to the old tty.
- Background yourself like this:

```
    fork && exit;
```

Safe Pipe Opens

Another interesting approach to IPC is making your single program go multiprocess and communicate between (or even amongst) yourselves. The `open()` function will accept a file argument of either `"−|"` or `"|−"` to do a very interesting thing: it forks a child connected to the filehandle you‘ve opened. The child is running the same program as the parent. This is useful for safely opening a file when running under an assumed UID or GID, for example. If you open a pipe *to* minus, you can write to the filehandle you opened and your kid will find it in his STDIN. If you open a pipe *from* minus, you can read from the filehandle you opened whatever your kid writes to his STDOUT.

```
use English;
my $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 back−ticks 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 back−tick 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{ALRM} = 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
    }
```

Note that these operations are full Unix forks, which means they may not be correctly implemented on alien systems. Additionally, these are not true multi−threading. If you‘d like to learn more about threading, see the ***modules*** file mentioned below in the SEE ALSO section.

Bidirectional Communication

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 **−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 tri−directional 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()` 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" );
    Writer->autoflush(); # default here, actually
    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 non−standard ***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.

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 multiline messages and responses that end with a period on an empty line ("\n.\n" terminates a message/response).

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
    require 5.002;
    use strict;
    use Socket;
    my ($remote,$port, $iaddr, $paddr, $proto, $line);

    $remote  = shift || 'localhost';
    $port    = shift || 2345;  # random port
    if ($port =~ /\D/) { $port = getservbyname($port, 'tcp') }
    die "No port" unless $port;
    $iaddr   = inet_aton($remote)       || die "no host: $remote";
    $paddr   = sockaddr_in($port, $iaddr);

    $proto   = getprotobyname('tcp');
    socket(SOCK, PF_INET, SOCK_STREAM, $proto)  || die "socket: $!";
    connect(SOCK, $paddr)    || die "connect: $!";
    while ($line = <SOCK>) {
        print $line;
    }

    close (SOCK)            || die "close: $!";
    exit;
```

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 multi−homed 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
    require 5.002;
    use strict;
    BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
    use Socket;
    use Carp;

    sub logmsg { print "$0 $$: @_ at ", scalar localtime, "\n" }

    my $port = shift || 2345;
    my $proto = getprotobyname('tcp');
    $port = $1 if $port =~ /(\d+)/; # untaint port number

    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, "\n";
    }
```

And here's a multi−threaded version. It's multi−threaded 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
    require 5.002;
```

```
use strict;
BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
use Socket;
use Carp;

sub spawn;  # forward declaration
sub logmsg { print "$0 $$: @_ at ", scalar localtime, "\n" }

my $port = shift || 2345;
my $proto = getprotobyname('tcp');
$port = $1 if $port =~ /(\d+)/; # untaint port number

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;

sub REAPER {
    $waitedpid = wait;
    $SIG{CHLD} = \&REAPER;  # loathe sysV
    logmsg "reaped $waitedpid" . ($? ? " with exit $?" : '');
}

$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 {
        print "Hello there, $name, it's now ", scalar localtime, "\n";
        exec '/usr/games/fortune'
            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
    require 5.002;
    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 print system";
    }
```

Here‘s a sample Unix−domain client:

```
    #!/usr/bin/perl −w
    require 5.002;
    use Socket;
    use strict;
    my ($rendezvous, $line);

    $rendezvous = shift || ’/tmp/catsock’;
    socket(SOCK, PF_UNIX, SOCK_STREAM, 0)       || die "socket: $!";
    connect(SOCK, sockaddr_un($rendezvous))     || die "connect: $!";
    while ($line = <SOCK>) {
        print $line;
    }
    exit;
```

And here‘s a corresponding server.

```
    #!/usr/bin/perl −Tw
    require 5.002;
    use strict;
    use Socket;
    use Carp;

    BEGIN { $ENV{PATH} = ’/usr/ucb:/bin’ }

    my $NAME = ’/tmp/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";

    $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: $!";
        };
    }
```

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.

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.

Here‘s a UDP program similar to the sample Internet TCP client given above. 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;
    require 5.002;
    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--;
    }
```

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.

```
    $IPC_PRIVATE = 0;
    $IPC_RMID = 0;
    $size = 2000;
    $key = shmget($IPC_PRIVATE, $size , 0777 );
    die unless defined $key;

    $message = "Message #1";
    shmwrite($key, $message, 0, 60 ) || die "$!";
    shmread($key,$buff,0,60) || die "$!";

    print $buff,"\n";

    print "deleting $key\n";
    shmctl($key ,$IPC_RMID, 0) || die "$!";
```

Here's an example of a semaphore:

```
    $IPC_KEY = 1234;
    $IPC_RMID = 0;
    $IPC_CREATE = 0001000;
    $key = semget($IPC_KEY, $nsems , 0666 | $IPC_CREATE );
    die if !defined($key);
    print "$key\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;
    $key = semget($IPC_KEY,  0 , 0 );
    die if !defined($key);

    $semnum = 0;
    $semflag = 0;

    # 'take' semaphore
    # wait for semaphore to be zero
    $semop = 0;
    $opstring1 = pack("sss", $semnum, $semop, $semflag);

    # Increment the semaphore count
    $semop = 1;
    $opstring2 = pack("sss", $semnum, $semop,  $semflag);
    $opstring = $opstring1 . $opstring2;

    semop($key,$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;
    $key = semget($IPC_KEY, 0, 0);
    die if !defined($key);

    $semnum = 0;
    $semflag = 0;

    # Decrement the semaphore count
    $semop = -1;
    $opstring = pack("sss", $semnum, $semop, $semflag);

    semop($key,$opstring) || die "$!";
```

WARNING

The SysV IPC code above was written long ago, and it's definitely clunky looking. It should at the very least be made to `use strict` and `require "sys/ipc.ph"`. Better yet, perhaps someone should create an `IPC::SysV` module the way we have the `Socket` module for normal client−server communications.

(... time passes)

Voila! Check out the IPC::SysV modules written by Jack Shirazi. You can find them at a CPAN store near you.

NOTES

If you are running under version 5.000 (dubious) or 5.001, you can still use most of the examples in this document. You may have to remove the `use strict` and some of the `my()` statements for 5.000, and for both you'll have to load in version 1.2 or older of the ***Socket.pm*** module, which is included in *perl5.002*.

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 that you should check the return values of these functions. Always begin your socket programs this way for optimal success, and don't forget to add **−T** taint checking flag to the pound−bang line for servers:

```
    #!/usr/bin/perl -w
    require 5.002;
    use strict;
    use sigtrap;
    use Socket;
```

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.

Because few vendors provide C libraries that are safely re−entrant, the prudent programmer will do little else within a handler beyond `die()` to raise an exception and longjmp(3) out.

AUTHOR

Tom Christiansen, with occasional vestiges of Larry Wall‘s original version.

SEE ALSO

Besides the obvious functions in *perlfunc*, you should also check out the ***modules*** file at your nearest CPAN site. (See *perlmod* 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.

NAME

perldebug − Perl debugging

DESCRIPTION

First of all, have you tried using the **−w** switch?

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 as 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 pre−loads a Perl library file containing the debugger itself.

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 its own state information.)

Leading white space before a command would cause the debugger to think it‘s *NOT* a debugger command but for Perl, so be careful not to do that.

Debugger Commands

The debugger understands the following commands:

h [command]
: Prints out a help message.

  If you supply another debugger command as an argument to the `h` command, it prints out the description for just that command. The special argument of `h h` produces a more compact help listing, designed to fit together on one screen.

  If the output the `h` command (or any command, for that matter) scrolls past your screen, either precede the command with a leading pipe symbol so it‘s run through your pager, as in

```
    DB> |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 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 `print` function.

  The details of printout are governed by multiple `O`ptions.

V [pkg [vars]]	Display all (or some) variables in package (defaulting to the `main` package) 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 regexps.

Nested data structures are printed out in a legible fashion, unlike the `print` function.

The details of printout are governed by multiple `O`ptions.

X [vars]	Same as `V currentpackage [vars]`.
T	Produce a stack backtrace. See below for details on its output.
s [expr]	Single step. Executes until it reaches 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 it reaches 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.
<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.
−	List previous window of lines.
w [line]	List window (a few lines) around the current line.
.	Return debugger pointer to the last−executed line and print it out.
f filename	Switch to viewing a different file or eval statement. If `filename` is not a full filename as found in values of %INC, it is considered as a regexp.
/pattern/	Search forwards for pattern; final / is optional.
?pattern?	Search backwards for pattern; final ? is optional.
L	List all breakpoints and actions.
S [[!]pattern]	List subroutine names [not] matching pattern.
t	Toggle trace mode (see also `AutoTrace` `O`ption).
t expr	Trace through execution of expr. For example:

```
 $ 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
```

or, with the `O`ption `frame=2` 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
```

b [line] [condition]

Set a breakpoint. If line is omitted, sets a breakpoint on the line that is about to be executed. If a condition is specified, it‘s evaluated each time the statement is reached and a breakpoint is taken only if the condition is true. Breakpoints may be set on only 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 at the first line of the named subroutine.

b postpone subname [condition]

Set breakpoint at first line of subroutine after it is compiled.

b load filename

Set breakpoint at the first executed line of the file. Filename should be a full name as found in values of %INC.

b compile subname

Sets breakpoint at the first statement executed after the subroutine is compiled.

d [line]

Delete a breakpoint at the specified line. If line is omitted, deletes the breakpoint on the line that is about to be executed.

D

Delete all installed breakpoints.

a [line] command

Set an action to be done before the line is 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
Delete all installed actions.

O [opt[=val]] [opt"val"] [opt?]...
Set or query values of options. val defaults to 1. opt can be abbreviated. Several options can be listed.

`recallCommand`, `ShellBang`
The characters used to recall command or spawn shell. By default, these are both set to `!`.

`pager`
Program to use for output of pager−piped commands (those beginning with a `|` character.) By default, `$ENV{PAGER}` will be used.

`tkRunning`
Run Tk while prompting (with ReadLine).

`signalLevel`, `warnLevel`, `dieLevel`
Level of verbosity. By default the debugger is in a sane verbose mode, thus it will print backtraces on all the warnings and die−messages which are going to be printed out, and will print a message when interesting uncaught signals arrive.

To disable this behaviour, set these values to 0. If `dieLevel` is 2, then the messages which will be caught by surrounding `eval` are also printed.

`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, "emacs like" message is used.

`inhibit_exit`
If 0, allows *stepping off* the end of the script.

`PrintRet`
affects printing of return value after `r` command.

`frame`
affects printing messages on entry and exit from subroutines. If `frame & 2` is false, messages are printed on entry only. (Printing on exit may be useful if inter(di)spersed with other messages.)

If `frame & 4`, arguments to functions are printed as well as the context and caller info. If `frame & 8`, overloaded `stringify` and `tied` `FETCH` are enabled on the printed arguments. The length at which the argument list is truncated is governed by the next option:

`maxTraceLen`
length at which the argument list is truncated when `frame` option‘s bit 4 is set.

The following options affect what happens with `V`, `X`, and `x` commands:

`arrayDepth`, `hashDepth`
Print only first N elements (‘’ for all).

`compactDump`, `veryCompact`
Change style of array and hash dump. 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.

`quote`, `HighBit`, `undefPrint`
Change style of string dump. Default value of `quote` is `auto`, one can enable either double−quotish dump, or single−quotish by setting it to `"` or `'`. By default, characters with high bit set are printed *as is*.

`UsageOnly` *very* rudimentally per−package memory usage dump. Calculates total size of strings in variables in the package.

During startup options are initialized from `$ENV{PERLDB_OPTS}.` You can put additional initialization options `TTY`, `noTTY`, `ReadLine`, and `NonStop` there.

Example rc file:

```
&parse_options("NonStop=1 LineInfo=db.out AutoTrace");
```

The script will run without human intervention, putting trace information into the file *db.out*. (If you interrupt it, you would better reset `LineInfo` to something "interactive"!)

`TTY` The TTY to use for debugging I/O.

`noTTY` If set, goes in `NonStop` mode, and would not connect to a TTY. If interrupt (or if control goes to debugger via explicit setting of `$DB::signal` or `$DB::single` from the Perl script), connects to a TTY specified by the `TTY` option at startup, or to a TTY found at runtime using `Term::Rendezvous` module of your choice.

This module should implement a method `new` which returns an object with two methods: `IN` and `OUT`, returning two filehandles to use for debugging input and output correspondingly. Method `new` may inspect an argument which is a value of `$ENV{PERLDB_NOTTY}` at startup, or is `"/tmp/perldbtty$$"` otherwise.

`ReadLine` If false, readline support in debugger is disabled, so you can debug ReadLine applications.

`NonStop` If set, 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="N f=2" perl −d myprogram
```

will run the script `myprogram` without human intervention, printing out the call tree with entry and exit points. Note that `N f=2` is equivalent to `NonStop=1 frame=2`. Note also that at the moment when this documentation was written all the options to the debugger could be uniquely abbreviated by the first letter (with exception of `Dump*` options).

Other examples may include

```
$ PERLDB_OPTS="N f A L=listing" perl -d myprogram
```

− runs script non−interactively, printing info on each entry into a subroutine and each executed line into the file ***listing***. (If you interrupt it, you would better reset `LineInfo` to something "interactive"!)

```
$ env "PERLDB_OPTS=R=0 TTY=/dev/ttyc" perl -d myprogram
```

may be useful for debugging a program which uses `Term::ReadLine` itself. Do not forget detach shell from the TTY in the window which corresponds to ***/dev/ttyc***, say, by issuing a command like

```
$ sleep 1000000
```

See *"Debugger Internals"* below for more details.

< [command]
: Set an action (Perl command) to happen before every debugger prompt. A multi−line command may be entered by backslashing the newlines. If `command` is missing, resets the list of actions.

<< command
: Add an action (Perl command) to happen before every debugger prompt. A multi−line command may be entered by backslashing the newlines.

> 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. If `command` is missing, resets the list of 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.

{ [command]
: Set an action (debugger command) to happen before every debugger prompt. A multi−line command may be entered by backslashing the newlines. If `command` is missing, resets the list of actions.

{{ command
: Add an action (debugger command) to happen before every debugger prompt. A multi−line command may be entered by backslashing the newlines.

! number
: Redo a previous command (default previous command).

! −number
: Redo number‘th−to−last 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` too.

H −number
: Display last n commands. Only commands longer than one character are listed. If number is omitted, lists them all.

q or ^D
: Quit. ("quit" doesn‘t work for this.) This is the only supported way to exit the debugger, though typing `exit` twice may do it too.

  Set an `O`ption `inhibit_exit` to 0 if you want to be able to *step off* the end the script. You may also need to set `$finished` to 0 at some moment if you want to step through global destruction.

R
: Restart the debugger by **exec**ing a new session. It tries to maintain your history across this, but internal settings and command line options may be lost.

  Currently the following setting are preserved: history, breakpoints, actions, debugger `O`ptions, and the following command−line options: **−w**, **−I**, and **−e**.

|dbcmd — Run debugger command, piping DB::OUT to current pager.

||dbcmd — Same as `|dbcmd` but DB::OUT is temporarily **select**ed as well. Often used with commands that would otherwise produce long output, such as

```
    |V main
```

= [alias value] — Define a command alias, like

```
    = quit q
```

or list current aliases.

command — Execute command as a Perl statement. A missing semicolon will be supplied.

m expr — The expression is evaluated, and the methods which may be applied to the result are listed.

m package — The methods which may be applied to objects in the `package` are listed.

Debugger input/output

Prompt — The debugger prompt is something like

```
    DB<8>
```

or even

```
    DB<<17>>
```

where that number is the command number, which you'd use to access with the built−in **csh**−like history mechanism, e.g., `!17` would repeat command number 17. The number of angle brackets indicates the 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 out the result of a function call that itself also has a breakpoint, or you step into an expression via `s/n/t expression` command.

Multi−line commands

If you want to enter a multi−line command, such as a subroutine definition with several statements, or a format, you may 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 tells whether the function was called in a scalar or list context (we bet you can tell which is which). What that says is that you were in the function `main::infested` when you ran the stack dump, and that it was called in a 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 a list context from the *camel_flea* file with four arguments. The last stack frame shows that `main::pests` was called in a scalar context, also from *camel_flea*, but from line 4.

Note that if you execute `T` command from inside an active `use` statement, the backtrace will contain both *`require`* frame and an *`eval EXPR`*) frame.

Listing
: Listing given via different flavors of `l` command looks like this:

```
  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);
```

Note that the breakable lines are marked with `:`, lines with breakpoints are marked by `b`, with actions by `a`, and the next executed line is marked by `==>`.

Frame listing
: When `frame` option is set, debugger would print entered (and optionally exited) subroutines in different styles.

What follows is the start of the listing of

```
env "PERLDB_OPTS=f=1 N" perl -d -V
```

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
```

4

```
in  $=main::BEGIN() from /dev/nul: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/
  in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') f
in  @=Config::myconfig() from /dev/nul: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), 'PATCHLEVEL') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), '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
```

6

```
in  $=main::BEGIN() from /dev/nul: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/
  in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') f
  out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') f
 out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/
out $=main::BEGIN() from /dev/nul:0
in  @=Config::myconfig() from /dev/nul: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), 'PATCHLEVEL') from lib/Config.pm:574
 out $=Config::FETCH(ref(Config), 'PATCHLEVEL') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'SUBVERSION') from lib/Config.pm:574
```

14

```
in  $=main::BEGIN() from /dev/nul: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/
  in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') f
  out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') f
 out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/
out $=main::BEGIN() from /dev/nul:0
in  @=Config::myconfig() from /dev/nul:0
 in  $=Config::FETCH('Config=HASH(0x1aa444)', 'package') from lib/Confi
 out $=Config::FETCH('Config=HASH(0x1aa444)', 'package') from lib/Confi
 in  $=Config::FETCH('Config=HASH(0x1aa444)', 'baserev') from lib/Confi
 out $=Config::FETCH('Config=HASH(0x1aa444)', 'baserev') from lib/Confi
```

In all the cases indentation of lines shows the call tree, if bit 2 of `frame` is set, then a line is printed on exit from a subroutine as well, if bit 4 is set, then the arguments are printed as well as the caller info, if bit 8 is set, the arguments are printed even if they are tied or references.

When a package is compiled, a line like this

```
    Package lib/Carp.pm.
```

is printed with proper indentation.

Debugging compile−time statements

If you have any compile−time executable statements (code within a BEGIN block or a `use` statement), these will `NOT` be stopped by debugger, although `require`s 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 `$DB::single` to the value 2, it's equivalent to having just typed the `n` command, whereas a value of 1 means the `s` command. The `$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 debugger, set a breakpoint on *load* of some module thusly

```
    DB<7> b load f:/perllib/lib/Carp.pm
  Will stop on load of `f:/perllib/lib/Carp.pm'.
```

and restart debugger by `R` command (if possible). One can use `b compile subname` for the same purpose.

Debugger Customization

Most probably you not want to modify the debugger, it contains enough hooks to satisfy most needs. You may change the behaviour of debugger from the debugger itself, using `O`ptions, from the command line via `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\$/';
```

One changes options from ***.perldb*** file via 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`, it is called after all the debugger initialization ends. ***.perldb*** may be contained in the current directory, or in the `LOGDIR`/`HOME` directory.

If you want to modify the debugger, copy ***perl5db.pl*** from the Perl library to another name and modify it as necessary. You'll also want to set your `PERL5DB` environment variable to say something like this:

```
    BEGIN { require "myperl5db.pl" }
```

As the last resort, one can use `PERL5DB` to customize debugger by directly setting internal variables or calling debugger functions.

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.

A rudimentary command−line completion is also available. Unfortunately, the names of lexical variables are not available for completion.

Editor Support for Debugging

If you have GNU **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 is also delivered 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.

(Historically, a similar setup for interacting with **vi** and the X11 window system had also been available, but at the time of this writing, no debugger support for **vi** currently exists.)

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. One of the most popular alternative debuggers for Perl is **DProf**, the Perl profiler. As of this writing, **DProf** is not included with the standard Perl distribution, but it is expected to be included soon, for certain values of "soon".

Meanwhile, you can fetch the Devel::Dprof module from CPAN. Assuming it's properly installed on your system, 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 Devel::DProf package) can be used to interpret the information which is in that profile.

Debugger support in perl

When you call the **caller** function (see *caller*) from the package DB, Perl sets the array @DB::args to contain the arguments the corresponding stack frame was called with.

If perl is run with **−d** option, the following additional features are enabled:

- Perl inserts the contents of `$ENV{PERL5DB}` (or `BEGIN {require 'perl5db.pl'}` if not present) before the first line of the application.
- The array `@{"_<$filename"}` is the line−by−line contents of `$filename` for all the compiled files. Same for `eval`ed strings which contain subroutines, or which are currently executed. The `$filename` for `eval`ed strings looks like `(eval 34)`.
- The hash `%{"_<$filename"}` contains breakpoints and action (it is keyed by line number), and individual entries are settable (as opposed to the whole hash). Only true/false is important to Perl, though the values used by ***perl5db.pl*** have the form `"$break_condition\0$action"`. Values

are magical in numeric context: they are zeros if the line is not breakable.

Same for evaluated strings which contain subroutines, or which are currently executed. The `$filename` for `eval`ed strings looks like `(eval 34)`.

- The scalar `${"_<$filename"}` contains `"_<$filename".` Same for evaluated strings which contain subroutines, or which are currently executed. The `$filename` for `eval`ed strings looks like `(eval 34)`.
- After each `require`d file is compiled, but before it is executed, `DB::postponed(*{"_<$filename"})` is called (if subroutine `DB::postponed` exists). Here the `$filename` is the expanded name of the `require`d file (as found in values of `%INC`).
- After each subroutine `subname` is compiled existence of `$DB::postponed{subname}` is checked. If this key exists, `DB::postponed(subname)` is called (if subroutine `DB::postponed` exists).
- A hash `%DB::sub` is maintained, with keys being subroutine names, values having the form `filename:startline-endline`. `filename` has the form `(eval 31)` for subroutines defined inside `eval`s.
- When execution of the application reaches a place that can have a breakpoint, a call to `DB::DB()` is performed if any one of variables `$DB::trace`, `$DB::single`, or `$DB::signal` is true. (Note that these variables are not `local`izable.) This feature is disabled when the control is inside `DB::DB()` or functions called from it (unless `$^D & (1<<30)`).
- When execution of the application reaches a subroutine call, a call to `&DB::sub(`*args*`)` is performed instead, with `$DB::sub` being the name of the called subroutine. (Unless the subroutine is compiled in the package `DB`.)

Note that no subroutine call is possible until `&DB::sub` is defined (for subroutines outside of package `DB`). (This restriction is recently lifted.)

(In fact, for the standard debugger the same is true if `$DB::deep` (how many levels of recursion deep into the debugger you can go before a mandatory break) is not defined.)

With the recent updates the minimal possible debugger consists of one line

```
  sub DB::DB {}
```

which is quite handy as contents of `PERL5DB` environment variable:

```
  env "PERL5DB=sub DB::DB {}" perl -d your-script
```

Another (a little bit more useful) minimal debugger can be created with the only line being

```
  sub DB::DB {print ++$i; scalar <STDIN>}
```

This debugger would print the sequential number of encountered statement, and would wait for your `CR` to continue.

The following debugger is quite functional:

```
  {
    package DB;
    sub DB  {}
    sub sub {print ++$i, " $sub\n"; &$sub}
  }
```

It prints the sequential number of subroutine call and the name of the called subroutine. Note that `&DB::sub` should be compiled into the package `DB`.

Debugger Internals

At the start, the debugger reads your rc file (***./.perldb*** or ***~/.perldb*** under UNIX), which can set important options. This file may define a subroutine `&afterinit` to be executed after the debugger is initialized.

After the rc file is read, the debugger reads environment variable PERLDB_OPTS and parses it as a rest of `O ...` line in debugger prompt.

It also maintains magical internal variables, such as `@DB::dbline`, `%DB::dbline`, which are aliases for `@{"::_<current_file"} %{"::_<current_file"}`. Here `current_file` is the currently selected (with the debugger‘s `f` command, or by flow of execution) file.

Some functions are provided to simplify customization. See *"Debugger Customization"* for description of `DB::parse_options(string)`. The function `DB::dump_trace(skip[, count])` skips the specified number of frames, and returns an array containing info about the caller frames (all if `count` is missing). Each entry is a hash with keys `context` (`$` or `@`), `sub` (subroutine name, or info about eval), `args` (`undef` or a reference to an array), `file`, and `line`.

The function `DB::print_trace(FH, skip[, count[, short]])` prints formatted info about caller frames. The last two functions may be convenient as arguments to `<`, `<<` commands.

Other resources

You did try the **−w** switch, didn‘t you?

BUGS

You cannot get the stack frame information or otherwise debug functions that were not compiled by Perl, such as 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.

NAME

perldiag − various Perl diagnostics

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 (mandatory).
(F) A fatal error (trappable).
(P) An internal error you should never see (trappable).
(X) A very fatal error (non-trappable).
(A) An alien error message (not generated by Perl).
```

Optional warnings are enabled by using the **−w** switch. 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*. Trappable errors may be trapped using the eval operator. See *eval*.

Some of these messages are generic. Spots that vary are denoted with a %s, just as in a printf format. Note that some messages start with a %s! The symbols `"%−?@` sort before the letters, while `[` and `\` sort after.

"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.

"my" variable %s masks earlier declaration in same scope

(S) 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.

"no" not allowed in expression

(F) The "no" keyword is recognized and executed at compile time, and returns no useful value. See *perlmod*.

"use" not allowed in expression

(F) The "use" keyword is recognized and executed at compile time, and returns no useful value. See *perlmod*.

% may only be used in unpack

(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*.

%s (...) interpreted as function

(W) 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)*.

%s argument is not a HASH element

(F) The argument to `exists()` must be a hash element, such as

```
$foo{$bar}
$ref->[12]->{"susie"}
```

%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"}
```

%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*.

%s found where operator expected

(S) 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.

%s had compilation errors

(F) The final summary message when a `perl -c` fails.

%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.

%s matches null string many times

(W) The pattern you've specified would be an infinite loop if the regular expression engine didn't specifically check for that. See *perlre*.

%s never introduced

(S) The symbol in question was declared but somehow went out of scope before it could possibly have been used.

%s syntax OK

(F) The final summary message when a `perl -c` succeeds.

%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.

%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: 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.

%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.

−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.

`−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.

500 Server error

See Server error.

?+* follows nothing in regexp

(F) You started a regular expression with a quantifier. Backslash it if you meant it literally. See *perlre*.

@ outside of string

(F) You had a pack template that specified an absolute position outside the string being unpacked. See *pack*.

`accept()` on closed fd

(W) You tried to do an accept on a closed socket. Did you forget to check the return value of your `socket()` call? See *accept*.

Allocation too large: %lx

(X) You can‘t allocate more than 64K on an MSDOS machine.

Allocation too large

(F) You can‘t allocate more than 2^31+"small amount" bytes.

Applying %s to %s will act on scalar(%s)

(W) The pattern match (//), substitution (s///), and translation (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* and *map* for alternatives.

Arg too short for msgsnd

(F) `msgsnd()` requires a string at least as long as sizeof(long).

Ambiguous use of %s resolved as %s

(W)(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.

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`.

Argument "%s" isn‘t numeric%s

(W) 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.

Array @%s missing the @ in argument %d of `%s()`

(D) 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.

Attempt to free non−arena SV: 0x%lx

(P) 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 non−existent 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 free temp prematurely

(W) 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) The reference counts got screwed up on symbol aliases.

Attempt to free unreferenced scalar

(W) 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 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*.

Bad arg length for %s, is %d, should be %d

(F) You passed a buffer of the wrong size to one of `msgctl()`, `semctl()` or `shmctl()`. In C parlance, the correct sizes are, respectively, sizeof(struct msqid_ds *), sizeof(struct semid_ds *), and sizeof(struct shmid_ds *).

Bad 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) 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 1.

This message can be quite often seen 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 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 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.

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.

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.

`bind()` on closed fd

(W) You tried to do a bind on a closed socket. Did you forget to check the return value of your `socket()` call? See *bind*.

Bizarre copy of %s in %s

(P) Perl detected an attempt to copy an internal value that is not copiable.

Callback called exit

(F) A subroutine invoked from an external package via `perl_call_sv()` exited by calling exit.

Can‘t "goto" outside a 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*.

Can‘t "last" outside a 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()`. 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*.

Can‘t "next" outside a 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()`. 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*.

Can‘t "redo" outside a 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()`. 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*.

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 break at that line

(S) A warning intended for while running within the debugger, indicating the line number specified wasn‘t the location of a statement that could be stopped at.

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 unblessed reference

(F) A method call must know what package it‘s supposed to run in. 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 neither an object reference nor a package name. (Perhaps it‘s null?) Something like this will reproduce the error:

```
$BADREF = undef;
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 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 %s in my

(F) Only scalar, array, and hash variables may be declared as lexical variables. They must have ordinary identifiers as names.

Can‘t do inplace edit on %s: %s

(S) The creation of the new file failed for the indicated reason.

Can‘t do in−place edit without backup

(F) You‘re on a system such as MSDOS 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 > 14 characters

(S) There isn‘t enough room in the filename to make a backup name for the file.

Can‘t do inplace edit: %s is not a regular file

(S) 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 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 do {n,m} with n > m

(F) Minima must be less than or equal to maxima. If you really want your regexp to match something 0 times, just put {0}. See *perlre*.

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) An `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 script to execute could not be found in the PATH, or at least not with the correct permissions.

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*.

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.)
```

Can‘t fork

(F) A fatal error occurred while trying to fork while opening a pipeline.

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.

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 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*.

Can‘t localize a reference

(F) You said something like `local $$ref`, which is not allowed because the compiler can‘t determine whether `$ref` will end up pointing to anything with a symbol table entry, and a symbol table entry is necessary to do a local.

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 locate %s in @INC

(F) You said to do (or require, or use) a file that couldn‘t be found in any of the libraries mentioned in @INC. Perhaps you need to set the PERL5LIB 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*.

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) The @ISA array contained the name of another package that doesn‘t seem to exist.

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.

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 non−existent substring

(P) The internal routine that does assignment to a `substr()` was handed a NULL.

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 open %s: %s

(S) An in−place edit couldn‘t open the original file for the indicated reason. Usually this is because you don‘t have read permission for the file.

Can‘t open bidirectional pipe

(W) You tried to say `open(CMD, "|cmd|")`, which is not supported. You can try any of several modules in the Perl library to do this, such as IPC::Open2. Alternately, direct the pipe‘s output to a file using ">", and then read it in under a different file handle.

Can‘t open error file %s as stderr

(F) An error peculiar to VMS. Perl does its own command line redirection, and couldn‘t open the file specified after ‘2>’ or ‘2>>’ on the command line for writing.

Can‘t open input file %s as stdin

(F) An error peculiar to VMS. Perl does its own command line redirection, and couldn‘t open the file specified after ‘<’ on the command line for reading.

Can‘t open output file %s as stdout

(F) An error peculiar to VMS. Perl does its own command line redirection, and couldn‘t open the file specified after ‘>’ or ‘>>’ on the command line for writing.

Can‘t open output pipe (name: %s)

(P) An error peculiar to VMS. Perl does its own command line redirection, and couldn‘t open the pipe into which to send data destined for stdout.

Can‘t open perl script "%s": %s

(F) The script you specified can‘t be opened for the indicated reason.

Can‘t rename %s to %s: %s, skipping file

(S) 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 reswap uid and euid

(P) The `setreuid()` call failed for some reason in the setuid emulator of suidperl.

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 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) Logarithms are defined on only positive real 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 Complex package available for 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 "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 for loop variable

(F) Only a simple scalar variable may be used as a loop variable on a foreach.

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 \1 to mean $1 in expression

(W) 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 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 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 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 de−lurk some insidious errors.

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 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 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.

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.

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.

Cannot 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.

chmod: mode argument is missing initial 0

(W) A novice will sometimes say

```
    chmod 777, $filename
```

not realizing that 777 will be interpreted as a decimal number, equivalent to 01411. Octal constants are introduced with a leading 0 in Perl, as in C.

Close on unopened file <%s>

(W) You tried to close a filehandle that was never opened.

`connect()` on closed fd

(W) You tried to do a connect on a closed socket. Did you forget to check the return value of your `socket()` call? See *connect*.

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*.

Corrupt malloc ptr 0x%lx at 0x%lx

(P) The malloc package that comes with Perl had an internal failure.

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.

Deep recursion on subroutine "%s"

(W) This subroutine has called itself (directly or indirectly) 100 times than it has returned. This probably indicates an infinite recursion, unless you‘re writing strange benchmark programs, in which case it indicates something else.

Did you mean &%s instead?

(W) You probably referred to an imported subroutine `&FOO` as `$FOO` or some such.

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.

Do you need to pre−declare %s?

(S) 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.

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.

Duplicate `free()` ignored

(S) An internal routine called `free()` on something that had already been freed.

elseif should be elsif

(S) 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.

END failed—cleanup aborted

(F) An untrapped exception was raised while executing an END subroutine. The interpreter is immediately exited.

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.

Execution of %s aborted due to compilation errors

(F) The final summary message when a Perl compilation fails.

Exiting eval via %s

(W) You are exiting an eval by unconventional means, such as a goto, or a loop control statement.

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*.

Exiting subroutine via %s

(W) You are exiting a subroutine by unconventional means, such as a goto, or a loop control statement.

Exiting substitution via %s

(W) You are exiting a substitution by unconventional means, such as a return, a goto, or a loop control statement.

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 never opened

(W) An I/O operation was attempted on a filehandle that was never initialized. You need to do an `open()` or a `socket()` call, or call a constructor from the FileHandle package.

Filehandle %s opened for only input

(W) 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*.

Filehandle opened for only input

(W) 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*.

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.

Final @ should be \@ or @name

(F) You must now decide whether the final @ in a string was meant to be a literal "at" 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.

Format %s redefined

(W) You redefined a format. To suppress this warning, say

```
    {
        local $^W = 0;
        eval "format 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.

Found = in conditional, should be ==

(W) You said

```
    if ($foo = 123)
```

when you meant

```
    if ($foo == 123)
```

(or something like that).

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{sock,peer}name()` on closed fd

(W) 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.

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".

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"), or explicitly qualified to say which package the global variable is in (using "::").

goto must have label

(F) Unlike with "next" or "last", you‘re not allowed to goto an unspecified destination. See *goto*.

Had to create %s unexpectedly

(S) 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) Really old Perl let you omit the % on hash names in some spots. This is now heavily deprecated.

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 non−standard names, or it may indicate that a logical name table has been corrupted.

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 here documents (e.g., `print <<EOF;`).

Under UNIX, this error is usually caused by executing Perl code — either the main program, a module, or an eval'd string — that was transferred over a network connection from a non−UNIX system without properly converting the text file format.

Under systems that use something other than '\n' to delimit lines of text, this error can also be caused by reading Perl code from a file handle that is in binary mode (as set by the `binmode` operator).

In either case, the Perl code in question will probably need to be converted with something like `s/\x0D\x0A?/\n/g` before it can be executed.

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 modulus zero

(F) You tried to divide a number by 0 to get the remainder. Most numbers don't take to this kindly.

Illegal octal digit

(F) You used an 8 or 9 in a octal number.

Illegal octal digit ignored

(W) You may have tried to use an 8 or 9 in a octal number. Interpretation of the octal number stopped before the 8 or 9.

In string, @%s now must be written as \@%s

(F) It used to be that Perl would try to guess whether you wanted an array interpolated or a literal @. It did this when the string was first used at runtime. Now strings are parsed at compile time, and ambiguous instances of @ must be disambiguated, either by prepending a backslash to indicate a literal, or by declaring (or using) the array within the program before the string (lexically). (Someday it will simply assume that an unbackslashed @ interpolates an array.)

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 PATH

(F) You can't use `system()`, `exec()`, or a piped open in a setuid or setgid script if `$ENV{PATH}` is 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 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 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*). 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 disaster in regexp

(P) Something went badly wrong in the regular expression parser.

internal urp in regexp at /%s/

(P) Something went badly awry in the regular expression parser.

invalid [] range in regexp

(F) The range specified in a character class had a minimum character greater than the maximum character. See *perlre*.

ioctl is not implemented

(F) Your machine apparently doesn't implement `ioctl()`, which is pretty strange for a machine that supports C.

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*.

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*.

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*.

`listen()` on closed fd

(W) You tried to do a listen on a closed socket. Did you forget to check the return value of your `socket()` call? See *listen*.

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*.

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) An underline in a decimal constant wasn't on a 3−digit boundary.

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 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 operator before %s?

(S) 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 bracket

(F) The lexer counted more opening curly brackets (braces) 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) 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.

Modification of non−creatable array value attempted, subscript %d

(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, subscript "%s"

(F) 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".

msg%s not implemented

(F) You don‘t have System V message IPC on your system.

Multidimensional syntax %s not supported

(W) Multidimensional arrays aren‘t written like `$foo[1,2,3]`. They‘re written like `$foo[1][2][3]`, as in C.

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.

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.

nested *?+ in regexp

(F) You can‘t quantify a quantifier without intervening parentheses. So things like ** or +* or ?* are illegal.

Note, however, that the minimal matching quantifiers, `*?`, `+?`, and `??` appear to be nested quantifiers, but aren‘t. See *perlre*.

No #! line

(F) The setuid emulator requires that scripts have a well−formed #! line even on machines that don‘t support the #! construct.

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 **−e** allowed in setuid scripts

(F) A setuid script can‘t be specified by the user.

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* and *import*. 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 whither 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 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 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 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 whither 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 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 **−I**

(F) The argument to **−I** must follow the **−I** immediately with no intervening space.

No such pipe open

(P) An error peculiar to VMS. The internal routine `my_pclose()` tried to close a pipe which hadn't been opened. This should have been caught earlier as an attempt to close an unopened filehandle.

No such signal: SIG%s

(W) 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 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 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 enough arguments for %s

(F) The function requires more arguments than you specified.

Not enough format arguments

(W) A format specified more picture fields than the next line supplied. See *perlform*.

Null filename used

(F) You can't require the null filename, especially because on many machines that means the current directory! See *require*.

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 OP IN RUN

(P) Some internal routine called `run()` with a null opcode pointer.

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.

Odd number of elements in hash list

(S) You specified an odd number of elements to a hash list, which is odd, because hash lists 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.

oops: oopsAV

(S) An internal warning that the grammar is screwed up.

oops: oopsHV

(S) 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) 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'".

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.

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.

page overflow

(W) A single call to `write()` produced more lines than can fit on a page. See *perlform*.

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: die %s

(P) We popped the context stack to an eval context, and then discovered it wasn't an eval context.

panic: do_match

(P) The internal `pp_match()` routine was called with invalid operational data.

panic: do_split

(P) Something terrible went wrong in setting up for the split.

panic: do_subst

(P) The internal `pp_subst()` routine was called with invalid operational data.

panic: do_trans

(P) The internal `do_trans()` routine was called with invalid operational data.

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: 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: 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: 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: 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: yylex

(P) The lexer got into a bad state while processing a case modifier.

Pareneses missing around "%s" list

(W) You said something like

```
my $foo, $bar = @_;
```

when you meant

```
my ($foo, $bar) = @_;
```

Remember that "my" and "local" bind closer than comma.

Perl %3.3f 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*.

Permission denied

(F) The setuid emulator in suidperl decided you were up to no good.

pid %d not a child

(W) 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.

POSIX getpgrp can‘t take an argument

(F) Your C compiler uses POSIX `getpgrp()`, which takes no argument, unlike the BSD version, which takes a pid.

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 exclamation marks 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 !;
```

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*.

Precedence problem: open %s should be open(%s)

(S) 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 "||".

print on closed filehandle %s

(W) The filehandle you're printing on got itself closed sometime before now. Check your logic flow.

printf on closed filehandle %s

(W) The filehandle you're writing to got itself closed sometime before now. Check your logic flow.

Probable precedence problem on %s

(W) The compiler found a bare word 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;
```

Prototype mismatch: (%s) vs (%s)

(S) The subroutine being defined had a pre−declared (forward) declaration with a different function prototype.

Read on closed filehandle <%s>

(W) The filehandle you're reading from got itself closed sometime before now. Check your logic flow.

Reallocation too large: %lx

(F) You can't allocate more than 64K on an MSDOS machine.

Recompile perl with **−D**DEBUGGING to use **−D** switch

(F) 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

(F) More than 100 levels of inheritance were used. Probably indicates an unintended loop in your inheritance hierarchy.

Reference miscount in `sv_replace()`

(W) The internal `sv_replace()` function was handed a new SV with a reference count of other than 1.

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.

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*.

Reversed %s= operator

(W) 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*.

Scalar value @%s[%s] better written as `$%s[%s]`

(W) 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) 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 with its setuid or setgid bit not 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.

`seek()` on unopened file

(W) You tried to use the `seek()` function on a filehandle that was either never opened or has been closed since.

select not implemented

(F) This machine doesn't implement the `select()` system call.

sem%s not implemented

(F) You don't have System V semaphore IPC on your system.

semi−panic: attempt to dup freed string

(S) The internal `newSVsv()` routine was called to duplicate a scalar that had previously been marked as free.

Semicolon seems to be missing

(W) A nearby syntax error was probably caused by a missing semicolon, or possibly some other missing operator, such as a comma.

Send on closed socket

(W) The filehandle you‘re sending to got itself closed sometime before now. Check your logic flow.

Sequence (?#... not terminated

(F) A regular expression comment must be terminated by a closing parenthesis. Embedded parentheses aren‘t allowed. See *perlre*.

Sequence (?%s...) not implemented

(F) A proposed regular expression extension has the character reserved but has not yet been written. See *perlre*.

Sequence (?%s...) not recognized

(F) You used a regular expression extension that doesn‘t make sense. See *perlre*.

Server error

Also known as "500 Server error".

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.com/perl/faq/idiots−guide.html
    http://www.perl.com/perl/faq/perl−cgi−faq.html
    ftp://rtfm.mit.edu/pub/usenet/news.answers/www/cgi−faq
    http://hoohoo.ncsa.uiuc.edu/cgi/interface.html
    http://www−genome.wi.mit.edu/WWW/faqs/www−security−faq.html
```

`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.

`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 `$<lt`, and your operating system doesn‘t support the `setruid()` system call (or equivalent), or at least Configure didn‘t think so.

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.

`shutdown()` on closed fd

(W) You tried to do a shutdown on a closed socket. Seems a bit superfluous.

SIG%s handler "%s" not defined

(W) 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*.

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*.

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*.

Stat on unopened file <%s>

(W) You tried to use the `stat()` function (or an equivalent file test) on a filehandle that was either never opened or has been closed since.

Statement unlikely to be reached

(W) 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.

Stub found while resolving method ‘%s’ overloading ‘%s’ in package ‘%s’

(P) Overloading resolution over @ISA tree may be broken by importation stubs. Stubs should never be implicitely created, but explicit calls to `can` may break this.

Subroutine %s redefined

(W) You redefined a subroutine. To suppress this warning, say

```
{
    local $^W = 0;
    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 a s/// or s{}{} construct. Remember that bracketing delimiters count nesting level.

Substitution replacement not terminated

(F) The lexer couldn‘t find the final delimiter of a s/// or s{}{} construct. Remember that bracketing delimiters count nesting level.

substr outside of string

(W) 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*.

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.

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.

System V IPC is not implemented on this machine

(F) You tried to do something with a function beginning with "sem", "shm", or "msg". See *semctl*, for example.

Syswrite on closed filehandle

(W) The filehandle you‘re writing to got itself closed sometime before now. Check your logic flow.

`tell()` on unopened file

(W) You tried to use the `tell()` function on a filehandle that was either never opened or has been closed since.

Test on unopened file <%s>

(W) You tried to invoke a file test operator on a filehandle that isn‘t open. Check your logic. See also *−X*.

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 *$[*.

The %s function is unimplemented

The function indicated isn't implemented on this architecture, according to the probings of Configure.

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 stat preceding `-l _` 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.

times not implemented

(F) Your version of the C library apparently doesn't do `times()`. I suspect you're not running on Unix.

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 "**-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.

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 many ('s
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 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.

trailing \ in regexp

(F) The regular expression ends with an unbackslashed backslash. Backslash it. See *perlre*.

Translation pattern not terminated

(F) The lexer couldn't find the interior delimiter of a tr/// or tr[][] construct.

Translation replacement not terminated

(F) The lexer couldn't find the final delimiter of a tr/// or tr[][] construct.

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: argument is missing initial 0

(W) A umask of 222 is incorrect. It should be 0222, because octal literals always start with 0 in Perl, as in C.

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) The exit code detected an internal inconsistency in how many execution contexts were entered and left.

Unbalanced saves: %d more saves than restores

(W) The exit code detected an internal inconsistency in how many values were temporarily localized.

Unbalanced scopes: %d more ENTERs than LEAVEs

(W) The exit code detected an internal inconsistency in how many blocks were entered and left.

Unbalanced tmps: %d more allocs than frees

(W) 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*.

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*.

Undefined top format "%s" called

(F) The format indicated doesn‘t seem to exist. Perhaps it‘s really in another package? See *perlform*.

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.

Unknown BYTEORDER

(F) There are no byte−swapping functions for a machine with this byte order.

unmatched `()` in regexp

(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. See *perlre*.

Unmatched right bracket

(F) The lexer counted more closing curly brackets (braces) than opening ones, so you‘re probably missing an opening bracket. As a general rule, you‘ll find the missing one (so to speak) near the place you were last editing.

unmatched [] in regexp

(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. See *perlre*.

Unquoted string "%s" may clash with future reserved word

(W) You used a bare word 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 \%03o ignored

(S) A garbage character was found in the input, and ignored, in case it‘s a weird control character on an EBCDIC machine, or some such.

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

(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) A file operation was attempted on a filename, and that operation failed, PROBABLY because the filename contained a newline, PROBABLY because you forgot to `chop()` or `chomp()` it off. See *chop*.

Unsupported directory function "%s" called

(F) Your machine doesn‘t support `opendir()` and `readdir()`.

Unsupported function %s

(F) This machines doesn‘t implement the indicated function, apparently. At least, Configure doesn‘t think so.

Unsupported socket function "%s" called

(F) Your machine doesn‘t support the Berkeley socket mechanism, or at least that‘s what Configure thought.

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".

Use of $# is deprecated

(D) This was an ill−advised attempt to emulate a poorly defined **awk** feature. Use an explicit `printf()` or `sprintf()` instead.

Use of `$*` is deprecated

(D) 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 %s in printf format not supported

(F) You attempted to use a feature of printf that is accessible from only C. This usually means there‘s a better way to do it in Perl.

Use of %s is deprecated

(D) 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 bare << to mean <<"" is deprecated

(D) You are now encouraged to use the explicitly quoted form if you wish to use a blank line as the terminator of the here−document.

Use of implicit split to @_ is deprecated

(D) 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 uninitialized value

(W) 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 an initial value to your variables.

Useless use of %s in void context

(W) 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.

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.

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" 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 "%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 re−bound to the current values of such variables.

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.

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) 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.

Write on closed filehandle

(W) The filehandle you‘re writing to got itself closed sometime before now. Check your logic flow.

X outside of string

(F) You had a pack template that specified a relative position before the beginning of the string being unpacked. See *pack*.

x outside of string

(F) You had a pack template that specified a relative position after the end of the string being unpacked. See *pack*.

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 can‘t use `−l` on a filehandle

(F) 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. Use a filename instead.

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 use the wrapsuid script in the eg directory to put a setuid C wrapper around your script.

You need to quote "%s"

(W) 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.)

`[gs]etsockopt()` on closed fd

(W) You tried to get or set a socket option on a closed socket. Did you forget to check the return value of your `socket()` call? See *getsockopt*.

\1 better written as $1

(W) 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.

‘|’ 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;
```

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 non−empty 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*.

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*.

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 *Process terminated by SIGTERM/SIGINT in perlos2*.

NAME

perlsec − Perl security

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 built−in 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.

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 setuid 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*), and 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. Any variable set within an expression that has previously referenced a tainted value itself becomes tainted, even if it is logically impossible for the tainted value to influence the variable. Because taintedness is associated with each scalar value, some elements of an array can be tainted and others not.

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;   # Secure (doesn't use sh)
    system "echo $hid";         # Insecure
    system "echo $data";        # Insecure until PATH set

    $path = $ENV{'PATH'};       # $path now tainted

    $ENV{'PATH'} = '/bin:/usr/bin';
    $ENV{'IFS'} = '' if $ENV{'IFS'} ne '';

    $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, but...
    open(FOO,"−|")
        or exec 'echo', $arg;   # OK
```

```
$shout = ‘echo $arg‘;        # Insecure, $shout now tainted

unlink $data, $arg;          # Insecure
umask $arg;                  # Insecure

exec "echo $arg";            # Insecure
exec "echo", $arg;           # Secure (doesn’t use the shell)
exec "sh", ’−c’, $arg;       # Considered secure, alas!
```

If you try to do something insecure, you will get a fatal error saying something like "Insecure dependency" or "Insecure PATH". Note that you can still write an insecure **system** or **exec**, but only by explicitly doing something like the last example above.

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 following *is_tainted()* function.

```
sub is_tainted {
    return ! eval {
        join(’’,@_), kill 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. The only way to bypass the tainting mechanism is by referencing sub−patterns 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* for further discussion and examples.

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.)

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.

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 setting `$> = $<`. (Remember group IDs, too!) 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 back−tick 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 back−ticks 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. By the time we get to the **exec**, tainting is turned off, however, so be careful what you call and what you pass it.

```
    use English;
    die unless defined $pid = open(KID, "−|");
    if ($pid) {           # parent
        while (<KID>) {
            # do something
        }
        close KID;
    } else {
        $EUID = $UID;
        $EGID = $GID;    # XXX: initgroups() not called
        $ENV{PATH} = "/bin:/usr/bin";
        exec ’myprog’, ’arg1’, ’arg2’;
        die "can’t exec myprog: $!";
    }
```

A similar strategy would work for wildcard expansion via `glob`.

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 setuid 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.

Security Bugs

Beyond the obvious problems that stem from giving special privileges to systems as flexible as scripts, on many versions of Unix, setuid 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−setuid) 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 the setuid bit set, which doesn‘t help much. Alternately, it can simply ignore the setuid bit 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 setuid script feature isn‘t disabled, Perl will complain loudly that your setuid script is insecure. You‘ll need to either disable the kernel setuid 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 setuid 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.

See the program **wrapsuid** in the ***eg*** directory of your Perl distribution for a convenient way to do this automatically for all your setuid Perl programs. It moves setuid scripts into files with the same name plus a leading dot, and then compiles a wrapper like the one above for each of them.

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 setuid 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 5.003 of Perl, a bug in the code of **suidperl** could introduce a security hole in systems compiled with strict POSIX compliance.

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.

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 (Filter::* from CPAN). 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.

NAME

perltrap − Perl traps for the unwary

DESCRIPTION

The biggest trap of all is forgetting to use the **−w** switch; see *perlrun*. The second biggest trap is not making your entire program runnable under `use strict`.

Awk Traps

Accustomed **awk** users should take special note of the following:

- The English module, loaded via

  ```
  use English;
  ```

  allows you to refer to special variables (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 "`$`" or "@" in Perl.
- Arrays index from 0. Likewise string positions in `substr()` and `index()`.
- You have to decide whether your array has numeric or string indices.
- Hash values do not spring into existence upon mere reference.
- You have to decide whether you want to use string or numeric comparisons.
- Reading an input line does not split it for you. You get to split it yourself to an array. And the `split()` operator has different arguments.
- The current input line is normally in `$_`, 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      $#ARGV or scalar @ARGV
  ARGV[0]   $0
  FILENAME  $ARGV
  ```

```
FNR$. − something
FS(whatever you like)
NF$#Fld, or some such
NR$.
OFMT$#
OF$,
OR$\
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.

C Traps

Cerebral 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.
- There‘s no switch statement. (But it‘s easy to build one on the fly.)
- Variables begin with "`$`" or "@" in Perl.
- `printf()` does not implement the "*" format for interpolating field widths, but it‘s trivial to use interpolation of double−quoted strings to achieve the same effect.
- Comments begin with "#", not "/*".
- You can‘t take the address of anything, although a similar operator in Perl is the backslash, which creates a reference.
- `ARGV` must be capitalized. `$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.
- Signal handlers deal with signal names, not numbers. Use `kill −l` to find their names on your system.

Sed Traps

Seasoned **sed** programmers should take note of the following:

- Backreferences in substitutions use "`$`" rather than "\".
- The pattern matching metacharacters "(", ")", and "|" do not have backslashes in front.
- The range operator is `...`, rather than comma.

Shell Traps

Sharp shell programmers should take note of the following:

- The back−tick operator does variable interpolation without regard to the presence of single quotes in the command.
- The back−tick 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, back−ticks, 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.

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 built−ins are unary operators (like `chop()` and `chdir()`) and which are list operators (like `print()` and `unlink()`). (User−defined subroutines can be **only** list operators, never unary ones.) See *perlop*.
- 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 ($_ = <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.

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 Bill Middleton <***wjm@best.com*** for inclusion. Also note that at least some of these can be caught with **−w**.

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
```

  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−elem array
  sub sub2{ return(1,2,3)}        # return a 3−elem array
  @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 dumps core (SEGV)
  ```

- 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: Bare word 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
```

- Discontinuance

  In Perl 4 (and versions of Perl 5 before 5.004), `'\r'` characters in Perl code were silently allowed, although they could cause (mysterious!) failures in certain constructs, particularly here documents. Now, `'\r'` characters cause an immediate fatal error. (Note: In this example, the notation **\015** represents the incorrect line ending. Depending upon your text viewer, it will look different.)

  ```
  print "foo";\015
  print "bar";

  # perl4     prints: foobar
  # perl5.003 prints: foobar
  # perl5.004 dies: Illegal character \015 (carriage return)
  ```

  See *perldiag* for full details.

- Deprecation

  Some error messages will be different.

- Discontinuance

  Some bugs may have been inadvertently removed. :−)

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
  ```

Numerical Traps

Perl4−to−Perl5 traps having to do with numerical operators, operands, or output from same.

- Numerical

  Formatted output and significant digits

  ```
  print 7.373504 − 0, "\n";
  printf "%20.18f\n", 7.373504 − 0;
  ```

```
# Perl4 prints:
7.375039999999996141
7.37503999999999614

# Perl5 prints:
7.373504
7.37503999999999614
```

- 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 an null, instead of 0

  ```
  $p = ($test == 1);
  print $p,"\n";

  # perl4 prints: 0
  # perl5 prints:
  ```

  Also see the *, etc.* tests for another example of this new feature...

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
  ```

- (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:

    # Another example

    *fred = *barney; # fred is aliased to barney
    @barney = (1, 2, 4);
    # @fred;
    print "@fred";  # should print "1, 2, 4"

    # perl4 prints: 1 2 4
    # perl5 prints: In string, @fred now must be written as \@fred
```

- (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
```

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()` funkiness (array argument converted to scalar array count) This test could be added to t/op/sprintf.t

  ```
  @z = ('%s%s', 'foo', 'bar');
  $x = sprintf(@z);
  if ($x eq 'foobar') {print "ok 2\n";} else {print "not ok 2 '$x'\n";}

  # perl4 prints: ok 2
  # perl5 prints: not ok 2
  ```

  `printf()` works fine, though:

  ```
  printf STDOUT (@z);
  print "\n";

  # perl4 prints: foobar
  # perl5 prints: foobar
  ```

  Probably a bug.

Precedence Traps

Perl4−to−Perl5 traps involving precedence order.

- Precedence

  LHS vs. RHS when both sides are getting an op.

  ```
  @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 errors: Precedence problem: open FOO should be open(FOO)
  ```

- 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

  concatenation precedence over filetest operator?

  ```
      -e $foo .= "q"

      # perl4 prints: no output
      # perl5 prints: Can't modify -e in concatenation
  ```

- Precedence

  Assignment to value takes precedence over assignment to key in perl5 when using the shift operator on both sides.

  ```
      @arr = ( 'left', 'right' );
      $a{shift @arr} = shift @arr;
      print join( ' ', keys %a );

      # perl4 prints: left
      # perl5 prints: right
  ```

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: blah blah 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; };
  }
  ```

  `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.

  This is probably a bug, and may change in future versions of Perl.

- 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 back−ticks) is now a normal substitution, with no back−tick 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

  Under perl4 and upto version 5.003, a failed `m//g` match used to reset the internal iterator, so that subsequent `m//g` match attempts began from the beginning of the string. In perl version 5.004 and later, failed `m//g` matches do not reset the iterator position (which can be found using the `pos()` function—see *pos*).

  ```
  $test = "foop";
  for (1..3) {
      print $1 while ($test =~ /(o)/g);
      # pos $test = 0;     # to get old behavior
  }
  ```

```
# perl4     prints: oooooo
# perl5.004 prints: oo
```

You may always reset the iterator yourself as shown in the commented line to get the old behavior.

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 main'SeeYa
  # perl5 prints: SIGTERM is now main::1
  ```

  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 a,b,c;

  # perl4 prints: yup yup yup yup abc
  # perl5 prints: abc
  ```

- `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
  ```

OS Traps

- (SysV)

  Under HPUX, and some other SysV OS's, 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 OS's, `seek()` on a file opened to append `>>` now does the right thing w.r.t. the `fopen()` man page. 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
```

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
    # perl5 errors : In string, @somewhere now must be written as \@somewhere
```

- Interpolation

  Double−quoted strings may no longer end with an unescaped `$` or @.

```
    $foo = "foo$";
    $bar = "bar@";
    print "foo is $foo, bar is $bar\n";

    # perl4 prints: foo is foo$, bar is bar@
    # 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 `$$x`" used to interpolate the pid at that point, but now apparently tries to dereference `$x`. `$$` by itself still works fine, however.

  ```
  print "this is $$x\n";
  
  # perl4 prints: this is XXXx   (XXX is the current pid)
  # perl5 prints: this is
  ```

- 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 references.

  ```
  print "$foo{"
  
  perl 4 prints: {
  perl 5 prints: syntax error
  ```

- Interpolation

  Similarly, watch out for:

```
$foo = "array";
print "\$$foo{bar}\n";

# perl4 prints: $array{bar}
# perl5 prints: $
```

  Perl 5 is looking for `$array{bar}` which doesn‘t exist, but perl 4 is happy just to expand `$foo` to "array" 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 ")"
```

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.
```

Unclassified Traps

Everything else.

- Unclassified

  `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`.

As always, if any of these are ever officially declared as bugs, they‘ll be fixed and removed.

NAME

perlstyle − Perl style guide

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 `$^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 brace 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 multi−level 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 bites.
- 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 translations 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 **−w** in effect. Consider giving away your code. Consider changing your whole world view. Consider... oh, never mind.
- Be consistent.
- Be nice.

NAME

perlpod − plain old documentation

DESCRIPTION

A pod−to−whatever translator reads a pod file paragraph by paragraph, and translates it to the appropriate output format. There are three kinds of paragraphs:

- A verbatim paragraph, distinguished by being indented (that is, it starts with space or tab). It should be reproduced exactly, with tabs assumed to be on 8−column boundaries. There are no special formatting escapes, so you can‘t italicize or anything like that. A \ means \, and nothing else.
- A command. All command paragraphs start with "=", followed by an identifier, followed by arbitrary text that the command can use however it pleases. Currently recognized commands are

```
    =head1 heading
    =head2 heading
    =item text
    =over N
    =back
    =cut
    =pod
    =for X
    =begin X
    =end X
```

The "=pod" directive does nothing beyond telling the compiler to lay off parsing code through the next "=cut". It‘s useful for adding another paragraph to the doc if you‘re mixing up code and pod a lot.

Head1 and head2 produce first and second level headings, with the text in the same paragraph as the "=headn" directive forming the heading description.

Item, over, and back require a little more explanation: "=over" starts a section specifically for the generation of a list using "=item" commands. At the end of your list, use "=back" to end it. You will probably want to give "4" as the number to "=over", as some formatters will use this for indentation. This should probably be a default. Note also that there are some basic rules to using =item: don‘t use them outside of an =over/=back block, use at least one inside an =over/=back block, you don‘t _have_ to include the =back if the list just runs off the document, 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., i.e., things that looks nothing like bullets or numbers. If you start with bullets or numbers, stick with them, as many formatters use the first "=item" type to decide how to format the list.

For, begin, and end let you include sections that are not interpreted as pod text, but passed directly to particular formatters. A formatter that can utilize that format will use the section, otherwise it will be completely ignored. The directive "=for" specifies that the entire next paragraph is in the format indicated by the first word after "=for", like this:

```
 =for html <br>
  <p> This is a raw HTML paragraph </p>
```

The paired commands "=begin" and "=end" work very similarly to "=for", but instead of only accepting a single paragraph, all text from "=begin" to a paragraph with a matching "=end" are treated as a particular format.

Here are some examples of how to use these:

```
 =begin html

 <br>Figure 1.<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.)

And don‘t forget, when using any command, that the command lasts up until the end of the **paragraph**, not the line. Hence in the examples below, you can see the blank lines after each command to end its paragraph.

Some examples of lists include:

```
=over 4

=item *

First item

=item *

Second item

=back

=over 4

=item Foo()

Description of Foo function

=item Bar()

Description of Bar function

=back
```

- An ordinary block of text. It will be filled, and maybe even justified. Certain interior sequences are recognized both here and in commands:

```
    I<text>     italicize text, used for emphasis or variables
    B<text>     embolden text, used for switches and programs
    S<text>     text contains non-breaking spaces
    C<code>     literal code
    L<name>     A link (cross reference) to name
                    L<name>             manual page
                    L<name/ident>       item in manual page
                    L<name/"sec">       section in other manual page
                    L<"sec">            section in this manual page
                                        (the quotes are optional)
                    L</"sec">           ditto
    F<file>     Used for filenames
    X<index>    An index entry
    ZE<lt>E<gt> A zero-width character
    E<escape>   A named character (very similar to HTML escapes)
                    E<lt>               A literal <
```

```
                                        E<gt>       A literal >
                                        (these are optional except in other interior
                                         sequences and when preceded by a capital letter)
                                        E<n>        Character number n (probably in ASCII)
                                        E<html>             Some non−numeric HTML entity, such
                                                            as E<Agrave>
```

That‘s it. The intent is simplicity, not power. I wanted paragraphs to 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**). I wanted the translator (and not me) to worry about whether " or ’ is a left quote or a right quote within filled text, and I wanted it to leave the quotes alone, dammit, in verbatim mode, so I could slurp in a working program, shift it over 4 spaces, and have it print out, er, verbatim. And presumably in a constant width font.

In particular, you can leave things like this verbatim in your text:

```
    Perl
    FILEHANDLE
    $variable
    function()
    manpage(3r)
```

Doubtless a few other commands or sequences will need to be added along the way, but I‘ve gotten along surprisingly well with just these.

Note that I‘m not at all claiming this to be sufficient for producing a book. I‘m just trying to make an idiot−proof common source for nroff, TeX, and other markup languages, as used for online documentation. Translators exist for **pod2man** (that‘s for nroff(1) and troff(1)), **pod2html**, **pod2latex**, and **pod2fm**.

Embedding Pods in Perl Modules

You can embed pod documentation in your Perl scripts. Start your documentation with a "=head1" command at the beginning, and end it with a "=cut" command. Perl will ignore the pod text. See any of the supplied library modules for examples. If you‘re going to put your pods at the end of the file, and you‘re using an __END__ or __DATA__ cut mark, make sure to put a blank line there before the first pod directive.

```
    __END__

    =head1 NAME

    modern − I am a modern module
```

If you had not had that blank line there, then the translators wouldn‘t have seen it.

Common Pod Pitfalls

- Pod translators usually will require paragraphs to be separated by completely empty lines. If you have an apparently blank line with some spaces on it, this can cause odd formatting.
- Translators will mostly add wording around a L<> link, so that `L<foo(1)>` becomes "the *foo*(1) manpage", for example (see **pod2man** for details). Thus, you shouldn‘t write things like `the L<foo> manpage`, if you want the translated document to read sensibly.
- The script ***pod/checkpods.PL*** in the Perl source distribution provides skeletal checking for lines that look blank but aren‘t **only**, but is there as a placeholder until someone writes Pod::Checker. The best way to check your pod is to pass it 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.

SEE ALSO

pod2man and *PODs: Embedded Documentation in perlsyn*

AUTHOR

Larry Wall

NAME

perlbook − Perl book information

DESCRIPTION

You can order 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.ora.com/ for an online order form.

Programming Perl, *Second Edition* is a reference work that covers nearly all of Perl, while *Learning Perl* is a tutorial that covers the most frequently used subset of the language. You might also check out the very handy, inexpensive, and compact *Perl 5 Desktop Reference*, especially when the thought of lugging the 676−page Camel around doesn‘t make much sense.

```
Programming Perl, Second Edition (the Camel Book):
    ISBN 1−56592−149−6      (English)

Learning Perl (the Llama Book):
    ISBN 1−56592−042−2      (English)
    ISBN 4−89502−678−1      (Japanese)
    ISBN 2−84177−005−2      (French)
    ISBN 3−930673−08−8      (German)

Perl 5 Desktop Reference (the reference card):

    ISBN 1−56592−187−9      (brief English)
```

NAME

perlembed − how to embed perl in your C program

DESCRIPTION

PREAMBLE

Do you want to:

Use C from Perl?

Read *perlcall* and *perlxs*.

Use a UNIX program from Perl?

Read about back−quotes and about `system` and `exec` in *perlfunc*.

Use Perl from Perl?

Read about *do* and *eval* and *require* and *use*.

Use C from C?

Rethink your design.

Use Perl from C?

Read on...

ROADMAP

Compiling your C program

There‘s one example in each of the eight sections:

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

This documentation is Unix specific; if you have information about how to embed Perl on other platforms, please send e−mail to <***orwant@tpj.com***.

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 5.00305, 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::Embed. (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.

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*, from the source distribution. Here‘s a bastardized, non−portable version of *miniperlmain.c* containing the essentials of embedding:

```
    #include <EXTERN.h>               /* from the Perl distribution     */
    #include <perl.h>                 /* from the Perl distribution     */

    static PerlInterpreter *my_perl;  /***    The Perl interpreter    ***/
    int main(int argc, char **argv, char **env)
    {
        my_perl = perl_alloc();
        perl_construct(my_perl);
        perl_parse(my_perl, NULL, argc, argv, (char **)NULL);
        perl_run(my_perl);
        perl_destruct(my_perl);
        perl_free(my_perl);
    }
```

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.

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()`*.

Calling a Perl subroutine from your C program

To call individual Perl subroutines, you can use any of the **perl_call_*** functions documented in the *perlcall* man page. In this example we‘ll use *perl_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 };
        my_perl = perl_alloc();
        perl_construct(my_perl);

        perl_parse(my_perl, NULL, argc, argv, NULL);

        /*** skipping perl_run() ***/

        perl_call_argv("showtime", G_DISCARD | G_NOARGS, args);

        perl_destruct(my_perl);
        perl_free(my_perl);
    }
```

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*, but in general it‘s considered good practice to ensure proper initialization of library code, including execution of all object `DESTROY` methods and package `END {}` blocks.

If you want to pass arguments to the Perl subroutine, you can add strings to the `NULL`−terminated `args` list passed to *perl_call_argv*. For other data types, or to examine return values, you‘ll need to manipulate the Perl stack. That‘s demonstrated in the last section of this document: *Fiddling with the Perl stack from your C program*.

Evaluating a Perl statement from your C program

One way to evaluate pieces of Perl code is to use *perl_eval_sv()*. We‘ve wrapped this inside our own *perl_eval()* function, which converts a command string to an SV, passing this and the *G_DISCARD* flag to *perl_eval_sv()*.

Arguably, this is the only routine you‘ll ever need to execute snippets of Perl code from within your C program. Your string can be as long as you wish; it can contain multiple statements; it can employ *use*, *require* and *do* to include external Perl files.

Our *perl_eval()* 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;

   I32 perl_eval(char *string)
   {
     return perl_eval_sv(newSVpv(string,0), G_DISCARD);
   }

   main (int argc, char **argv, char **env)
   {
     char *embedding[] = { "", "−e", "0" };
     STRLEN length;

     my_perl = perl_alloc();
     perl_construct( my_perl );

     perl_parse(my_perl, NULL, 3, embedding, NULL);
     perl_run(my_perl);
                                        /** Treat $a as an integer **/
     perl_eval("$a = 3; $a **= 2");
     printf("a = %d\n", SvIV(perl_get_sv("a", FALSE)));
```

```
                                                   /** Treat $a as a float **/
       perl_eval("$a = 3.14; $a **= 2");
       printf("a = %f\n", SvNV(perl_get_sv("a", FALSE)));

                                                   /** Treat $a as a string **/
       perl_eval("$a = 'rekcaH lreP rehtonA tsuJ'; $a = reverse($a); ");
       printf("a = %s\n", SvPV(perl_get_sv("a", FALSE), length));

       perl_destruct(my_perl);
       perl_free(my_perl);
     }
```

All of those strange functions with *sv* in their names help convert Perl scalars to C types. They‘re described in *perlguts*.

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
```

Performing Perl pattern matches and substitutions from your C program

Our *perl_eval()* 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()*.

```
   char match(char *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(char *string[], char *pattern);
```

Given a pointer to a string and an `=~` operation (e.g., `s/bob/robert/g` or `tr[A-Z][a-z]`), `substitute()` modifies the string according to the operation, returning the number of substitutions made.

```
   int matches(char *string, char *pattern, char **matches[]);
```

Given a string, a pattern, and a pointer to an empty array of strings, `matches()` evaluates `$string =~ $pattern` in an array context, and fills in *matches* with the array elements (allocating memory as it does so), 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;
   I32 perl_eval(char *string)
   {
      return perl_eval_sv(newSVpv(string,0), G_DISCARD);
   }
   /** match(string, pattern)
   **
   ** Used for matches in a scalar context.
   **
   ** Returns 1 if the match was successful; 0 otherwise.
   **/
   char match(char *string, char *pattern)
   {
     char *command;
```

```
    command = malloc(sizeof(char) * strlen(string) + strlen(pattern) + 37);
    sprintf(command, "$string = ’%s’; $return = $string =~ %s",
                     string, pattern);
    perl_eval(command);
    free(command);
    return SvIV(perl_get_sv("return", FALSE));
  }
  /** substitute(string, pattern)
  **
  ** Used for =~ operations that modify their left−hand side (s/// and tr///)
  **
  ** Returns the number of successful matches, and
  ** modifies the input string if there were any.
  **/
  int substitute(char *string[], char *pattern)
  {
    char *command;
    STRLEN length;
    command = malloc(sizeof(char) * strlen(*string) + strlen(pattern) + 35);
    sprintf(command, "$string = ’%s’; $ret = ($string =~ %s)",
                     *string, pattern);
    perl_eval(command);
    free(command);
    *string = SvPV(perl_get_sv("string", FALSE), length);
    return SvIV(perl_get_sv("ret", FALSE));
  }
  /** matches(string, pattern, matches)
  **
  ** Used for matches in an array context.
  **
  ** Returns the number of matches,
  ** and fills in **matches with the matching substrings (allocates memory!)
  **/
  int matches(char *string, char *pattern, char **match_list[])
  {
    char *command;
    SV *current_match;
    AV *array;
    I32 num_matches;
    STRLEN length;
    int i;
    command = malloc(sizeof(char) * strlen(string) + strlen(pattern) + 38);
    sprintf(command, "$string = ’%s’; @array = ($string =~ %s)",
                     string, pattern);
    perl_eval(command);
    free(command);
    array = perl_get_av("array", FALSE);
    num_matches = av_len(array) + 1; /** assume $[ is 0 **/
    *match_list = (char **) malloc(sizeof(char *) * num_matches);
    for (i = 0; i <= num_matches; i++) {
      current_match = av_shift(array);
      (*match_list)[i] = SvPV(current_match, length);
    }
    return num_matches;
```

```
        }
        main (int argc, char **argv, char **env)
        {
          char *embedding[] = { "", "-e", "0" };
          char *text, **match_list;
          int num_matches, i;
          int j;
          my_perl = perl_alloc();
          perl_construct( my_perl );
          perl_parse(my_perl, NULL, 3, embedding, NULL);
          perl_run(my_perl);

          text = (char *) malloc(sizeof(char) * 486); /** A long string follows! **/
          sprintf(text, "%s", "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", match_list[i]);
          printf("\n");
          for (i = 0; i < num_matches; i++) {
            free(match_list[i]);
          }
          free(match_list);
          /** 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", text);
          }
          /** Attempt a substitution **/
          if (!substitute(&text, "s/Perl/C/")) {
            printf("substitute: s/Perl/C...No substitution made.\n\n");
          }
          free(text);
          perl_destruct(my_perl);
          perl_free(my_perl);
        }
```

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.
```

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*.

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 built−in 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 */
  perl_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[2];

      my_perl = perl_alloc();
      perl_construct( my_perl );

      my_argv[1] = (char *) malloc(10);
      sprintf(my_argv[1], "power.pl");

      perl_parse(my_perl, NULL, argc, my_argv, NULL);
      perl_run(my_perl);

      PerlPower(3, 4);                      /*** Compute 3 ** 4 ***/

      perl_destruct(my_perl);
      perl_free(my_perl);
    }
```

Compile and run:

```
    % cc -o power power.c `perl -MExtUtils::Embed -e ccopts -e ldopts`

    % power
    3 to the 4th power is 81.
```

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 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 *perl_call_argv*, 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* whenever possible.

```
 package Embed::Persistent;
```

```
#persistent.pl

use strict;
use vars '%Cache';

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;
}

#borrowed from Safe.pm
sub delete_package {
    my $pkg = shift;
    my ($stem, $leaf);

    no strict 'refs';
    $pkg = "main::$pkg\::";    # expand to full symbol table name
    ($stem, $leaf) = $pkg =~ m/(.*::)(\w+::)$/;

    my $stem_symtab = *{$stem}{HASH};

    delete $stem_symtab->{$leaf};
}

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

    static PerlInterpreter *perl = NULL;

    int
    main(int argc, char **argv, char **env)
    {
        char *embedding[] = { "", "persistent.pl" };
        char *args[] = { "", DO_CLEAN, NULL };
        char filename [1024];
        int exitstatus = 0;

        if((perl = perl_alloc()) == NULL) {
           fprintf(stderr, "no memory!");
           exit(1);
        }
        perl_construct(perl);

        exitstatus = perl_parse(perl, NULL, 2, embedding, NULL);

        if(!exitstatus) {
           exitstatus = perl_run(perl);

           while(printf("Enter file name: ") && gets(filename)) {

               /* call the subroutine, passing it the filename as an argument */
               args[0] = filename;
               perl_call_argv("Embed::Persistent::eval_file",
                              G_DISCARD | G_EVAL, args);

               /* check $@ */
               if(SvTRUE(GvSV(errgv)))
                   fprintf(stderr, "eval error: %s\n", SvPV(GvSV(errgv),na));
           }
        }

        perl_destruct_level = 0;
        perl_destruct(perl);
        perl_free(perl);
```

```
         exit(exitstatus);
     }
```

Now compile:

```
 % cc −o persistent persistent.c ‘perl −MExtUtils::Embed −e ccopts −e ldopts‘
```

Here‘s a 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
```

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, the global variable `perl_destruct_level` is set to , since extra cleaning isn‘t needed when a program has only one interpreter.

Setting `perl_destruct_level` to `1` makes everything squeaky clean:

```
 perl_destruct_level = 1;

 while(1) {
     ...
     /* reset global variables here with perl_destruct_level = 1 */
     perl_construct(my_perl);
     ...
     /* clean and reset _everything_ during perl_destruct */
     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.

Now suppose we have more than one interpreter instance running at the same time. This is feasible, but only if you used the `−DMULTIPLICITY` flag when building Perl. By default, that sets `perl_destruct_level` to `1`.

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 = perl_alloc(),
        *two_perl = perl_alloc();
    char *one_args[] = { "one_perl", SAY_HELLO };
    char *two_args[] = { "two_perl", SAY_HELLO };

    perl_construct(one_perl);
    perl_construct(two_perl);

    perl_parse(one_perl, NULL, 3, one_args, (char **)NULL);
    perl_parse(two_perl, NULL, 3, two_args, (char **)NULL);

    perl_run(one_perl);
    perl_run(two_perl);

    perl_destruct(one_perl);
    perl_destruct(two_perl);

    perl_free(one_perl);
    perl_free(two_perl);
}
```

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
```

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:

```
#ifdef __cplusplus
#  define EXTERN_C extern "C"
#else
#  define EXTERN_C extern
#endif
```

```
static void xs_init _((void));

EXTERN_C void boot_DynaLoader _((CV* cv));
EXTERN_C void boot_Socket _((CV* cv));

EXTERN_C void
xs_init()
{
       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* and *perlguts* for more details.

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.

AUTHOR

Jon Orwant and <***orwant@tpj.com*** and Doug MacEachern <***dougm@osf.org***, with small contributions from Tim Bunce, Tom Christiansen, Hallvard Furuseth, Dov Grobgeld, and Ilya Zakharevich.

Check out Doug‘s article on embedding in Volume 1, Issue 4 of The Perl Journal. Info about TPJ is available from http://tpj.com.

February 1, 1997

Some of this material is excerpted from Jon Orwant‘s book: *Perl 5 Interactive*, Waite Group Press, 1996 (ISBN 1−57169−064−6) and appears courtesy of Waite Group Press.

COPYRIGHT

NAME

perlapio − perl‘s IO abstraction interface.

SYNOPSIS

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

    PerlIO *PerlIO_open(const char *,const char *);
    int     PerlIO_close(PerlIO *);

    int     PerlIO_stdoutf(const char *,...)
    int     PerlIO_puts(PerlIO *,const char *);
    int     PerlIO_putc(PerlIO *,int);
    int     PerlIO_write(PerlIO *,const void *,size_t);
    int     PerlIO_printf(PerlIO *, const char *,...);
    int     PerlIO_vprintf(PerlIO *, const char *, va_list);
    int     PerlIO_flush(PerlIO *);

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

    int     PerlIO_getc(PerlIO *);
    int     PerlIO_ungetc(PerlIO *,int);
    int     PerlIO_read(PerlIO *,void *,size_t);

    int     PerlIO_fileno(PerlIO *);
    PerlIO *PerlIO_fdopen(int, const char *);
    PerlIO *PerlIO_importFILE(FILE *);
    FILE   *PerlIO_exportFILE(PerlIO *);
    FILE   *PerlIO_findFILE(PerlIO *);
    void    PerlIO_releaseFILE(PerlIO *,FILE *);

    void    PerlIO_setlinebuf(PerlIO *);

    long    PerlIO_tell(PerlIO *);
    int     PerlIO_seek(PerlIO *,off_t,int);
    int     PerlIO_getpos(PerlIO *,Fpos_t *)
    int     PerlIO_setpos(PerlIO *,Fpos_t *)
    void    PerlIO_rewind(PerlIO *);

    int     PerlIO_has_base(PerlIO *);
    int     PerlIO_has_cntptr(PerlIO *);
    int     PerlIO_fast_gets(PerlIO *);
    int     PerlIO_canset_cnt(PerlIO *);

    char   *PerlIO_get_ptr(PerlIO *);
    int     PerlIO_get_cnt(PerlIO *);
    void    PerlIO_set_cnt(PerlIO *,int);
    void    PerlIO_set_ptrcnt(PerlIO *,char *,int);
    char   *PerlIO_get_base(PerlIO *);
    int     PerlIO_get_bufsiz(PerlIO *);
```

DESCRIPTION

Perl‘s source code should use the above functions instead of those defined in ANSI C‘s *stdio.h*, *perlio.h* will the `#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 *

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

`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()` arguments are the same.

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

These are is `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 to `fread()` and `fwrite()`. Note that arguments are different, there is only one "count" and order has "file" first.

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

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

PerlIO_ungetc(c,f)

This corresponds to `ungetc()`. Note that arguments have been revised to have "file" first.

PerlIO_getc(f)

This corresponds to `getc()`.

PerlIO_eof(f)

This corresponds to `feof()`.

PerlIO_error(f)

This corresponds to `ferror()`.

PerlIO_fileno(f)

This corresponds to `fileno()`, note that on some platforms, the meaning of "fileno" may not match UNIX.

PerlIO_clearerr(f)

This corresponds to `clearerr()`, i.e., clears ‘eof’ and ‘error’ flags for the "stream".

PerlIO_flush(f)

This corresponds to `fflush()`.

PerlIO_tell(f)

This corresponds to `ftell()`.

PerlIO_seek(f,o,w)

This corresponds to `fseek()`.

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

These correspond to `fgetpos()` and `fsetpos()`. 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()`. Note may be redefined in terms of `PerlIO_seek()` at some point.

`PerlIO_tmpfile()`

This corresponds to `tmpfile()`, i.e., returns an anonymous PerlIO which will automatically be deleted when closed.

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 if perlio is implemented on top of (say) sfio then mechanisms must exist to create a FILE * which can be passed to library code which is going to use stdio calls.

PerlIO_importFILE(f,flags)

Used to get a PerlIO * from a FILE *. May need additional arguments, interface under review.

PerlIO_exportFILE(f,flags)

Given an PerlIO * return a ‘native’ FILE * suitable for passing to code expecting to be compiled and linked with ANSI C *stdio.h*.

The fact that such a FILE * has been ‘exported’ is recorded, and may affect future PerlIO operations on the original PerlIO *.

PerlIO_findFILE(f)

Returns previously ‘exported’ FILE * (if any). Place holder until interface is fully defined.

PerlIO_releaseFILE(p,f)

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

PerlIO_setlinebuf(f)

This corresponds to `setlinebuf()`. Use is deprecated pending further discussion. (Perl core uses it *only* when "dumping" is has nothing to do with `$|` auto−flush.)

In addition to user API 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. This section is really of interest to only those concerned with detailed perl−core behaviour or implementing a PerlIO mapping.

PerlIO_has_cntptr(f)

Implementation can return pointer to current position in the "buffer" and a count of bytes available in the buffer.

PerlIO_get_ptr(f)

Return pointer to next readable byte in buffer.

PerlIO_get_cnt(f)

Return count of readable bytes in the buffer.

PerlIO_canset_cnt(f)

Implementation can adjust its idea of number of bytes in the buffer.

PerlIO_fast_gets(f)

Implementation has all the interfaces required to allow perl‘s fast code to handle <FILE mechanism.

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

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`.

PerlIO_set_cnt(f,c)

Obscure − set count of bytes in the buffer. Deprecated. Currently used in only doio.c to force count < −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".

PerlIO_has_base(f)

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.

PerlIO_get_bufsiz(f)

Return *total size* of buffer.

NAME

perlxs − XS language reference manual

DESCRIPTION

Introduction

XS is a language used to create an extension interface between Perl and some C library which one wishes to use with Perl. The XS interface is combined with the library to create a new library which can be linked to Perl. An **XSUB** is a function in the XS language and is the core component of the Perl application interface.

The XS compiler is called **xsubpp**. This compiler will embed the constructs necessary to let an XSUB, which is really a C function in disguise, manipulate Perl values and creates the glue necessary to let Perl access the XSUB. The compiler uses **typemaps** to determine how to map C function parameters and variables to Perl values. The default typemap handles many common C types. A supplement typemap must be created to handle special structures and types for the library being linked.

See *perlxstut* for a tutorial on the whole extension creation process.

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.

The Anatomy of an XSUB

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
```

When using C pointers the indirection operator `*` should be considered part of the type and the address operator & should be considered part of the variable, as is demonstrated in the `rpcb_gettime()` function above. See the section on typemaps for more about handling qualifiers and unary operators in C types.

The function name and the return type must be placed on separate lines.

```
  INCORRECT                        CORRECT

  double sin(x)                    double
    double x                       sin(x)
                                     double x
```

The function body 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.

```
  CORRECT

  double
  sin(x)
  double x
```

The Argument Stack

The argument stack is used to store the values which are sent as parameters to the XSUB and to store the XSUB's return value. In reality all Perl functions keep their values on this stack at 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.

The RETVAL Variable

The RETVAL variable is a magic variable which always matches the return type of the C library function. The **xsubpp** compiler will supply this variable in each XSUB and by default will use it 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 supply a RETVAL variable for that function. When using the PPCODE: directive the RETVAL variable is not needed, unless used explicitly.

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 *truely* `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 "truely−void" and "old−practice−declared−as−void" functions. Hence your code is at mercy of this heuristics unless you use `SV *` as return value.)

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 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
```

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 ]
```

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.

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_
```

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, such as the `sin()` function above, the RETVAL variable is automatically designated as an output value. In 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);
```

The CODE: Keyword

This keyword is used in more complicated XSUBs which require special handling for the C function. The RETVAL variable is available but will not be returned unless it is specified under the OUTPUT: keyword.

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
```

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
```

The NO_INIT Keyword

The NO_INIT keyword is used to indicate that a function parameter is being used as only 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 as only 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
```

Initializing Function Parameters

Function parameters are normally initialized with their values from the argument stack. The typemaps contain the code segments which are used to transfer the Perl values to the C parameters. The programmer, however, is allowed to override the typemaps and supply alternate initialization code.

The following code demonstrates how to supply initialization code for function parameters. The initialization code is eval'd by the compiler before it is added to the output so anything which should be interpreted literally, such as double quotes, must be protected with backslashes.

```
     bool_t
     rpcb_gettime(host,timep)
          char *host = (char *)SvPV(ST(0),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.

Default Parameter Values

Default values can be specified for function parameters by placing an assignment statement in the parameter list. The default value may be a number or a string. 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. Perl will call this XSUB 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
```

The PREINIT: Keyword

The PREINIT: keyword allows extra variables to be declared before the typemaps are expanded. If a variable is declared in a CODE: block then that variable will follow any typemap code. This may result in a C syntax error. To force the variable to be declared before the typemap code, place it into a PREINIT: block. 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
```

A 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
```

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 this XSUB contains a comment like `/*scope*/` then scoping will automatically be enabled for that XSUB.

To enable scoping:

```
    SCOPE: ENABLE
```

To disable scoping:

```
    SCOPE: DISABLE
```

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
```

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";
          CODE:
                  if( items > 1 )
                       host = (char *)SvPV(ST(1), na);
                  RETVAL = rpcb_gettime( host, &timep );
          OUTPUT:
          timep
          RETVAL
```

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 are not used together within the same XSUB.

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 called `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");
```

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 mneumonic 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) = &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 *API LISTING in perlguts* for other `XSRETURN` macros.

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
```

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.

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");
```

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
```

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
```

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* for information about Perl prototypes.

```
    bool_t
    rpcb_gettime(timep, ...)
          time_t timep = NO_INIT
          PROTOTYPE: $;$
          PREINIT:
          char *host = "localhost";
          CODE:
                  if( items > 1 )
                       host = (char *)SvPV(ST(1), na);
                  RETVAL = rpcb_gettime( host, &timep );
          OUTPUT:
          timep
```

```
                RETVAL
```

The ALIAS: Keyword

The ALIAS: keyword allows an XSUB to have two 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
```

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 |
```

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 *"The ALIAS: Keyword"*), or maybe via the `items` variable (see *"Variable−length Parameter Lists"*). 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 );
```

The & Unary Operator

The & unary operator is used to tell the compiler that it should dereference the object when it calls the C function. This is used when a CODE: block is not used and the object is a not a pointer type (the object is an `int` or `long` but not a `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
```

Inserting Comments and C Preprocessor Directives

C preprocessor directives are allowed within BOOT:, PREINIT: INIT:, CODE:, PPCODE:, 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.

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.

Using XS With C++

If a function is defined as a C++ method then it will assume 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 method is defined as static 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 functions will expect an object as the first parameter. The xsubpp compiler will call that object `THIS` and will use it to call the specified method. So in the C++ code the `blue()` and `set_blue()` methods will be called in the following manner.

```
     RETVAL = THIS->blue();

     THIS->set_blue( val );
```

If the function‘s name is **DESTROY** then the C++ `delete` function will be called and `THIS` will be given as its parameter.

```
     void
     color::DESTROY()
```

The C++ code will call `delete`.

```
     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 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 referenc
                XSRETURN_UNDEF;
        }
```

Interface Strategy

When designing an interface between Perl and a C library a straight translation from C to XS is often sufficient. The interface will often be very C−like and occasionally nonintuitive, especially when the C function modifies one of its parameters. 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 which modify their parameters. The XSUBs for these functions may be able to return lists to Perl, or may be candidates to return undef or an empty list in case of failure.

Identify which values are used by only the C and XSUB functions themselves. 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 can be handled in XS with the & unary operator on the variable name while others will require the use of the * operator on the type name. In general it is easier to work with the & operator.

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.

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.

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 labeled `TYPEMAP`, `INPUT`, and `OUTPUT`. 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. Each of the sections of the typemap must be preceded by one of the TYPEMAP, INPUT, or OUTPUT keywords.

The default typemap in the `ext` 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
```

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";
```

XS VERSION

This document covers features supported by `xsubpp` 1.935.

AUTHOR

Dean Roehrich <***roehrich@cray.com*** Jul 8, 1996

NAME

perlXStut − Tutorial for XSUBs

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* 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 to ease the reader slowly into building extensions.

VERSION CAVEAT

This tutorial tries hard to keep up with the latest development versions of Perl. This often means that it is sometimes in advance of the latest released version of Perl, and that certain features described here might not work on earlier versions. This section will keep track of when various features were added to Perl 5.

- In versions of Perl 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 Perl 5.002 prior to version beta 3, the line in the .xs file about "PROTOTYPES: DISABLE" will cause a compiler error. Simply remove that line from the file.

- In versions of Perl 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".

DYNAMIC VERSUS STATIC

It is commonly thought that if a system does not have the capability to load a library dynamically, you cannot build XSUBs. This is incorrect. You *can* build them, but you must link the XSUB‘s 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.

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 created.

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;

require Exporter;
require DynaLoader;

@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.
@EXPORT = qw(

);
$VERSION = '0.01';

bootstrap Mytest $VERSION;

# Preloaded methods go here.

# Autoload methods go after __END__, and are processed by the autosplit progr

1;
__END__
# Below is the stub of documentation for your module. You better edit it!
```

And the Mytest.xs file should look something like this:

```
#ifdef __cplusplus
extern "C" {
#endif
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#ifdef __cplusplus
}
#endif

PROTOTYPES: DISABLE

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");
```

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 shortened for clarity):

```
        % make
        umask 0 && cp Mytest.pm ./blib/Mytest.pm
        perl xsubpp −typemap typemap Mytest.xs >Mytest.tc && mv Mytest.tc Mytest.c
        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
```

Now, although there is already a test.pl template ready for us, for this example only, we‘ll create a special test script. Create a file called hello that looks like this:

```
        #! /opt/perl5/bin/perl

        use ExtUtils::testlib;

        use Mytest;

        Mytest::hello();
```

Now we run the script and we should see the following output:

```
        % perl hello
        Hello, world!
        %
```

EXAMPLE 2

Now let‘s add to our extension a subroutine that will take a single argument and return 1 if the argument is even, 0 if the argument 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. The semi−colon at the end of that line is also optional.

Any white space may be between the "int" and "input". It is also okay for the four lines starting at the "CODE:" line to not be indented. However, for readability purposes, it is suggested that you indent them 8 spaces (or one normal tab stop).

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.

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/perl5.002b2/bin/perl (lots of −I arguments) test.pl
1..4
ok 1
ok 2
ok 3
ok 4
%
```

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 files <extension>.pm and <extension>.xs 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 and invoking the Makefile created a directory 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 use 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 that are set in the .pm file 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. The @EXPORT array tells Perl which of the extension‘s routines should be placed into the calling package‘s namespace.

It‘s important to select what to export carefully. Do NOT export method names and do NOT export anything else *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 then you can export any of the functions via another array, called @EXPORT_OK.

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.

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", and ensure all your test files end with the suffix ".t". The Makefile will properly run all these test files.

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 out "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.

You might be wondering if you can round a constant. To see what happens, add the following line to test.pl temporarily:

```
        &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!

WHAT‘S NEW HERE?

Two things are new here. First, we‘ve made some changes to Makefile.PL. In this case, we‘ve specified an extra library to link in, the math library libm. We‘ll talk later about how to write XSUBs that can call every routine in a library.

Second, the value of the function is being passed back not as the function‘s return value, but through the same variable that was passed into the function.

INPUT AND OUTPUT PARAMETERS

You specify the parameters that will be passed into the XSUB just after you declare the function return value and name. Each parameter line starts with optional white space, and may have an optional terminating semicolon.

The list of output parameters occurs 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, the value we wanted returned was contained in the same variable we passed in, so we listed it (and not RETVAL) in the OUTPUT: section.

THE XSUBPP COMPILER

The compiler xsubpp takes the XS code in the .xs file and converts 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.

THE TYPEMAP FILE

The xsubpp compiler 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 part attempts to map various C data types to a coded flag, which has some correspondence with the various Perl types. The second part contains C code which xsubpp uses for input parameters. The third part contains C code which xsubpp uses for output parameters. We‘ll talk more about the C code later.

Let‘s now take a look at a portion of the .c file created for our extension.

```
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 marked 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.

WARNING

In general, it‘s not a good idea to write extensions that modify their input parameters, as in Example 3. However, to accommodate better calling pre−existing C routines, which often do modify their input parameters, this behavior is tolerated. The next example will show how to do this.

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 testlib 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(a, b, c)
        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

        static ::       libmylib$(LIB_EXT)

        libmylib$(LIB_EXT): $(O_FILES)
                $(AR) cr libmylib$(LIB_EXT) $(O_FILES)
                $(RANLIB) libmylib$(LIB_EXT)

        ';
        }
```

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 following key−value pair to the WriteMakefile call:

```
'MYEXTLIB' => 'mylib/libmylib$(LIB_EXT)',
```

and a new replacement subroutine too:

```
sub MY::postamble {
'
$(MYEXTLIB): mylib/Makefile
        cd mylib && $(MAKE)
';
}
```

(Note: Most makes will require that there be a tab character that indents the line "cd mylib && `$(MAKE)",` similarly for the Makefile in the subdirectory.)

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 lines setting @EXPORT to @EXPORT_OK (there are two: one in the line beginning "use vars" and one setting the array itself). 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 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 see 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 often useful not to check for equality, but rather the difference being below a certain epsilon factor, 0.01 in this case)

Run "make test" and all should be well.

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 declaration with the full path to the mylib.h header file.
- 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 available to the Perl script (in this case, by calling `&main::TESTVAL).` There‘s also some XS code to allow calls to the `constant` routine.
- The .pm file has 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 going to be used by only 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 remove most or all of the items in the @EXPORT array.
- If our include file contained #include directives, these would not be processed at all 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 the addition of only 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 specified simply that the library to be created here was a static archive (as opposed to a dynamically loadable library) and provided the commands to build it.

SPECIFYING ARGUMENTS TO XSUBPP

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 in the .xs file, you are really passing three pieces of information for each one 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 exact way in which the argument should be used in the call to the library function from this XSUB. This would mean whether or not to place a "&" before the argument or not, meaning the argument expects to be passed the address of the specified data type.

There is a difference between the two arguments in this hypothetical function:

```
int
foo(a,b)
        char    &a
        char *  b
```

The first 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 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 identically parse the following function argument lists:

```
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 complete example 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.

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. The "ST" is actually a macro that points to the n'th argument on the argument stack. ST(0) is thus 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.

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 loaded only when called depends on where in the .pm file the subroutine definition is placed.

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 man page format, then placed in the blib directory. It will be copied to Perl's man page 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.

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.

SEE ALSO

For more information, consult *perlguts*, *perlxs*, *perlmod*, and *perlpod*.

Author

Jeff Okamoto <***okamoto@corp.hp.com***

Reviewed and assisted by Dean Roehrich, Ilya Zakharevich, Andreas Koenig, and Tim Bunce.

Last Changed

1996/7/10

NAME

perlguts − Perl‘s Internal Functions

DESCRIPTION

This document attempts to describe some of the internal functions of the Perl executable. It is far from complete and probably contains many errors. Please refer any questions or comments to the author below.

Variables

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.

What is an "IV"?

Perl uses a special typedef IV which is a simple integer type that is guaranteed to be large enough to hold a pointer (as well as an integer).

Perl also uses two special typedefs, I32 and I16, which will always be at least 32−bits and 16−bits long, respectively.

Working with SV‘s

An SV can be created and loaded with one command. There are four types of values that can be loaded: an integer value (IV), a double (NV), a string, (PV), and another scalar (SV).

The four routines are:

```
SV*  newSViv(IV);
SV*  newSVnv(double);
SV*  newSVpv(char*, int);
SV*  newSVsv(SV*);
```

To change the value of an *already−existing* SV, there are five routines:

```
void  sv_setiv(SV*, IV);
void  sv_setnv(SV*, double);
void  sv_setpvn(SV*, char*, int)
void  sv_setpv(SV*, char*);
void  sv_setsv(SV*, SV*);
```

Notice that you can choose to specify the length of the string to be assigned by using `sv_setpvn` 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.

All SV‘s that will contain strings should, but need not, 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*)
SvNV(SV*)
SvPV(SV*, STRLEN len)
```

which will automatically coerce the actual scalar type into an IV, 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 global variable `na`. Remember, however, that Perl allows arbitrary strings of data that may both contain NUL‘s and might not be terminated by a NUL.

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*, char*);
    void  sv_catpvn(SV*, char*, int);
    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 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.

If you know the name of a scalar variable, you can get a pointer to its SV by using the following:

```
    SV*  perl_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 `sv_undef`. Its address can be used whenever an `SV*` is needed.

There are also the two values `sv_yes` and `sv_no`, which contain Boolean TRUE and FALSE values, respectively. Like `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 `&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 `&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 the section on *Reference Counts and Mortality*).

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.

In general, though, it‘s best to use the `Sv*V` macros.

Working with AV‘s

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 SV‘s:

```
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 SV‘s can be destroyed, if so desired.

Once the AV has been created, the following operations are possible on AV‘s:

```
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`. 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 `key` elements. If `key` is less than the current 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*  perl_get_av("package::varname", FALSE);
```

This returns NULL if the variable does not exist.

Working with HV's

To create an HV, you use the following routine:

```
HV*  newHV();
```

Once the HV has been created, the following operations are possible on HV's:

```
SV**  hv_store(HV*, char* key, U32 klen, SV* val, U32 hash);
SV**  hv_fetch(HV*, 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 pre−computed 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*, char* key, U32 klen);
SV*   hv_delete(HV*, 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 a 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*  perl_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:

```
i = klen;
hash = 0;
s = key;
while (i--)
    hash = hash * 33 + *s++;
```

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 *API LISTING* later in this document 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 *API LISTING* later in this document 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)
```

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.
```

Blessed References and Class Objects

References are also used to support object−oriented programming. In the 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. The `stash` argument specifies which class the reference will belong to. See the section on *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, char* classname);
```

Copies integer or double into an SV whose reference is `rv`. SV is blessed if `classname` is non−null.

```
    SV* sv_setref_iv(SV* rv, char* classname, IV iv);
    SV* sv_setref_nv(SV* rv, 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, 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, char* classname, PV iv, int length);

        int sv_isa(SV* sv, char* name);
        int sv_isobject(SV* sv);
```

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*  perl_get_sv("package::varname", TRUE);
    AV*  perl_get_av("package::varname", TRUE);
    HV*  perl_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
                "Indentifier <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.

Reference Counts and Mortality

Perl uses an reference count−driven garbage collection mechanism. SV's, AV's, or HV's (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 xV's. 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 xV's 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.

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.

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, 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.

The mortal routines are not just for SV's — AV's and HV's can be made mortal by passing their address (type−casted to `SV*`) to the `sv_2mortal` or `sv_mortalcopy` routines.

Stashes and Globs

A "stash" is a hash that contains all of the different objects that are contained 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
File Handle
Directory Handle
Format
Subroutine
```

There is a single stash called "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 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(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*.

Double−Typed SV‘s

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 = perl_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`.

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.

Assigning Magic

Perl adds magic to an SV using the sv_magic function:

```
void sv_magic(SV* sv, SV* obj, int how, 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 set the `SVt_PVMG` flag for the `sv`. Perl then continues by adding it 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 `namlem` arguments are used to associate a string with the magic, typically the name of a variable. `namlem` is stored in the `mg_len` field and if `name` is non−null and `namlem` = 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 Table" section below. The `how` argument is also stored in the `mg_type` field.

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 "#", 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.

Magic Virtual Tables

The `mg_virtual` field in the `MAGIC` structure is a pointer to a `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 after 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 '\0') contains:

```
    { magic_get, magic_set, magic_len, 0, 0 }
```

Thus, when an SV is determined to be magical and of type '\0', 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_`.

The current kinds of Magic Virtual Tables are:

mg_type	MGVTBL	Type of magical
\0	vtbl_sv	Regexp???
A	vtbl_amagic	Operator Overloading
a	vtbl_amagicelem	Operator Overloading
c	0	Used in Operator Overloading
B	vtbl_bm	Boyer−Moore???
E	vtbl_env	%ENV hash
e	vtbl_envelem	%ENV hash element
g	vtbl_mglob	Regexp /g flag???
I	vtbl_isa	@ISA array
i	vtbl_isaelem	@ISA array element
L	0 (but sets RMAGICAL)	Perl Module/Debugger???
l	vtbl_dbline	Debugger?
o	vtbl_collxfrm	Locale transformation
P	vtbl_pack	Tied Array or Hash
p	vtbl_packelem	Tied Array or Hash element
q	vtbl_packelem	Tied Scalar or Handle
S	vtbl_sig	Signal Hash
s	vtbl_sigelem	Signal Hash element
t	vtbl_taint	Taintedness
U	vtbl_uvar	???
v	vtbl_vec	Vector
x	vtbl_substr	Substring???
y	vtbl_itervar	Shadow "foreach" iterator variable
*	vtbl_glob	GV???
#	vtbl_arylen	Array Length
.	vtbl_pos	$. scalar variable
~	None	Used by certain extensions

When an uppercase and lowercase letter both exist in the table, then the uppercase letter is 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.

The '~' magic type is defined specifically for use by extensions and will not be used by perl itself. Extensions can use ~ 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).

Note that because multiple extensions may be using ~ magic it is important for extensions to take extra care with it. Typically only using it on objects blessed into the same class as the extension is sufficient. It may also be appropriate to add an I32 'signature' at the top of the private data area and check that.

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, 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.

Subroutines

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 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 the macros to push IV's, doubles, strings, and SV pointers respectively:

```
PUSHi(IV)
PUSHn(double)
PUSHp(char*, I32)
PUSHs(SV*)
```

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 macros:

```
XPUSHi(IV)
XPUSHn(double)
XPUSHp(char*, I32)
XPUSHs(SV*)
```

These macros automatically adjust the stack for you, if needed. Thus, you do not need to call `EXTEND` to extend the stack.

For more information, consult *perlxs* and *perlxstut*.

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  perl_call_sv(SV*, I32);
I32  perl_call_pv(char*, I32);
I32  perl_call_method(char*, I32);
I32  perl_call_argv(char*, I32, register char**);
```

The routine most often used is `perl_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.

When using any of these routines (except `perl_call_argv`), the programmer must manipulate the Perl stack. These include the following macros and functions:

```
dSP
PUSHMARK()
PUTBACK
SPAGAIN
ENTER
SAVETMPS
FREETMPS
LEAVE
XPUSH*()
POP*()
```

For a detailed description of calling conventions from C to Perl, consult *perlcall*.

Memory Allocation

It is suggested that you use 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.

```
New(x, pointer, number, type);
Newc(x, pointer, number, type, cast);
Newz(x, pointer, number, type);
```

These three macros are used to initially allocate memory.

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.

```
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.

```
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).

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*.

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[pni]`.

Scratchpads

The question remains on when the SV‘s 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 SV‘s 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 OP‘s and *target*s is not 1−to−1. Different OP‘s in the compile tree of the unit can use the same target, if this would not conflict with the expected life of the temporary.

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 (sometime soon) **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.

Compiled code

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 reflect 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.

Examining the tree

If you have your perl compiled for debugging (usually done with `-D optimize=-g` on `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`.

Compile pass 1: check routines

The tree is created by the *pseudo−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 is 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***).

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.

Compile pass 2: context propagation

When a context for a part of compile tree is known, it is propagated down through the tree. Aat 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).

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 compilications 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.

API LISTING

This is a listing of functions, macros, flags, and variables that may be useful to extension writers or that may be found while reading other extensions.

AvFILL See `av_len`.

av_clear Clears an array, making it empty.

```
        void    av_clear _((AV* ar));
```

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*`.

```
        SV**    av_fetch _((AV* ar, I32 key, I32 lval));
```

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 `&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, otherwise it can be dereferenced to get the original `SV*`.

```
        SV**    av_store _((AV* ar, I32 key, SV* val));
```

av_undef
Undefines the array.

```
        void    av_undef _((AV* ar));
```

av_unshift
Unshift an SV onto the beginning of the array. The array will grow automatically to accommodate the addition.

```
        void    av_unshift _((AV* ar, I32 num));
```

CLASS
Variable which is setup by `xsubpp` to indicate the class name for a C++ XS constructor. This is always a `char*`. See `THIS` and *Using XS With C++ in perlxs*.

Copy
The XSUB−writer‘s interface to the C `memcpy` function. The `s` is the source, `d` is the destination, `n` is the number of items, and `t` is the type.

```
        (void) Copy( s, d, n, t );
```

croak
This is the XSUB−writer‘s interface to Perl‘s `die` function. Use this function the same way you use the C `printf` function. See `warn`.

CvSTASH
Returns the stash of the CV.

```
        HV * CvSTASH( SV* sv )
```

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 `DBsub`.

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 `DBsingle`. The sub name can be found by

```
        SvPV( GvSV( DBsub ), na )
```

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 `DBsingle`.

dMARK
Declare a stack marker variable, `mark`, for the XSUB. See `MARK` and `dORIGMARK`.

dORIGMARK
Saves the original stack mark for the XSUB. See `ORIGMARK`.

dowarn
The C variable which corresponds to Perl‘s `$^W` warning variable.

dSP Declares a stack pointer variable, `sp`, for the XSUB. See `SP`.

dXSARGS

Sets up stack and mark pointers for an XSUB, calling dSP and dMARK. This is usually handled automatically by `xsubpp`. Declares the `items` variable to indicate the number of items on the stack.

dXSI32 Sets up the `ix` variable for an XSUB which has aliases. This is usually handled automatically by `xsubpp`.

dXSI32 Sets up the `ix` variable for an XSUB which has aliases. This is usually handled automatically by `xsubpp`.

ENTER Opening bracket on a callback. See `LEAVE` and *perlcall*.

```
ENTER;
```

EXTEND Used to extend the argument stack for an XSUB‘s return values.

```
EXTEND( sp, int x );
```

FREETMPS

Closing bracket for temporaries on a callback. See `SAVETMPS` and *perlcall*.

```
FREETMPS;
```

G_ARRAY

Used to indicate array context. See `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*.

GIMME The XSUB−writer‘s equivalent to Perl‘s `wantarray`. Returns `G_SCALAR` or `G_ARRAY` for scalar or array context.

G_NOARGS

Indicates that no arguments are being sent to a callback. See *perlcall*.

G_SCALAR

Used to indicate scalar context. See `GIMME` and *perlcall*.

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 accessable 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 `perl_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, char* name, STRLEN len, I32 level)
```

gv_fetchmethod

Returns the glob which contains the subroutine to call to invoke the method on the `stash`. In fact in the presense of autoloading this may be the glob for "AUTOLOAD". In this case the corresponing variable `$AUTOLOAD` is already setup.

Note that if you want to keep this 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.

This function grants `"SUPER"` token as a prefix of the method name.

Has 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 `perl_call_sv` apply equally to `gv_fetchmethod`.

```
GV*     gv_fetchmethod _((HV* stash, char* name));
```

gv_stashpv

Returns a pointer to the stash for a specified package. 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 _((char* name, I32 create));
```

gv_stashsv

Returns a pointer to the stash for a specified package. See `gv_stashpv`.

```
HV*     gv_stashsv _((SV* sv, I32 create));
```

GvSV Return the SV from the GV.

HEf_SVKEY

This flag, used in the length slot of hash entries and magic structures, specifies the structure contains a `SV*` pointer where a `char*` pointer is to be expected. (For information only—not to be used).

HeHASH Returns the computed hash (type `U32`) stored in the hash entry.

```
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.

```
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.

```
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 `na`. 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.

```
HePV(HE* he, STRLEN len)
```

HeSVKEY

Returns the key as an `SV*`, or `Nullsv` if the hash entry does not contain an `SV*` key.

```
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.

```
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*.

```
HeSVKEY_set(HE* he, SV* sv)
```

HeVAL Returns the value slot (type SV*) stored in the hash entry.

```
HeVAL(HE* he)
```

hv_clear Clears a hash, making it empty.

```
void    hv_clear _((HV* tb));
```

hv_delayfree_ent

Releases a hash entry, such as while iterating though the hash, but delays actual freeing of key and value until the end of the current statement (or thereabouts) with sv_2mortal. See hv_iternext and hv_free_ent.

```
void    hv_delayfree_ent _((HV* hv, HE* entry));
```

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, char* key, U32 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 pre−computed 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, char* key, U32 klen));
```

hv_exists_ent

Returns a boolean indicating whether the specified hash key exists. hash can be a valid pre−computed 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 a SV*.

```
SV**    hv_fetch _((HV* tb, char* key, U32 klen, I32 lval));
```

hv_fetch_ent

Returns the hash entry which corresponds to the specified key in the hash. hash must be a valid pre−computed 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.

```
HE*     hv_fetch_ent  _((HV* tb, SV* key, I32 lval, U32 hash));
```

hv_free_ent

Releases a hash entry, such as while iterating though the hash. See `hv_iternext` and `hv_delayfree_ent`.

```
        void    hv_free_ent _((HV* hv, HE* entry));
```

hv_iterinit Prepares a starting point to traverse a hash table.

```
        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`.

```
        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_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));
```

HvNAME Returns the package name of a stash. See `SvSTASH`, `CvSTASH`.

```
        char *HvNAME (HV* stash)
```

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 pre−computed hash value; if it is zero then Perl will compute it. The return value will be null if the operation failed, otherwise it can be dereferenced to get the original `SV*`.

```
        SV**    hv_store _((HV* tb, char* key, U32 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 pre−computed 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 entry was stored in a tied hash. Otherwise the contents of the return value can be accessed using the `He???` macros described here.

```
        HE*     hv_store_ent  _((HV* tb, SV* key, SV* val, U32 hash));
```

hv_undef Undefines the hash.

```
        void    hv_undef _((HV* tb));
```

isALNUM
: Returns a boolean indicating whether the C `char` is an ascii alphanumeric character or digit.

```
int isALNUM (char c)
```

isALPHA
: Returns a boolean indicating whether the C `char` is an ascii alphabetic character.

```
int isALPHA (char c)
```

isDIGIT
: Returns a boolean indicating whether the C `char` is an ascii digit.

```
int isDIGIT (char c)
```

isLOWER
: Returns a boolean indicating whether the C `char` is a lowercase character.

```
int isLOWER (char c)
```

isSPACE
: Returns a boolean indicating whether the C `char` is whitespace.

```
int isSPACE (char c)
```

isUPPER
: Returns a boolean indicating whether the C `char` is an uppercase character.

```
int isUPPER (char c)
```

items
: Variable which is setup by `xsubpp` to indicate the number of items on the stack. See *Variable−length Parameter Lists in perlxs*.

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*.

LEAVE
: Closing bracket on a callback. See `ENTER` and *perlcall*.

```
LEAVE;
```

MARK
: Stack marker variable for the XSUB. See `dMARK`.

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 *, char *, STRLEN));
```

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_len
: Report on the SV‘s length. See `sv_magic`.

```
U32     mg_len _((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));
```

Move The XSUB−writer‘s interface to the C `memmove` function. The `s` is the source, `d` is the destination, `n` is the number of items, and `t` is the type.

```
        (void) Move( s, d, n, t );
```

na A variable which may be used with `SvPV` to tell Perl to calculate the string length.

New The XSUB−writer‘s interface to the C `malloc` function.

```
        void * New( x, void *ptr, int size, type )
```

Newc The XSUB−writer‘s interface to the C `malloc` function, with cast.

```
        void * Newc( x, void *ptr, int size, type, cast )
```

Newz The XSUB−writer‘s interface to the C `malloc` function. The allocated memory is zeroed with `memzero`.

```
        void * Newz( x, void *ptr, int size, type )
```

newAV Creates a new AV. The reference count is set to 1.

```
        AV*     newAV _((void));
```

newHV Creates a new HV. The reference count is set to 1.

```
        HV*     newHV _((void));
```

newRV_inc

Creates an RV wrapper for an SV. The reference count for the original SV is incremented.

```
        SV*     newRV_inc _((SV* ref));
```

For historical reasons, "newRV" is a synonym for "newRV_inc".

newRV_noinc

Creates an RV wrapper for an SV. The reference count for the original SV is **not** incremented.

```
        SV*     newRV_noinc _((SV* ref));
```

newSV Creates a new SV. The `len` parameter indicates the number of bytes of preallocated string space the SV should have. The reference count for the new SV is set to 1.

```
        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 double into it. The reference count for the SV is set to 1.

```
        SV*     newSVnv _((NV i));
```

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 then Perl will compute the length.

```
        SV*     newSVpv _((char* s, STRLEN len));
```

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, char* classname));
```

newSVsv
: Creates a new SV which is an exact duplicate of the original SV.

```
SV*     newSVsv _((SV* old));
```

newXS
: Used by `xsubpp` to hook up XSUBs as Perl subs.

newXSproto
: Used by `xsubpp` to hook up XSUBs as Perl subs. Adds Perl prototypes to the subs.

Nullav
: Null AV pointer.

Nullch
: Null character pointer.

Nullcv
: Null CV pointer.

Nullhv
: Null HV pointer.

Nullsv
: Null SV pointer.

ORIGMARK
: The original stack mark for the XSUB. See `dORIGMARK`.

perl_alloc
: Allocates a new Perl interpreter. See *perlembed*.

perl_call_argv
: Performs a callback to the specified Perl sub. See *perlcall*.

```
I32     perl_call_argv _((char* subname, I32 flags, char** argv));
```

perl_call_method
: Performs a callback to the specified Perl method. The blessed object must be on the stack. See *perlcall*.

```
I32     perl_call_method _((char* methname, I32 flags));
```

perl_call_pv
: Performs a callback to the specified Perl sub. See *perlcall*.

```
I32     perl_call_pv _((char* subname, I32 flags));
```

perl_call_sv
: Performs a callback to the Perl sub whose name is in the SV. See *perlcall*.

```
I32     perl_call_sv _((SV* sv, I32 flags));
```

perl_construct
: Initializes a new Perl interpreter. See *perlembed*.

perl_destruct
: Shuts down a Perl interpreter. See *perlembed*.

perl_eval_sv
: Tells Perl to `eval` the string in the SV.

```
I32     perl_eval_sv _((SV* sv, I32 flags));
```

perl_free
: Releases a Perl interpreter. See *perlembed*.

perl_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.

```
AV*     perl_get_av _((char* name, I32 create));
```

perl_get_cv
: Returns the CV of the specified Perl sub. 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.

```
CV*     perl_get_cv _((char* name, I32 create));
```

perl_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.

```
HV*     perl_get_hv _((char* name, I32 create));
```

perl_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.

```
SV*     perl_get_sv _((char* name, I32 create));
```

perl_parse
: Tells a Perl interpreter to parse a Perl script. See *perlembed*.

perl_require_pv
: Tells Perl to `require` a module.

```
void    perl_require_pv _((char* pv));
```

perl_run
: Tells a Perl interpreter to run. See *perlembed*.

POPi
: Pops an integer off the stack.

```
int POPi();
```

POPl
: Pops a long off the stack.

```
long POPl();
```

POPp
: Pops a string off the stack.

```
char * POPp();
```

POPn
: Pops a double off the stack.

```
double POPn();
```

POPs
: Pops an SV off the stack.

```
SV* POPs();
```

PUSHMARK
: Opening bracket for arguments on a callback. See `PUTBACK` and *perlcall*.

```
PUSHMARK(p)
```

PUSHi
: Push an integer onto the stack. The stack must have room for this element. See `XPUSHi`.

```
PUSHi(int d)
```

PUSHn
: Push a double onto the stack. The stack must have room for this element. See `XPUSHn`.

```
PUSHn(double d)
```

PUSHp
: Push a string onto the stack. The stack must have room for this element. The `len` indicates the length of the string. See `XPUSHp`.

```
PUSHp(char *c, int len )
```

PUSHs
Push an SV onto the stack. The stack must have room for this element. See `XPUSHs`.

```
PUSHs(sv)
```

PUTBACK
Closing bracket for XSUB arguments. This is usually handled by `xsubpp`. See `PUSHMARK` and *perlcall* for other uses.

```
PUTBACK;
```

Renew
The XSUB−writer‘s interface to the C `realloc` function.

```
void * Renew( void *ptr, int size, type )
```

Renewc
The XSUB−writer‘s interface to the C `realloc` function, with cast.

```
void * Renewc( void *ptr, int size, type, cast )
```

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*.

safefree
The XSUB−writer‘s interface to the C `free` function.

safemalloc
The XSUB−writer‘s interface to the C `malloc` function.

saferealloc
The XSUB−writer‘s interface to the C `realloc` function.

savepv
Copy a string to a safe spot. This does not use an SV.

```
char*   savepv _((char* sv));
```

savepvn
Copy a string to a safe spot. The `len` indicates number of bytes to copy. This does not use an SV.

```
char*   savepvn _((char* sv, I32 len));
```

SAVETMPS
Opening bracket for temporaries on a callback. See `FREETMPS` and *perlcall*.

```
SAVETMPS;
```

SP
Stack pointer. This is usually handled by `xsubpp`. See `dSP` and `SPAGAIN`.

SPAGAIN
Re−fetch the stack pointer. Used after a callback. See *perlcall*.

```
SPAGAIN;
```

ST
Used to access elements on the XSUB‘s stack.

```
SV* ST(int x)
```

strEQ
Test two strings to see if they are equal. Returns true or false.

```
int 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.

```
int 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.

```
int 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.

```
        int 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.

```
        int strLT( char *s1, char *s2 )
```

strNE
: Test two strings to see if they are different. Returns true or false.

```
        int 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.

```
        int strnEQ( char *s1, char *s2 )
```

strnNE
: Test two strings to see if they are different. The `len` parameter indicates the number of bytes to compare. Returns true or false.

```
        int strnNE( char *s1, char *s2, int len )
```

sv_2mortal
: Marks an SV as mortal. The SV will be destroyed when the current context ends.

```
        SV*     sv_2mortal _((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.

```
        void    sv_catpv _((SV* sv, char* ptr));
```

sv_catpvn
: Concatenates the string onto the end of the string which is in the SV. The `len` indicates number of bytes to copy.

```
        void    sv_catpvn _((SV* sv, char* ptr, STRLEN len));
```

sv_catsv
: Concatenates the string from SV `ssv` onto the end of the string in SV `dsv`.

```
        void    sv_catsv _((SV* dsv, SV* ssv));
```

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`.

```
        I32     sv_cmp _((SV* sv1, SV* sv2));
```

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`.

```
        I32     sv_cmp _((SV* sv1, SV* sv2));
```

SvCUR
: Returns the length of the string which is in the SV. See `SvLEN`.

```
        int SvCUR (SV* sv)
```

SvCUR_set
: Set the length of the string which is in the SV. See `SvCUR`.

```
        SvCUR_set (SV* sv, int val )
```

sv_dec Auto−decrement of the value in the SV.

```
void    sv_dec _((SV* sv));
```

sv_dec Auto−decrement of the value in the SV.

```
void    sv_dec _((SV* sv));
```

SvEND Returns a pointer to the last character in the string which is in the SV. See `SvCUR`. Access the character as

```
*SvEND(sv)
```

sv_eq Returns a boolean indicating whether the strings in the two SVs are identical.

```
I32     sv_eq _((SV* sv1, SV* sv2));
```

SvGROW

Expands the character buffer in the SV. Calls `sv_grow` to perform the expansion if necessary. Returns a pointer to the character buffer.

```
char * SvGROW( SV* sv, int len )
```

sv_grow Expands the character buffer in the SV. This will use `sv_unref` and will upgrade the SV to `SVt_PV`. Returns a pointer to the character buffer. Use `SvGROW`.

sv_inc Auto−increment of the value in the SV.

```
void    sv_inc _((SV* sv));
```

SvIOK Returns a boolean indicating whether the SV contains an integer.

```
int SvIOK (SV* SV)
```

SvIOK_off

Unsets the IV status of an SV.

```
SvIOK_off (SV* sv)
```

SvIOK_on

Tells an SV that it is an integer.

```
SvIOK_on (SV* sv)
```

SvIOK_only

Tells an SV that it is an integer and disables all other OK bits.

```
SvIOK_on (SV* sv)
```

SvIOK_only

Tells an SV that it is an integer and disables all other OK bits.

```
SvIOK_on (SV* sv)
```

SvIOKp Returns a boolean indicating whether the SV contains an integer. Checks the **private** setting. Use `SvIOK`.

```
int SvIOKp (SV* SV)
```

sv_isa Returns a boolean indicating whether the SV is blessed into the specified class. This does not know how to check for subtype, so it doesn‘t work in an inheritance relationship.

```
int     sv_isa _((SV* sv, char* name));
```

SvIV Returns the integer which is in the SV.

```
int SvIV (SV* sv)
```

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));
```

SvIVX Returns the integer which is stored in the SV.

```
int  SvIVX (SV* sv);
```

SvLEN Returns the size of the string buffer in the SV. See `SvCUR`.

```
int SvLEN (SV* sv)
```

sv_len Returns the length of the string in the SV. Use `SvCUR`.

```
STRLEN  sv_len _((SV* sv));
```

sv_len Returns the length of the string in the SV. Use `SvCUR`.

```
STRLEN  sv_len _((SV* sv));
```

sv_magic Adds magic to an SV.

```
void    sv_magic _((SV* sv, SV* obj, int how, char* name, I32 namlen)
```

sv_mortalcopy

Creates a new SV which is a copy of the original SV. The new SV is marked as mortal.

```
SV*     sv_mortalcopy _((SV* oldsv));
```

SvOK Returns a boolean indicating whether the value is an SV.

```
int SvOK (SV* sv)
```

sv_newmortal

Creates a new SV which is mortal. The reference count of the SV is set to 1.

```
SV*     sv_newmortal _((void));
```

sv_no This is the `false` SV. See `sv_yes`. Always refer to this as `&sv_no.`

SvNIOK Returns a boolean indicating whether the SV contains a number, integer or double.

```
int SvNIOK (SV* SV)
```

SvNIOK_off

Unsets the NV/IV status of an SV.

```
SvNIOK_off (SV* sv)
```

SvNIOKp Returns a boolean indicating whether the SV contains a number, integer or double. Checks the **private** setting. Use `SvNIOK`.

```
int SvNIOKp (SV* SV)
```

SvNOK Returns a boolean indicating whether the SV contains a double.

```
int SvNOK (SV* SV)
```

SvNOK_off

Unsets the NV status of an SV.

```
SvNOK_off (SV* sv)
```

SvNOK_on
: Tells an SV that it is a double.

```
        SvNOK_on (SV* sv)
```

SvNOK_only
: Tells an SV that it is a double and disables all other OK bits.

```
        SvNOK_on (SV* sv)
```

SvNOK_only
: Tells an SV that it is a double and disables all other OK bits.

```
        SvNOK_on (SV* sv)
```

SvNOKp
: Returns a boolean indicating whether the SV contains a double. Checks the **private** setting. Use `SvNOK`.

```
        int SvNOKp (SV* SV)
```

SvNV
: Returns the double which is stored in the SV.

```
        double SvNV (SV* sv);
```

SvNVX
: Returns the double which is stored in the SV.

```
        double SvNVX (SV* sv);
```

SvPOK
: Returns a boolean indicating whether the SV contains a character string.

```
        int SvPOK (SV* SV)
```

SvPOK_off
: Unsets the PV status of an SV.

```
        SvPOK_off (SV* sv)
```

SvPOK_on
: Tells an SV that it is a string.

```
        SvPOK_on (SV* sv)
```

SvPOK_only
: Tells an SV that it is a string and disables all other OK bits.

```
        SvPOK_on (SV* sv)
```

SvPOK_only
: Tells an SV that it is a string and disables all other OK bits.

```
        SvPOK_on (SV* sv)
```

SvPOKp
: Returns a boolean indicating whether the SV contains a character string. Checks the **private** setting. Use `SvPOK`.

```
        int SvPOKp (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. If `len` is `na` then Perl will handle the length on its own.

```
        char * SvPV (SV* sv, int len )
```

SvPVX
: Returns a pointer to the string in the SV. The SV must contain a string.

```
        char * SvPVX (SV* sv)
```

SvREFCNT
Returns the value of the object's reference count.

```
        int 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.

```
        void SvREFCNT_inc (SV* sv)
```

SvROK Tests if the SV is an RV.

```
        int SvROK (SV* sv)
```

SvROK_off
Unsets the RV status of an SV.

```
        SvROK_off (SV* sv)
```

SvROK_on
Tells an SV that it is an RV.

```
        SvROK_on (SV* sv)
```

SvRV Dereferences an RV to return the SV.

```
        SV*     SvRV (SV* sv);
```

sv_setiv Copies an integer into the given SV.

```
        void    sv_setiv _((SV* sv, IV num));
```

sv_setnv Copies a double into the given SV.

```
        void    sv_setnv _((SV* sv, double num));
```

sv_setpv Copies a string into an SV. The string must be null−terminated.

```
        void    sv_setpv _((SV* sv, char* ptr));
```

sv_setpvn
Copies a string into an SV. The `len` parameter indicates the number of bytes to be copied.

```
        void    sv_setpvn _((SV* sv, char* ptr, STRLEN len));
```

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 be returned and will have a reference count of 1.

```
        SV*     sv_setref_iv _((SV *rv, 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 be returned and will have a reference count of 1.

```
        SV*     sv_setref_nv _((SV *rv, char *classname, double 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 `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 be returned and will have a reference count of 1.

```
        SV*     sv_setref_pv _((SV *rv, char *classname, void* pv));
```

Do not use with integral 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_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 be returned and will have a reference count of 1.

```
        SV*     sv_setref_pvn _((SV *rv, char *classname, char* pv, I32 n));
```

Note that `sv_setref_pv` copies the pointer while this copies the string.

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.

```
        void    sv_setsv _((SV* dsv, SV* ssv));
```

SvSTASH

Returns the stash of the SV.

```
        HV * SvSTASH (SV* sv)
```

SVt_IV Integer 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`.

SVt_NV Double type flag for scalars. See `svtype`.

SvTRUE Returns a boolean indicating whether Perl would evaluate the SV as true or false, defined or undefined.

```
        int SvTRUE (SV* sv)
```

SvTYPE Returns the type of the SV. See `svtype`.

```
        svtype  SvTYPE (SV* sv)
```

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.

SvUPGRADE
: Used to upgrade an SV to a more complex form. Uses `sv_upgrade` to perform the upgrade if necessary. See `svtype`.

```
        bool    SvUPGRADE _((SV* sv, svtype mt));
```

sv_upgrade
: Upgrade an SV to a more complex form. Use `SvUPGRADE`. See `svtype`.

sv_undef
: This is the `undef` SV. Always refer to this as `&sv_undef`.

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`. See `SvROK_off`.

```
        void    sv_unref _((SV* sv));
```

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.

```
        void    sv_usepvn _((SV* sv, char* ptr, STRLEN len));
```

sv_yes
: This is the `true` SV. See `sv_no`. Always refer to this as `&sv_yes`.

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*.

toLOWER
: Converts the specified character to lowercase.

```
        int toLOWER (char c)
```

toUPPER
: Converts the specified character to uppercase.

```
        int toUPPER (char c)
```

warn
: This is the XSUB−writer‘s interface to Perl‘s `warn` function. Use this function the same way you use the C `printf` function. See `croak()`.

XPUSHi
: Push an integer onto the stack, extending the stack if necessary. See `PUSHi`.

```
        XPUSHi(int d)
```

XPUSHn
: Push a double onto the stack, extending the stack if necessary. See `PUSHn`.

```
        XPUSHn(double d)
```

XPUSHp
: Push a string onto the stack, extending the stack if necessary. The `len` indicates the length of the string. See `PUSHp`.

```
        XPUSHp(char *c, int len)
```

XPUSHs
: Push an SV onto the stack, extending the stack if necessary. See `PUSHs`.

```
        XPUSHs(sv)
```

XS Macro to declare an XSUB and its C parameter list. This is handled by `xsubpp`.

XSRETURN

Return from XSUB, indicating number of items on the stack. This is usually handled by `xsubpp`.

```
XSRETURN(int x);
```

XSRETURN_EMPTY

Return an empty list from an XSUB immediately.

```
XSRETURN_EMPTY;
```

XSRETURN_IV

Return an integer from an XSUB immediately. Uses `XST_mIV`.

```
XSRETURN_IV(IV v);
```

XSRETURN_NO

Return `&sv_no` from an XSUB immediately. Uses `XST_mNO`.

```
XSRETURN_NO;
```

XSRETURN_NV

Return an double from an XSUB immediately. Uses `XST_mNV`.

```
XSRETURN_NV(NV v);
```

XSRETURN_PV

Return a copy of a string from an XSUB immediately. Uses `XST_mPV`.

```
XSRETURN_PV(char *v);
```

XSRETURN_UNDEF

Return `&sv_undef` from an XSUB immediately. Uses `XST_mUNDEF`.

```
XSRETURN_UNDEF;
```

XSRETURN_YES

Return `&sv_yes` from an XSUB immediately. Uses `XST_mYES`.

```
XSRETURN_YES;
```

XST_mIV Place an integer into the specified position `i` on the stack. The value is stored in a new mortal SV.

```
XST_mIV( int i, IV v );
```

XST_mNV

Place a double into the specified position `i` on the stack. The value is stored in a new mortal SV.

```
XST_mNV( int i, NV v );
```

XST_mNO

Place `&sv_no` into the specified position `i` on the stack.

```
XST_mNO( int i );
```

XST_mPV

Place a copy of a string into the specified position `i` on the stack. The value is stored in a new mortal SV.

```
XST_mPV( int i, char *v );
```

XST_mUNDEF

Place `&sv_undef` into the specified position `i` on the stack.

```
XST_mUNDEF( int i );
```

XST_mYES

Place `&sv_yes` into the specified position `i` on the stack.

```
XST_mYES( int i );
```

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*.

Zero

The XSUB−writer's interface to the C `memzero` function. The `d` is the destination, `n` is the number of items, and `t` is the type.

```
(void) Zero( d, n, t );
```

EDITOR

Jeff Okamoto <***okamoto@corp.hp.com***

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, and Ulrich Pfeifer.

API Listing by Dean Roehrich <***roehrich@cray.com***.

DATE

Version 31.3: 1997/3/14

NAME

perlcall − Perl calling conventions from C

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*.

THE PERL_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 perl_call_sv(SV* sv, I32 flags) ;
    I32 perl_call_pv(char *subname, I32 flags) ;
    I32 perl_call_method(char *methname, I32 flags) ;
    I32 perl_call_argv(char *subname, I32 flags, register char **argv) ;
```

The key function is *perl_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 *perl_call_sv* to invoke the Perl subroutine.

All the *perl_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.

perl_call_sv

perl_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 perl_call_sv*, shows how you can make use of *perl_call_sv*.

perl_call_pv

The function, *perl_call_pv*, is similar to *perl_call_sv* except it expects its first parameter to be a C char* which identifies the Perl subroutine you want to call, e.g., `perl_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"`.

perl_call_method

The function *perl_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 perl_call_method* for an example of using *perl_call_method*.

perl_call_argv

perl_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 perl_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.

FLAG VALUES

The `flags` parameter in all the *perl_call_** functions is a bit mask which can consist of any combination of the symbols defined below, OR‘ed together.

G_SCALAR

Calls the Perl subroutine in a scalar context. This is the default context flag setting for all the *perl_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 *perl_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 *perl_call_** function. The section *Returning a list in a scalar context* shows an example of this behaviour.

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 an array 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 *perl_call_** function.

The value returned by the *perl_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.

G_DISCARD

By default, the *perl_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.

G_NOARGS

Whenever a Perl subroutine is called using one of the *perl_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 *perl_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`.

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 of events occurs, the process will terminate immediately. If though, 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 *perl_call_** function you need to check the `$@` variable as you would in a normal Perl script.

The value returned from the *perl_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 *perl_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 of using G_EVAL.

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 *perl_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.

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` macro. This will return `G_SCALAR` if you have been called in a scalar context and `G_ARRAY` if in an array context. An example of using the `GIMME` macro is shown in section *Using GIMME*.

KNOWN PROBLEMS

This section outlines all known problems that exist in the *perl_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 *perl_call_** will be wrong.

2. In Perl 5.000 and 5.001 there is a problem with using *perl_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) ;
        perl_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 later), or use the G_EVAL flag with *perl_call_** as shown below

```
    void
    Call_fred()
        CODE:
        PUSHMARK(sp) ;
        perl_call_pv("fred", G_EVAL|G_DISCARD|G_NOARGS) ;
        fprintf(stderr, "back in Call_fred\n") ;
```

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 *perl_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 *perl_call_pv* and *perl_call_sv*, you should always try to use *perl_call_sv*. See *Using perl_call_sv* for details.

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) ;
        perl_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 *perl_call_pv*.
4. As *perl_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 *perl_call_pv*. It will always be 0.

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 ;

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

    perl_call_pv("LeftString", G_DISCARD);
}
```

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`.
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 *perl_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 *perl_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. 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.

5. 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 the *XSUBs and the Argument Stack in perlguts* for details on how the XPUSH macros work.

6. Finally, *LeftString* can now be called via the *perl_call_pv* function.

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 = perl_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 *perl_call_pv*.

2. Because we are interested in what is returned from *Adder* we cannot specify G_DISCARD. This means that we will have to tidy up the Perl stack and dispose of any temporary values ourselves. 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, plus it will also dump the mortal SV‘s 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.

3. 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 *perl_call_pv* call.

   If you are making use of the Perl stack pointer in your code you must always refresh the your local copy using SPAGAIN whenever you make use of the *perl_call_** functions or any other Perl internal function.

4. Although only a single value was expected to be returned from *Adder*, it is still good practice to check the return code from *perl_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.

5. 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
   ```

6. 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.

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 = perl_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 array 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.

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 = perl_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 returned from the subroutine, *Adder* actually made it back to *call_AddSubScalar*.

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 = perl_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 *perl_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 *perl_call_pv*.

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 = perl_call_pv("Subtract", G_EVAL|G_SCALAR);

        SPAGAIN ;

        /* Check the eval first */
        if (SvTRUE(GvSV(errgv)))
        {
            printf ("Uh oh − %s\n", SvPV(GvSV(errgv), na)) ;
            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(GvSV(errgv)))
       {
           printf ("Uh oh − %s\n", SvPV(GvSV(errgv), na)) ;
           POPs ;
       }
   ```

   is the direct equivalent of this bit of Perl

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

   `errgv` is a perl global of type `GV *` that points to the symbol table entry containing the error.

`GvSV(errgv)` therefore refers to the C equivalent of `$@.`

3. Note that the stack is popped using `POPs` in the block where `SvTRUE(GvSV(errgv))` is true. This is necessary because whenever a *perl_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*.

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 *perl_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 *perl_call_pv* call in call_Subtract reads:

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

will preserve the error and restore reliable error handling.

Using perl_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) ;
    perl_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 *perl_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 *perl_call_sv* instead of *perl_call_pv*.

```
    void
    CallSubSV(name)
        SV *    name
        CODE:
        PUSHMARK(sp) ;
        perl_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, *perl_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) ;
        perl_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) ;
        perl_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`.

Using perl_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 *perl_call_argv* which will call *PrintList*.

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

static void
call_PrintList()
{
    dSP ;

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

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

Using perl_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;

    perl_call_method(method, G_DISCARD) ;

void
call_PrintID(class, method)
    char *  class
    char *  method
    CODE:
    PUSHMARK(sp);
    XPUSHs(sv_2mortal(newSVpv(class, 0))) ;
    PUTBACK;

    perl_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 *perl_call_method*.

Using GIMME

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

```
void
PrintContext()
    CODE:
    if (GIMME == G_SCALAR)
        printf ("Context is Scalar\n") ;
    else
        printf ("Context is Array\n") ;
```

and here is some Perl to test it

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

The output from that will be

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

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 *perl_call_** function or values returned via the stack) have been freed by one of these methods

- specifying the G_DISCARD flag with *perl_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 --> perl_call --> perl
                                                             |
    perl <-- XSUB <-- external library <-- perl_call <----+
```

After processing of the error using *perl_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 --> perl_call --> perl
                                                         |
                      event handler <-- perl_call --<--+
                      ...
                      event handler --> perl_call --> perl
                                                         |
                      event handler <-- perl_call --<--+
                      ...
                      event handler --> perl_call --> perl
                                                         |
                        event handler <-- perl_call --<--+
```

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

```
    event handler --> perl_call --> perl
```

for the 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.

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 */
        perl_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 */
        perl_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)
    charbuffer ;
    {
        ...
    }
```

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 pre−define 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 */
        perl_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 re−compiling. 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.

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 = perl_call_pv("AddSubtract", G_ARRAY);
```

```
            SPAGAIN ;
            sp -= count ;
            ax = (sp - 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 - 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.

SEE ALSO

perlxs, *perlguts*, *perlembed*

AUTHOR

Paul Marquess <***pmarquess@bfsec.bt.co.uk***

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.

DATE

Version 1.2, 16th Jan 1996

NAME

AnyDBM_File − provide framework for multiple DBMs

NDBM_File, ODBM_File, SDBM_File, GDBM_File − various DBM implementations

SYNOPSIS

```
    use AnyDBM_File;
```

DESCRIPTION

This module is a "pure virtual base class"—it has nothing of its own. It‘s just there to inherit from one of the various DBM packages. It prefers ndbm for compatibility reasons with Perl 4, then Berkeley DB (See *DB_File*), GDBM, SDBM (which is always there—it comes with Perl), and finally ODBM. This way old programs that used to use NDBM via `dbmopen()` can still do so, but new ones can reorder @ISA:

```
    @AnyDBM_File::ISA = qw(DB_File GDBM_File NDBM_File);
```

Note, however, that an explicit use overrides the specified order:

```
    use GDBM_File;
    @AnyDBM_File::ISA = qw(DB_File GDBM_File NDBM_File);
```

will only find GDBM_File.

Having multiple DBM implementations makes it trivial to copy database formats:

```
    use POSIX; use NDBM_File; use DB_File;
    tie %newhash,  'DB_File', $new_filename, O_CREAT|O_RDWR;
    tie %oldhash,  'NDBM_File', $old_filename, 1, 0;
    %newhash = %oldhash;
```

DBM Comparisons

Here‘s a partial table of features the different packages offer:

	odbm	ndbm	sdbm	gdbm	bsd-db
Linkage comes w/ perl	yes	yes	yes	yes	yes
Src comes w/ perl	no	no	yes	no	no
Comes w/ many unix os	yes	yes[0]	no	no	no
Builds ok on !unix	?	?	yes	yes	?
Code Size	?	?	small	big	big
Database Size	?	?	small	big?	ok[1]
Speed	?	?	slow	ok	fast
FTPable	no	no	yes	yes	yes
Easy to build	N/A	N/A	yes	yes	ok[2]
Size limits	1k	4k	1k[3]	none	none
Byte-order independent	no	no	no	no	yes
Licensing restrictions	?	?	no	yes	no

[0] on mixed universe machines, may be in the bsd compat library, which is often shunned.

[1] Can be trimmed if you compile for one access method.

[2] See *DB_File*. Requires symbolic links.

[3] By default, but can be redefined.

SEE ALSO

dbm(3), ndbm(3), DB_File(3)

NAME

AutoLoader − load functions only on demand

SYNOPSIS

```
    package FOOBAR;
    use Exporter;
    use AutoLoader;
    @ISA = qw(Exporter AutoLoader);
```

DESCRIPTION

This module tells its users that functions in the FOOBAR package are to be autoloaded from ***auto/$AUTOLOAD.al***. See *Autoloading in perlsub* and *AutoSplit*.

__END__

The module using the autoloader should have the special marker `__END__` prior to the actual subroutine declarations. All code that is before the marker will be loaded and compiled when the module is used. At the marker, perl will cease reading and parsing. See also the **AutoSplit** module, a utility that automatically splits a module into a collection of files for autoloading.

When a subroutine not yet in memory is called, the `AUTOLOAD` function attempts to locate it in a directory relative to the location of the module file itself. As an example, assume ***POSIX.pm*** is located in ***/usr/local/lib/perl5/POSIX.pm***. The autoloader will look for perl subroutines for this package in ***/usr/local/lib/perl5/auto/POSIX/*.al***. The `.al` file is named using the subroutine name, sans package.

Loading Stubs

The **AutoLoader** module provide a special `import()` method that will load the stubs (from ***autosplit.ix*** file) of the calling module. These stubs are needed to make inheritance work correctly for class modules.

Modules that inherit from **AutoLoader** should always ensure that they override the `AutoLoader−`import() method. If the module inherit from **Exporter** like shown in the *synopis* section this is already taken care of. For class methods an empty `import()` would do nicely:

```
  package MyClass;
  use AutoLoader;        # load stubs
  @ISA=qw(AutoLoader);
  sub import {}          # hide AutoLoader::import
```

You can also set up autoloading by importing the AUTOLOAD function instead of inheriting from **AutoLoader**:

```
  package MyClass;
  use AutoLoader;        # load stubs
  *AUTOLOAD = \&AutoLoader::AUTOLOAD;
```

Package Lexicals

Package lexicals declared with `my` in the main block of a package using the **AutoLoader** will not be visible to auto−loaded functions, due to the fact that the given scope ends at the `__END__` marker. A module using such variables as package globals will not work properly under the **AutoLoader**.

The `vars` pragma (see *vars in perlmod*) may be used in such situations as an alternative to explicitly qualifying all globals with the package namespace. Variables pre−declared with this pragma will be visible to any autoloaded routines (but will not be invisible outside the package, unfortunately).

AutoLoader vs. SelfLoader

The **AutoLoader** is a counterpart to the **SelfLoader** module. Both delay the loading of subroutines, but the **SelfLoader** accomplishes the goal via the `__DATA__` marker rather than `__END__`. While this avoids the use of a hierarchy of disk files and the associated open/close for each routine loaded, the **SelfLoader** suffers a disadvantage in the one−time parsing of the lines after `__DATA__`, after which routines are cached. **SelfLoader** can also handle multiple packages in a file.

AutoLoader only reads code as it is requested, and in many cases should be faster, but requires a machanism like **AutoSplit** be used to create the individual files. The **ExtUtils::MakeMaker** will invoke **AutoSplit** automatically if the **AutoLoader** is used in a module source file.

CAVEAT

On systems with restrictions on file name length, the file corresponding to a subroutine may have a shorter name that the routine itself. This can lead to conflicting file names. The *AutoSplit* package warns of these potential conflicts when used to split a module.

NAME

AutoSplit − split a package for autoloading

SYNOPSIS

```
perl −e 'use AutoSplit; autosplit_lib_modules(@ARGV)' ...
```

```
use AutoSplit; autosplit($file, $dir, $keep, $check, $modtime);
```

for perl versions 5.002 and later:

```
perl −MAutoSplit −e 'autosplit($ARGV[0], $ARGV[1], $k, $chk, $modtime)' ...
```

DESCRIPTION

This function will split up your program into files that the AutoLoader module can handle. It is used by both the standard perl libraries and by the MakeMaker utility, to automatically configure libraries for autoloading.

The `autosplit` interface splits the specified file into a hierarchy rooted at the directory `$dir.` It creates directories as needed to reflect class hierarchy, and creates the file ***autosplit.ix***. This file acts as both forward declaration of all package routines, and as timestamp for the last update of the hierarchy.

The remaining three arguments to `autosplit` govern other options to the autosplitter. If the third argument, *`$keep`*, is false, then any pre−existing `.al` files in the autoload directory are removed if they are no longer part of the module (obsoleted functions). The fourth argument, *`$check`*, instructs `autosplit` to check the module currently being split to ensure that it does include a `use` specification for the AutoLoader module, and skips the module if AutoLoader is not detected. Lastly, the *`$modtime`* argument specifies that `autosplit` is to check the modification time of the module against that of the `autosplit.ix` file, and only split the module if it is newer.

Typical use of AutoSplit in the perl MakeMaker utility is via the command−line with:

```
perl −e 'use AutoSplit; autosplit($ARGV[0], $ARGV[1], 0, 1, 1)'
```

Defined as a Make macro, it is invoked with file and directory arguments; `autosplit` will split the specified file into the specified directory and delete obsolete `.al` files, after checking first that the module does use the AutoLoader, and ensuring that the module is not already currently split in its current form (the modtime test).

The `autosplit_lib_modules` form is used in the building of perl. It takes as input a list of files (modules) that are assumed to reside in a directory **lib** relative to the current directory. Each file is sent to the autosplitter one at a time, to be split into the directory **lib/auto**.

In both usages of the autosplitter, only subroutines defined following the perl special marker *__END__* are split out into separate files. Some routines may be placed prior to this marker to force their immediate loading and parsing.

CAVEATS

Currently, `AutoSplit` cannot handle multiple package specifications within one file.

DIAGNOSTICS

`AutoSplit` will inform the user if it is necessary to create the top−level directory specified in the invocation. It is preferred that the script or installation process that invokes `AutoSplit` have created the full directory path ahead of time. This warning may indicate that the module is being split into an incorrect path.

`AutoSplit` will warn the user of all subroutines whose name causes potential file naming conflicts on machines with drastically limited (8 characters or less) file name length. Since the subroutine name is used as the file name, these warnings can aid in portability to such systems.

Warnings are issued and the file skipped if `AutoSplit` cannot locate either the *__END__* marker or a "package Name;"−style specification.

`AutoSplit` will also emit general diagnostics for inability to create directories or files.

NAME

Benchmark − benchmark running times of code

timethis − run a chunk of code several times

timethese − run several chunks of code several times

timeit − run a chunk of code and see how long it goes

SYNOPSIS

```
    timethis ($count, "code");

    timethese($count, {
        'Name1' => '...code1...',
        'Name2' => '...code2...',
    });

    $t = timeit($count, '...other code...')
    print "$count loops of other code took:",timestr($t),"\n";
```

DESCRIPTION

The Benchmark module encapsulates a number of routines to help you figure out how long it takes to execute some code.

Methods

new Returns the current time. Example:

```
    use Benchmark;
    $t0 = new Benchmark;
    # ... your code here ...
    $t1 = new Benchmark;
    $td = timediff($t1, $t0);
    print "the code took:",timestr($td),"\n";
```

debug Enables or disable debugging by setting the `$Benchmark::Debug` flag:

```
    debug Benchmark 1;
    $t = timeit(10, ' 5 ** $Global ');
    debug Benchmark 0;
```

Standard Exports

The following routines will be exported into your namespace if you use the Benchmark module:

timeit(COUNT, CODE)

Arguments: COUNT is the number of time to run the loop, and the second is the code to run. CODE may be a string containing the code, a reference to the function to run, or a reference to a hash containing keys which are names and values which are more CODE specs.

Side−effects: prints out noise to standard out.

Returns: a Benchmark object.

timethis
timethese
timediff
timestr

Optional Exports

The following routines will be exported into your namespace if you specifically ask that they be imported:

clearcache

clearallcache

disablecache

enablecache

NOTES

The data is stored as a list of values from the time and times functions:

```
      ($real, $user, $system, $children_user, $children_system)
```

in seconds for the whole loop (not divided by the number of rounds).

The timing is done using time(3) and times(3).

Code is executed in the caller‘s package.

Enable debugging by:

```
    $Benchmark::debug = 1;
```

The time of the null loop (a loop with the same number of rounds but empty loop body) is subtracted from the time of the real loop.

The null loop times are cached, the key being the number of rounds. The caching can be controlled using calls like these:

```
    clearcache($key);
    clearallcache();

    disablecache();
    enablecache();
```

INHERITANCE

Benchmark inherits from no other class, except of course for Exporter.

CAVEATS

The real time timing is done using time(2) and the granularity is therefore only one second.

Short tests may produce negative figures because perl can appear to take longer to execute the empty loop than a short test; try:

```
    timethis(100,'1');
```

The system time of the null loop might be slightly more than the system time of the loop with the actual code and therefore the difference might end up being < 0.

More documentation is needed :−(especially for styles and formats.

AUTHORS

Jarkko Hietaniemi <***Jarkko.Hietaniemi@hut.fi***>, Tim Bunce <***Tim.Bunce@ig.co.uk***>

MODIFICATION HISTORY

September 8th, 1994; by Tim Bunce.

NAME

carp − warn of errors (from perspective of caller)

croak − die of errors (from perspective of caller)

confess − die of errors with stack backtrace

SYNOPSIS

```
    use Carp;
    croak "We’re outta here!";
```

DESCRIPTION

The Carp routines are useful in your own modules because they act like `die()` or `warn()`, but report where the error was in the code they were called from. Thus if you have a routine `Foo()` that has a `carp()` in it, then the `carp()` will report the error as occurring where `Foo()` was called, not where `carp()` was called.

NAME

CPAN − query, download and build perl modules from CPAN sites

SYNOPSIS

Interactive mode:

```
perl −MCPAN −e shell;
```

Batch mode:

```
use CPAN;

autobundle, clean, install, make, recompile, test
```

DESCRIPTION

The CPAN module is designed to automate the make and install of perl modules and extensions. It includes some searching capabilities and knows how to use Net::FTP or LWP (or lynx or an external ftp client) to fetch the raw data from the net.

Modules are fetched from one or more of the mirrored CPAN (Comprehensive Perl Archive Network) sites and unpacked in a dedicated directory.

The CPAN module also supports the concept of named and versioned ‘bundles’ of modules. Bundles simplify the handling of sets of related modules. See BUNDLES below.

The package contains a session manager and a cache manager. There is no status retained between sessions. The session manager keeps track of what has been fetched, built and installed in the current session. The cache manager keeps track of the disk space occupied by the make processes and deletes excess space according to a simple FIFO mechanism.

All methods provided are accessible in a programmer style and in an interactive shell style.

Interactive Mode

The interactive mode is entered by running

```
    perl −MCPAN −e shell
```

which puts you into a readline interface. You will have most fun if you install Term::ReadKey and Term::ReadLine to enjoy both history and completion.

Once you are on the command line, type ‘h’ and the rest should be self−explanatory.

The most common uses of the interactive modes are

Searching for authors, bundles, distribution files and modules

There are corresponding one−letter commands `a`, `b`, `d`, and `m` for each of the four categories and another, `i` for any of the mentioned four. Each of the four entities is implemented as a class with slightly differing methods for displaying an object.

Arguments you pass to these commands are either strings matching exact the identification string of an object or regular expressions that are then matched case−insensitively against various attributes of the objects. The parser recognizes a regualar expression only if you enclose it between two slashes.

The principle is that the number of found objects influences how an item is displayed. If the search finds one item, we display the result of object−>as_string, but if we find more than one, we display each as object−>as_glimpse. E.g.

```
cpan> a ANDK
Author id = ANDK
    EMAIL        a.koenig@franz.ww.TU−Berlin.DE
    FULLNAME     Andreas König

cpan> a /andk/
```

```
    Author id = ANDK
        EMAIL        a.koenig@franz.ww.TU−Berlin.DE
        FULLNAME     Andreas König

    cpan> a /and.*rt/
    Author          ANDYD (Andy Dougherty)
    Author          MERLYN (Randal L. Schwartz)
```

make, test, install, clean modules or distributions

These commands do indeed exist just as written above. Each of them takes any number of arguments and investigates for each what it might be. Is it a distribution file (recognized by embedded slashes), this file is being processed. Is it a module, CPAN determines the distribution file where this module is included and processes that.

Any `make`, `test`, and `readme` are run unconditionally. A

```
  install <distribution_file>
```

also is run unconditionally. But for

```
  install <module>
```

CPAN checks if an install is actually needed for it and prints *Foo up to date* in case the module doesnt need to be updated.

CPAN also keeps track of what it has done within the current session and doesnt try to build a package a second time regardless if it succeeded or not. The `force` command takes as first argument the method to invoke (currently: make, test, or install) and executes the command from scratch.

Example:

```
    cpan> install OpenGL
    OpenGL is up to date.
    cpan> force install OpenGL
    Running make
    OpenGL−0.4/
    OpenGL−0.4/COPYRIGHT
    [...]
```

readme, look module or distribution

These two commands take only one argument, be it a module or a distribution file. `readme` displays the README of the associated distribution file. `Look` gets and untars (if not yet done) the distribution file, changes to the appropriate directory and opens a subshell process in that directory.

CPAN::Shell

The commands that are available in the shell interface are methods in the package CPAN::Shell. If you enter the shell command, all your input is split by the `Text::ParseWords::shellwords()` routine which acts like most shells do. The first word is being interpreted as the method to be called and the rest of the words are treated as arguments to this method.

autobundle

`autobundle` writes a bundle file into the `$CPAN::Config−>{cpan_home}/Bundle` directory. The file contains a list of all modules that are both available from CPAN and currently installed within @INC. The name of the bundle file is based on the current date and a counter.

recompile

`recompile()` is a very special command in that it takes no argument and runs the make/test/install cycle with brute force over all installed dynamically loadable extensions (aka XS modules) with ‘force’ in effect. Primary purpose of this command is to finish a network installation. Imagine, you have a common source tree for two different architectures. You decide to do a completely independent fresh installation. You start on one architecture with the help of a Bundle file produced earlier. CPAN installs the whole Bundle for you, but when you try to repeat the job on the second architecture, CPAN responds with a `"Foo up to date"` message for all modules. So you will be glad to run recompile in the second architecture and youre done.

Another popular use for `recompile` is to act as a rescue in case your perl breaks binary compatibility. If one of the modules that CPAN uses is in turn depending on binary compatibility (so you cannot run CPAN commands), then you should try the CPAN::Nox module for recovery.

Programmers interface

If you do not enter the shell, the available shell commands are both available as methods (`CPAN::Shell->install(...)`) and as functions in the calling package (`install(...)`). The programmers interface has beta status. Do not heavily rely on it, changes may still be necessary.

Cache Manager

Currently the cache manager only keeps track of the build directory (`$CPAN::Config−`{build_dir}). It is a simple FIFO mechanism that deletes complete directories below `build_dir` as soon as the size of all directories there gets bigger than `$CPAN::Config−`{build_cache} (in MB). The contents of this cache may be used for later re−installations that you intend to do manually, but will never be trusted by CPAN itself. This is due to the fact that the user might use these directories for building modules on different architectures.

There is another directory (`$CPAN::Config−`{keep_source_where}) where the original distribution files are kept. This directory is not covered by the cache manager and must be controlled by the user. If you choose to have the same directory as build_dir and as keep_source_where directory, then your sources will be deleted with the same fifo mechanism.

Bundles

A bundle is just a perl module in the namespace Bundle:: that does not define any functions or methods. It usually only contains documentation.

It starts like a perl module with a package declaration and a `$VERSION` variable. After that the pod section looks like any other pod with the only difference, that *one special pod section* exists starting with (verbatim):

```
=head1 CONTENTS
```

In this pod section each line obeys the format

```
Module_Name [Version_String] [− optional text]
```

The only required part is the first field, the name of a module (eg. Foo::Bar, ie. *not* the name of the distribution file). The rest of the line is optional. The comment part is delimited by a dash just as in the man page header.

The distribution of a bundle should follow the same convention as other distributions.

Bundles are treated specially in the CPAN package. If you say ‘install Bundle::Tkkit’ (assuming such a bundle exists), CPAN will install all the modules in the CONTENTS section of the pod. You can install your own Bundles locally by placing a conformant Bundle file somewhere into your @INC path. The `autobundle()` command which is available in the shell interface does that for you by including all currently installed modules in a snapshot bundle file.

There is a meaningless Bundle::Demo available on CPAN. Try to install it, it usually does no harm, just demonstrates what the Bundle interface looks like.

Prerequisites

If you have a local mirror of CPAN and can access all files with "file:" URLs, then you only need a perl better than perl5.003 to run this module. Otherwise Net::FTP is strongly recommended. LWP may be required for non−UNIX systems or if your nearest CPAN site is associated with an URL that is not `ftp:`.

If you have neither Net::FTP nor LWP, there is a fallback mechanism implemented for an external ftp command or for an external lynx command.

This module presumes that all packages on CPAN

- declare their `$VERSION` variable in an easy to parse manner. This prerequisite can hardly be relaxed because it consumes by far too much memory to load all packages into the running program just to determine the `$VERSION` variable . Currently all programs that are dealing with version use something like this

  ```
      perl −MExtUtils::MakeMaker −le \
          ’print MM−>parse_version($ARGV[0])’ filename
  ```

  If you are author of a package and wonder if your `$VERSION` can be parsed, please try the above method.

- come as compressed or gzipped tarfiles or as zip files and contain a Makefile.PL (well we try to handle a bit more, but without much enthusiasm).

Debugging

The debugging of this module is pretty difficult, because we have interferences of the software producing the indices on CPAN, of the mirroring process on CPAN, of packaging, of configuration, of synchronicity, and of bugs within CPAN.pm.

In interactive mode you can try "o debug" which will list options for debugging the various parts of the package. The output may not be very useful for you as it‘s just a byproduct of my own testing, but if you have an idea which part of the package may have a bug, it‘s sometimes worth to give it a try and send me more specific output. You should know that "o debug" has built−in completion support.

Floppy, Zip, and all that Jazz

CPAN.pm works nicely without network too. If you maintain machines that are not networked at all, you should consider working with file: URLs. Of course, you have to collect your modules somewhere first. So you might use CPAN.pm to put together all you need on a networked machine. Then copy the `$CPAN::Config−`{keep_source_where} (but not `$CPAN::Config−`{build_dir}) directory on a floppy. This floppy is kind of a personal CPAN. CPAN.pm on the non−networked machines works nicely with this floppy.

CONFIGURATION

When the CPAN module is installed a site wide configuration file is created as CPAN/Config.pm. The default values defined there can be overridden in another configuration file: CPAN/MyConfig.pm. You can store this file in `$HOME/.cpan/CPAN/MyConfig.pm` if you want, because `$HOME/.cpan` is added to the search path of the CPAN module before the `use()` or `require()` statements.

Currently the following keys in the hash reference `$CPAN::Config` are defined:

```
  build_cache        size of cache for directories to build modules
  build_dir          locally accessible directory to build modules
  index_expire       after how many days refetch index files
  cpan_home          local directory reserved for this package
  gzip               location of external program gzip
  inactivity_timeout breaks interactive Makefile.PLs after that
                     many seconds inactivity. Set to 0 to never break.
  inhibit_startup_message
                     if true, does not print the startup message
  keep_source        keep the source in a local directory?
  keep_source_where  where keep the source (if we do)
  make               location of external program make
  make_arg           arguments that should always be passed to ’make’
  make_install_arg   same as make_arg for ’make install’
  makepl_arg         arguments passed to ’perl Makefile.PL’
  pager              location of external program more (or any pager)
  tar                location of external program tar
```

```
  unzip              location of external program unzip
  urllist    arrayref to nearby CPAN sites (or equivalent locations)
```

You can set and query each of these options interactively in the cpan shell with the command set defined within the `o conf` command:

o conf <scalar option>

prints the current value of the *scalar option*

o conf <scalar option> <value>

Sets the value of the *scalar option* to *value*

o conf <list option>

prints the current value of the *list option* in MakeMaker‘s neatvalue format.

o conf <list option> [shift|pop]

shifts or pops the array in the *list option* variable

o conf <list option> [unshift|push|splice] <list>

works like the corresponding perl commands.

SECURITY

There‘s no strong security layer in CPAN.pm. CPAN.pm helps you to install foreign, unmasked, unsigned code on your machine. We compare to a checksum that comes from the net just as the distribution file itself. If somebody has managed to tamper with the distribution file, they may have as well tampered with the CHECKSUMS file. Future development will go towards strong authentification.

EXPORT

Most functions in package CPAN are exported per default. The reason for this is that the primary use is intended for the cpan shell or for oneliners.

BUGS

we should give coverage for _all_ of the CPAN and not just the __PAUSE__ part, right? In this discussion CPAN and PAUSE have become equal — but they are not. PAUSE is authors/ and modules/. CPAN is PAUSE plus the clpa/, doc/, misc/, ports/, src/, scripts/.

Future development should be directed towards a better intergration of the other parts.

AUTHOR

Andreas König <a.koenig@mind.de>

SEE ALSO

perl(1), CPAN::Nox(3)

NAME

getcwd − get pathname of current working directory

SYNOPSIS

```
use Cwd;
$dir = cwd;

use Cwd;
$dir = getcwd;

use Cwd;
$dir = fastgetcwd;

use Cwd 'chdir';
chdir "/tmp";
print $ENV{'PWD'};
```

DESCRIPTION

The `getcwd()` function re−implements the getcwd(3) (or getwd(3)) functions in Perl.

The `fastcwd()` function looks the same as `getcwd()`, but runs faster. It's also more dangerous because you might conceivably `chdir()` out of a directory that you can't `chdir()` back into.

The `cwd()` function looks the same as getcwd and fastgetcwd but is implemented using the most natural and safe form for the current architecture. For most systems it is identical to 'pwd' (but without the trailing line terminator). It is recommended that cwd (or another `*cwd()` function) is used in *all* code to ensure portability.

If you ask to override your `chdir()` built−in function, then your PWD environment variable will be kept up to date. (See *Overriding Builtin Functions*.) Note that it will only be kept up to date if all packages which use chdir import it from Cwd.

NAME

Class::Template − struct/member template builder

SYNOPSIS

```
use Class::Template;
struct(name => { key1 => type1, key2 => type2 });

package Myobj;
use Class::Template;
members Myobj { key1 => type1, key2 => type2 };
```

DESCRIPTION

This module uses perl5 classes to create nested data types.

EXAMPLES

- Example 1

```
use Class::Template;

struct( rusage => {
        ru_utime => timeval,
        ru_stime => timeval,
});

struct( timeval => [
        tv_secs  => '$',
        tv_usecs => '$',
]);

my $s = new rusage;
```

- Example 2

```
package OBJ;
use Class::Template;

members OBJ {
        'a'     => '$',
        'b'     => '$',
};

members OBJ2 {
        'd'     => '@',
        'c'     => '$',
};

package OBJ2; @ISA = (OBJ);

sub new {
        my $r = InitMembers( &OBJ::InitMembers() );
        bless $r;
}
```

NOTES

Use '%' if the member should point to an anonymous hash. Use '@' if the member should point to an anonymous array.

When using % and @ the method requires one argument for the key or index into the hash or array.

Prefix the %, @, or $ with '*' to indicate you want to retrieve pointers to the values rather than the values themselves.

NAME

Devel::SelfStubber − generate stubs for a SelfLoading module

SYNOPSIS

To generate just the stubs:

```
    use Devel::SelfStubber;
    Devel::SelfStubber−>stub(’MODULENAME’,’MY_LIB_DIR’);
```

or to generate the whole module with stubs inserted correctly

```
    use Devel::SelfStubber;
    $Devel::SelfStubber::JUST_STUBS=0;
    Devel::SelfStubber−>stub(’MODULENAME’,’MY_LIB_DIR’);
```

MODULENAME is the Perl module name, e.g. Devel::SelfStubber, NOT ‘Devel/SelfStubber’ or ‘Devel/SelfStubber.pm’.

MY_LIB_DIR defaults to ’.’ if not present.

DESCRIPTION

Devel::SelfStubber prints the stubs you need to put in the module before the __DATA__ token (or you can get it to print the entire module with stubs correctly placed). The stubs ensure that if a method is called, it will get loaded. They are needed specifically for inherited autoloaded methods.

This is best explained using the following example:

Assume four classes, A,B,C & D.

A is the root class, B is a subclass of A, C is a subclass of B, and D is another subclass of A.

```
                        A
                       / \
                      B   D
                     /
                    C
```

If D calls an autoloaded method ‘foo’ which is defined in class A, then the method is loaded into class A, then executed. If C then calls method ‘foo‘, and that method was reimplemented in class B, but set to be autoloaded, then the lookup mechanism never gets to the AUTOLOAD mechanism in B because it first finds the method already loaded in A, and so erroneously uses that. If the method foo had been stubbed in B, then the lookup mechanism would have found the stub, and correctly loaded and used the sub from B.

So, for classes and subclasses to have inheritance correctly work with autoloading, you need to ensure stubs are loaded.

The SelfLoader can load stubs automatically at module initialization with the statement `‘SelfLoader−>load_stubs()‘`;, but you may wish to avoid having the stub loading overhead associated with your initialization (though note that the SelfLoader::load_stubs method will be called sooner or later − at latest when the first sub is being autoloaded). In this case, you can put the sub stubs before the __DATA__ token. This can be done manually, but this module allows automatic generation of the stubs.

By default it just prints the stubs, but you can set the global `$Devel::SelfStubber::JUST_STUBS` to 0 and it will print out the entire module with the stubs positioned correctly.

At the very least, this is useful to see what the SelfLoader thinks are stubs − in order to ensure future versions of the SelfStubber remain in step with the SelfLoader, the SelfStubber actually uses the SelfLoader to determine which stubs are needed.

NAME

DirHandle − supply object methods for directory handles

SYNOPSIS

```
use DirHandle;
$d = new DirHandle ".";
if (defined $d) {
    while (defined($_ = $d−>read)) { something($_); }
    $d−>rewind;
    while (defined($_ = $d−>read)) { something_else($_); }
    undef $d;
}
```

DESCRIPTION

The `DirHandle` method provide an alternative interface to the `opendir()`, `closedir()`, `readdir()`, and `rewinddir()` functions.

The only objective benefit to using `DirHandle` is that it avoids namespace pollution by creating globs to hold directory handles.

NAME

English − use nice English (or awk) names for ugly punctuation variables

SYNOPSIS

```
use English;
...
if ($ERRNO =~ /denied/) { ... }
```

DESCRIPTION

This module provides aliases for the built−in variables whose names no one seems to like to read. Variables with side−effects which get triggered just by accessing them (like `$0`) will still be affected.

For those variables that have an **awk** version, both long and short English alternatives are provided. For example, the `$/` variable can be referred to either `$RS` or `$INPUT_RECORD_SEPARATOR` if you are using the English module.

See *perlvar* for a complete list of these.

NAME

Env − perl module that imports environment variables

SYNOPSIS

```
use Env;
use Env qw(PATH HOME TERM);
```

DESCRIPTION

Perl maintains environment variables in a pseudo−hash named %ENV. For when this access method is inconvenient, the Perl module `Env` allows environment variables to be treated as simple variables.

The `Env::import()` function ties environment variables with suitable names to global Perl variables with the same names. By default it does so with all existing environment variables (`keys %ENV`). If the import function receives arguments, it takes them to be a list of environment variables to tie; it‘s okay if they don‘t yet exist.

After an environment variable is tied, merely use it like a normal variable. You may access its value

```
@path = split(/:/, $PATH);
```

or modify it

```
$PATH .= ":.";
```

however you‘d like. To remove a tied environment variable from the environment, assign it the undefined value

```
undef $PATH;
```

AUTHOR

Chip Salzenberg <***chip@fin.uucp***>

NAME

Exporter − Implements default import method for modules

SYNOPSIS

In module ModuleName.pm:

```
package ModuleName;
require Exporter;
@ISA = qw(Exporter);

@EXPORT = qw(...);            # symbols to export by default
@EXPORT_OK = qw(...);         # symbols to export on request
%EXPORT_TAGS = tag => [...];  # define names for sets of symbols
```

In other files which wish to use ModuleName:

```
use ModuleName;               # import default symbols into my package

use ModuleName qw(...);       # import listed symbols into my package

use ModuleName ();            # do not import any symbols
```

DESCRIPTION

The Exporter module implements a default `import` method which many modules choose to inherit rather than implement their own.

Perl automatically calls the `import` method when processing a `use` statement for a module. Modules and `use` are documented in *perlfunc* and *perlmod*. Understanding the concept of modules and how the `use` statement operates is important to understanding the Exporter.

Selecting 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 symbol 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 informally indicate 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, since 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.

Other module design guidelines can be found in *perlmod*.

Specialised Import Lists

If the first entry in an import list begins with !, : or / then the list is treated as a series of specifications which either add to or delete from the list of names to import. They are processed left to right. Specifications are in the form:

```
    [!]name          This name only
    [!]:DEFAULT      All names in @EXPORT
    [!]:tag          All names in $EXPORT_TAGS{tag} anonymous list
    [!]/pattern/     All names in @EXPORT and @EXPORT_OK which match
```

A leading ! indicates that matching names should be deleted from the list of names to import. If the first specification is a deletion it is treated as though preceded by :DEFAULT. If you just want to import extra names in addition to the default set you will still need to include :DEFAULT explicitly.

e.g., Module.pm defines:

```
    @EXPORT      = qw(A1 A2 A3 A4 A5);
    @EXPORT_OK   = qw(B1 B2 B3 B4 B5);
    %EXPORT_TAGS = (T1 => [qw(A1 A2 B1 B2)], T2 => [qw(A1 A2 B3 B4)]);

    Note that you cannot use tags in @EXPORT or @EXPORT_OK.
    Names in EXPORT_TAGS must also appear in @EXPORT or @EXPORT_OK.
```

An application using Module can say something like:

```
    use Module qw(:DEFAULT :T2 !B3 A3);
```

Other examples include:

```
    use Socket qw(!/^[AP]F_/ !SOMAXCONN !SOL_SOCKET);
    use POSIX  qw(:errno_h :termios_h !TCSADRAIN !/^EXIT/);
```

Remember that most patterns (using //) will need to be anchored with a leading ^, e.g., `/^EXIT/` rather than `/EXIT/`.

You can say `BEGIN { $Exporter::Verbose=1 }` to see how the specifications are being processed and what is actually being imported into modules.

Module Version Checking

The Exporter module will convert an attempt to import a number from a module into a call to `$module_name->require_version($value).` This can be used to validate that the version of the module being used is greater than or equal to the required version.

The Exporter module supplies a default require_version method which checks the value of `$VERSION` in the exporting module.

Since the default require_version method treats the `$VERSION` number as a simple numeric value it will regard version 1.10 as lower than 1.9. For this reason it is strongly recommended that you use numbers with at least two decimal places, e.g., 1.09.

Managing Unknown Symbols

In some situations you may want to prevent certain symbols from being exported. Typically this applies to extensions which have functions or constants that may not exist on some systems.

The names of any symbols that cannot be exported should be listed in the `@EXPORT_FAIL` array.

If a module attempts to import any of these symbols the Exporter will will give the module an opportunity to handle the situation before generating an error. The Exporter will call an export_fail method with a list of the failed symbols:

```
  @failed_symbols = $module_name->export_fail(@failed_symbols);
```

If the export_fail method returns an empty list then no error is recorded and all the requested symbols are exported. If the returned list is not empty then an error is generated for each symbol and the export fails. The Exporter provides a default export_fail method which simply returns the list unchanged.

Uses for the export_fail method include giving better error messages for some symbols and performing lazy architectural checks (put more symbols into @EXPORT_FAIL by default and then take them out if someone actually tries to use them and an expensive check shows that they are usable on that platform).

Tag Handling Utility Functions

Since the symbols listed within %EXPORT_TAGS must also appear in either @EXPORT or @EXPORT_OK, two utility functions are provided which allow you to easily add tagged sets of symbols to

@EXPORT or @EXPORT_OK:

```
%EXPORT_TAGS = (foo => [qw(aa bb cc)], bar => [qw(aa cc dd)]);

Exporter::export_tags('foo');     # add aa, bb and cc to @EXPORT
Exporter::export_ok_tags('bar');  # add aa, cc and dd to @EXPORT_OK
```

Any names which are not tags are added to @EXPORT or @EXPORT_OK unchanged but will trigger a warning (with `-w`) to avoid misspelt tags names being silently added to @EXPORT or @EXPORT_OK. Future versions may make this a fatal error.

NAME

ExtUtils::Command − utilities to replace common UNIX commands in Makefiles etc.

SYNOPSYS

```
perl −MExtUtils::command −e cat files... > destination
perl −MExtUtils::command −e mv source... destination
perl −MExtUtils::command −e cp source... destination
perl −MExtUtils::command −e touch files...
perl −MExtUtils::command −e rm_f file...
perl −MExtUtils::command −e rm_rf directories...
perl −MExtUtils::command −e mkpath directories...
perl −MExtUtils::command −e eqtime source destination
perl −MExtUtils::command −e chmod mode files...
perl −MExtUtils::command −e test_f file
```

DESCRIPTION

The module is used in Win32 port to replace common UNIX commands. Most commands are wrapers on generic modules File::Path and File::Basename.

cat Concatenates all files menthion on command line to STDOUT.

eqtime src dst

Sets modified time of dst to that of src

rm_f files....

Removes directories − recursively (even if readonly)

rm_f files....

Removes files (even if readonly)

touch files ...

Makes files exist, with current timestamp

mv source... destination

Moves source to destination. Multiple sources are allowed if destination is an existing directory.

cp source... destination

Copies source to destination. Multiple sources are allowed if destination is an existing directory.

chmod mode files...

Sets UNIX like permissions ‘mode’ on all the files.

mkpath directory...

Creates directory, including any parent directories.

test_f file

Tests if a file exists

BUGS

eqtime does not work right on Win32 due to problems with `utime()` built−in command.

Should probably be Auto/Self loaded.

SEE ALSO

ExtUtils::MakeMaker, ExtUtils::MM_Unix, ExtUtils::MM_Win32

AUTHOR

Nick Ing−Simmons <***nick@ni−s.u−net.com***.

NAME

ExtUtils::Embed − Utilities for embedding Perl in C/C++ applications

SYNOPSIS

```
 perl −MExtUtils::Embed −e xsinit
 perl −MExtUtils::Embed −e ldopts
```

DESCRIPTION

ExtUtils::Embed provides utility functions for embedding a Perl interpreter and extensions in your C/C++ applications. Typically, an application **Makefile** will invoke ExtUtils::Embed functions while building your application.

@EXPORT

ExtUtils::Embed exports the following functions:

`xsinit()`, `ldopts()`, `ccopts()`, `perl_inc()`, `ccflags()`, `ccdlflags()`, `xsi_header()`, `xsi_protos()`, `xsi_body()`

FUNCTIONS

`xsinit()`

Generate C/C++ code for the XS initializer function.

When invoked as ‘`perl −MExtUtils::Embed −e xsinit −`‘ the following options are recognized:

−o <output filename> (Defaults to **perlxsi.c**)

−o STDOUT will print to STDOUT.

−std (Write code for extensions that are linked with the current Perl.)

Any additional arguments are expected to be names of modules to generate code for.

When invoked with parameters the following are accepted and optional:

`xsinit($filename,$std,[@modules])`

Where,

`$filename` is equivalent to the **−o** option.

`$std` is boolean, equivalent to the **−std** option.

[@modules] is an array ref, same as additional arguments mentioned above.

Examples

```
 perl −MExtUtils::Embed −e xsinit -- −o xsinit.c Socket
```

This will generate code with an **xs_init** function that glues the perl **Socket::bootstrap** function to the C **boot_Socket** function and writes it to a file named "xsinit.c".

Note that **DynaLoader** is a special case where it must call **boot_DynaLoader** directly.

```
 perl −MExtUtils::Embed −e xsinit
```

This will generate code for linking with **DynaLoader** and each static extension found in **`$Config{static_ext}`.** The code is written to the default file name **perlxsi.c**.

```
 perl −MExtUtils::Embed −e xsinit -- −o xsinit.c −std DBI DBD::Oracle
```

Here, code is written for all the currently linked extensions along with code for **DBI** and **DBD::Oracle**.

If you have a working **DynaLoader** then there is rarely any need to statically link in any other extensions.

`ldopts()`

Output arguments for linking the Perl library and extensions to your application.

When invoked as `‘perl −MExtUtils::Embed −e ldopts −‘` the following options are recognized:

−std

Output arguments for linking the Perl library and any extensions linked with the current Perl.

−I <path1:path2>

Search path for ModuleName.a archives. Default path is **@INC**. Library archives are expected to be found as **/some/path/auto/ModuleName/ModuleName.a** For example, when looking for **Socket.a** relative to a search path, we should find **auto/Socket/Socket.a**

When looking for **DBD::Oracle** relative to a search path, we should find **auto/DBD/Oracle/Oracle.a**

Keep in mind, you can always supply **/my/own/path/ModuleName.a** as an additional linker argument.

— <list of linker args>

Additional linker arguments to be considered.

Any additional arguments found before the **—** token are expected to be names of modules to generate code for.

When invoked with parameters the following are accepted and optional:

`ldopts($std,[@modules],[@link_args],$path)`

Where,

`$std` is boolean, equivalent to the **−std** option.

[@modules] is equivalent to additional arguments found before the **—** token.

[@link_args] is equivalent to arguments found after the **—** token.

`$path` is equivalent to the **−I** option.

In addition, when ldopts is called with parameters, it will return the argument string rather than print it to STDOUT.

Examples

```
perl −MExtUtils::Embed −e ldopts
```

This will print arguments for linking with **libperl.a**, **DynaLoader** and extensions found in **`$Config{static_ext}`**. This includes libraries found in **`$Config{libs}`** and the first ModuleName.a library for each extension that is found by searching **@INC** or the path specifed by the **−I** option. In addition, when ModuleName.a is found, additional linker arguments are picked up from the **extralibs.ld** file in the same directory.

```
perl −MExtUtils::Embed −e ldopts -- −std Socket
```

This will do the same as the above example, along with printing additional arguments for linking with the **Socket** extension.

```
perl −MExtUtils::Embed −e ldopts -- DynaLoader
```

This will print arguments for linking with just the **DynaLoader** extension and **libperl.a**.

```
perl −MExtUtils::Embed −e ldopts -- −std Msql -- −L/usr/msql/lib −lmsql
```

Any arguments after the second '—' token are additional linker arguments that will be examined for potential conflict. If there is no conflict, the additional arguments will be part of the output.

`perl_inc()`

For including perl header files this function simply prints:

```
 -I$Config{archlibexp}/CORE
```

So, rather than having to say:

```
 perl -MConfig -e 'print "-I$Config{archlibexp}/CORE"'
```

Just say:

```
 perl -MExtUtils::Embed -e perl_inc
```

`ccflags()`, `ccdlflags()`

These functions simply print `$Config{ccflags}` and `$Config{ccdlflags}`

`ccopts()`

This function combines `perl_inc()`, `ccflags()` and `ccdlflags()` into one.

`xsi_header()`

This function simply returns a string defining the same **EXTERN_C** macro as **perlmain.c** along with #including **perl.h** and **EXTERN.h**.

xsi_protos(@modules)

This function returns a string of **boot_$ModuleName** prototypes for each @modules.

xsi_body(@modules)

This function returns a string of calls to **newXS()** that glue the module **bootstrap** function to **boot_ModuleName** for each @modules.

xsinit() uses the xsi_* functions to generate most of it's code.

EXAMPLES

For examples on how to use **ExtUtils::Embed** for building C/C++ applications with embedded perl, see the eg/ directory and *perlembed*.

SEE ALSO

perlembed

AUTHOR

Doug MacEachern <***dougm@osf.org***>

Based on ideas from Tim Bunce <***Tim.Bunce@ig.co.uk***> and **minimod.pl** by Andreas Koenig <***k@anna.in−berlin.de***> and Tim Bunce.

NAME

ExtUtils::Install − install files from here to there

SYNOPSIS

use ExtUtils::Install;

install(`$hashref,$verbose,$nonono);`

uninstall(`$packlistfile,$verbose,$nonono);`

pm_to_blib(`$hashref);`

DESCRIPTION

Both `install()` and `uninstall()` are specific to the way ExtUtils::MakeMaker handles the installation and deinstallation of perl modules. They are not designed as general purpose tools.

`install()` takes three arguments. A reference to a hash, a verbose switch and a don‘t−really−do−it switch. The hash ref contains a mapping of directories: each key/value pair is a combination of directories to be copied. Key is a directory to copy from, value is a directory to copy to. The whole tree below the "from" directory will be copied preserving timestamps and permissions.

There are two keys with a special meaning in the hash: "read" and "write". After the copying is done, install will write the list of target files to the file named by `$hashref−>{write}.` If there is another file named by `$hashref−>{read},` the contents of this file will be merged into the written file. The read and the written file may be identical, but on AFS it is quite likely, people are installing to a different directory than the one where the files later appear.

`uninstall()` takes as first argument a file containing filenames to be unlinked. The second argument is a verbose switch, the third is a no−don‘t−really−do−it−now switch.

`pm_to_blib()` takes a hashref as the first argument and copies all keys of the hash to the corresponding values efficiently. Filenames with the extension pm are autosplit. Second argument is the autosplit directory.

NAME

ExtUtils::Liblist − determine libraries to use and how to use them

SYNOPSIS

```
require ExtUtils::Liblist;
```

```
ExtUtils::Liblist::ext($potential_libs, $Verbose);
```

DESCRIPTION

This utility takes a list of libraries in the form `−llib1 −llib2 −llib3` and prints out lines suitable for inclusion in an extension Makefile. Extra library paths may be included with the form `−L/another/path` this will affect the searches for all subsequent libraries.

It returns an array of four scalar values: EXTRALIBS, BSLOADLIBS, LDLOADLIBS, and LD_RUN_PATH.

Dependent libraries can be linked in one of three ways:

- For static extensions

  by the ld command when the perl binary is linked with the extension library. See EXTRALIBS below.

- For dynamic extensions

  by the ld command when the shared object is built/linked. See LDLOADLIBS below.

- For dynamic extensions

  by the DynaLoader when the shared object is loaded. See BSLOADLIBS below.

EXTRALIBS

List of libraries that need to be linked with when linking a perl binary which includes this extension Only those libraries that actually exist are included. These are written to a file and used when linking perl.

LDLOADLIBS and LD_RUN_PATH

List of those libraries which can or must be linked into the shared library when created using ld. These may be static or dynamic libraries. LD_RUN_PATH is a colon separated list of the directories in LDLOADLIBS. It is passed as an environment variable to the process that links the shared library.

BSLOADLIBS

List of those libraries that are needed but can be linked in dynamically at run time on this platform. SunOS/Solaris does not need this because ld records the information (from LDLOADLIBS) into the object file. This list is used to create a .bs (bootstrap) file.

PORTABILITY

This module deals with a lot of system dependencies and has quite a few architecture specific **if**s in the code.

VMS implementation

The version of `ext()` which is executed under VMS differs from the Unix−OS/2 version in several respects:

- Input library and path specifications are accepted with or without the `−l` and `−L` prefices used by Unix linkers. If neither prefix is present, a token is considered a directory to search if it is in fact a directory, and a library to search for otherwise. Authors who wish their extensions to be portable to Unix or OS/2 should use the Unix prefixes, since the Unix−OS/2 version of `ext()` requires them.
- Wherever possible, shareable images are preferred to object libraries, and object libraries to plain object files. In accordance with VMS naming conventions, `ext()` looks for files named *lib*shr and *lib*rtl; it also looks for *lib*lib and lib*lib* to accomodate Unix conventions used in some ported software.
- For each library that is found, an appropriate directive for a linker options file is generated. The return values are space−separated strings of these directives, rather than elements used on the linker command line.

- LDLOADLIBS and EXTRALIBS are always identical under VMS, and BSLOADLIBS and LD_RIN_PATH are always empty.

In addition, an attempt is made to recognize several common Unix library names, and filter them out or convert them to their VMS equivalents, as appropriate.

In general, the VMS version of `ext()` should properly handle input from extensions originally designed for a Unix or VMS environment. If you encounter problems, or discover cases where the search could be improved, please let us know.

SEE ALSO

ExtUtils::MakeMaker

NAME

ExtUtils::MM_OS2 − methods to override UN*X behaviour in ExtUtils::MakeMaker

SYNOPSIS

```
 use ExtUtils::MM_OS2; # Done internally by ExtUtils::MakeMaker if needed
```

DESCRIPTION

See ExtUtils::MM_Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

NAME

ExtUtils::MM_Unix − methods used by ExtUtils::MakeMaker

SYNOPSIS

```
require ExtUtils::MM_Unix;
```

DESCRIPTION

The methods provided by this package are designed to be used in conjunction with ExtUtils::MakeMaker. When MakeMaker writes a Makefile, it creates one or more objects that inherit their methods from a package `MM`. MM itself doesn't provide any methods, but it ISA ExtUtils::MM_Unix class. The inheritance tree of MM lets operating specific packages take the responsibility for all the methods provided by MM_Unix. We are trying to reduce the number of the necessary overrides by defining rather primitive operations within ExtUtils::MM_Unix.

If you are going to write a platform specific MM package, please try to limit the necessary overrides to primitive methods, and if it is not possible to do so, let's work out how to achieve that gain.

If you are overriding any of these methods in your Makefile.PL (in the MY class), please report that to the makemaker mailing list. We are trying to minimize the necessary method overrides and switch to data driven Makefile.PLs wherever possible. In the long run less methods will be overridable via the MY class.

METHODS

The following description of methods is still under development. Please refer to the code for not suitably documented sections and complain loudly to the makemaker mailing list.

Not all of the methods below are overridable in a Makefile.PL. Overridable methods are marked as (o). All methods are overridable by a platform specific MM_*.pm file (See *ExtUtils::MM_VMS*) and *ExtUtils::MM_OS2*).

Preloaded methods

canonpath

No physical check on the filesystem, but a logical cleanup of a path. On UNIX eliminated successive slashes and successive "/.".

catdir

Concatenate two or more directory names to form a complete path ending with a directory. But remove the trailing slash from the resulting string, because it doesn't look good, isn't necessary and confuses OS2. Of course, if this is the root directory, don't cut off the trailing slash :−)

catfile

Concatenate one or more directory names and a filename to form a complete path ending with a filename

curdir

Returns a string representing of the current directory. "." on UNIX.

rootdir

Returns a string representing of the root directory. "/" on UNIX.

updir

Returns a string representing of the parent directory. ".." on UNIX.

SelfLoaded methods

c_o (o)

Defines the suffix rules to compile different flavors of C files to object files.

cflags (o)

Does very much the same as the cflags script in the perl distribution. It doesn't return the whole compiler command line, but initializes all of its parts. The const_cccmd method then actually returns the definition of the CCCMD macro which uses these parts.

clean (o)

Defines the clean target.

const_cccmd (o)

Returns the full compiler call for C programs and stores the definition in CONST_CCCMD.

const_config (o)

Defines a couple of constants in the Makefile that are imported from %Config.

const_loadlibs (o)

Defines EXTRALIBS, LDLOADLIBS, BSLOADLIBS, LD_RUN_PATH. See *ExtUtils::Liblist* for details.

constants (o)

Initializes lots of constants and .SUFFIXES and .PHONY

depend (o)

Same as macro for the depend attribute.

dir_target (o)

Takes an array of directories that need to exist and returns a Makefile entry for a .exists file in these directories. Returns nothing, if the entry has already been processed. We‘re helpless though, if the same directory comes as `$(FOO)` _and_ as "bar". Both of them get an entry, that‘s why we use "::".

dist (o)

Defines a lot of macros for distribution support.

dist_basics (o)

Defines the targets distclean, distcheck, skipcheck, manifest.

dist_ci (o)

Defines a check in target for RCS.

dist_core (o)

Defeines the targets dist, tardist, zipdist, uutardist, shdist

dist_dir (o)

Defines the scratch directory target that will hold the distribution before tar−ing (or shar−ing).

dist_test (o)

Defines a target that produces the distribution in the scratchdirectory, and runs ‘perl Makefile.PL; make ;make test’ in that subdirectory.

dlsyms (o)

Used by AIX and VMS to define DL_FUNCS and DL_VARS and write the *.exp files.

dynamic (o)

Defines the dynamic target.

dynamic_bs (o)

Defines targets for bootstrap files.

dynamic_lib (o)

Defines how to produce the *.so (or equivalent) files.

exescan

Deprecated method. Use libscan instead.

extliblist

Called by init_others, and calls ext ExtUtils::Liblist. See *ExtUtils::Liblist* for details.

file_name_is_absolute

Takes as argument a path and returns true, if it is an absolute path.

find_perl

Finds the executables PERL and FULLPERL

Methods to actually produce chunks of text for the Makefile

The methods here are called for each MakeMaker object in the order specified by @ExtUtils::MakeMaker::MM_Sections.

force (o)

Just writes FORCE:

guess_name

Guess the name of this package by examining the working directory's name. MakeMaker calls this only if the developer has not supplied a NAME attribute.

has_link_code

Returns true if C, XS, MYEXTLIB or similar objects exist within this object that need a compiler. Does not descend into subdirectories as `needs_linking()` does.

init_dirscan

Initializes DIR, XS, PM, C, O_FILES, H, PL_FILES, MAN*PODS, EXE_FILES.

init_main

Initializes NAME, FULLEXT, BASEEXT, PARENT_NAME, DLBASE, PERL_SRC, PERL_LIB, PERL_ARCHLIB, PERL_INC, INSTALLDIRS, INST_*, INSTALL*, PREFIX, CONFIG, AR, AR_STATIC_ARGS, LD, OBJ_EXT, LIB_EXT, MAP_TARGET, LIBPERL_A, VERSION_FROM, VERSION, DISTNAME, VERSION_SYM.

init_others

Initializes EXTRALIBS, BSLOADLIBS, LDLOADLIBS, LIBS, LD_RUN_PATH, OBJECT, BOOTDEP, PERLMAINCC, LDFROM, LINKTYPE, NOOP, FIRST_MAKEFILE, MAKEFILE, NOECHO, RM_F, RM_RF, TEST_F, TOUCH, CP, MV, CHMOD, UMASK_NULL

install (o)

Defines the install target.

installbin (o)

Defines targets to install EXE_FILES.

libscan (o)

Takes a path to a file that is found by init_dirscan and returns false if we don't want to include this file in the library. Mainly used to exclude RCS, CVS, and SCCS directories from installation.

linkext (o)

Defines the linkext target which in turn defines the LINKTYPE.

lsdir

Takes as arguments a directory name and a regular expression. Returns all entries in the directory that match the regular expression.

macro (o)

Simple subroutine to insert the macros defined by the macro attribute into the Makefile.

makeaperl (o)

Called by staticmake. Defines how to write the Makefile to produce a static new perl.

By default the Makefile produced includes all the static extensions in the perl library. (Purified versions of library files, e.g., DynaLoader_pure_p1_c0_032.a are automatically ignored to avoid link errors.)

makefile (o)

Defines how to rewrite the Makefile.

manifypods (o)

Defines targets and routines to translate the pods into manpages and put them into the INST_* directories.

maybe_command

Returns true, if the argument is likely to be a command.

maybe_command_in_dirs

method under development. Not yet used. Ask Ilya :−)

needs_linking (o)

Does this module need linking? Looks into subdirectory objects (see also `has_link_code()`)

nicetext

misnamed method (will have to be changed). The MM_Unix method just returns the argument without further processing.

On VMS used to insure that colons marking targets are preceded by space − most Unix Makes don‘t need this, but it‘s necessary under VMS to distinguish the target delimiter from a colon appearing as part of a filespec.

parse_version

parse a file and return what you think is `$VERSION` in this file set to

pasthru (o)

Defines the string that is passed to recursive make calls in subdirectories.

path

Takes no argument, returns the environment variable PATH as an array.

perl_script

Takes one argument, a file name, and returns the file name, if the argument is likely to be a perl script. On MM_Unix this is true for any ordinary, readable file.

perldepend (o)

Defines the dependency from all *.h files that come with the perl distribution.

pm_to_blib

Defines target that copies all files in the hash PM to their destination and autosplits them. See *ExtUtils::Install/DESCRIPTION*

post_constants (o)

Returns an empty string per default. Dedicated to overrides from within Makefile.PL after all constants have been defined.

post_initialize (o)

Returns an empty string per default. Used in Makefile.PLs to add some chunk of text to the Makefile after the object is initialized.

postamble (o)

Returns an empty string. Can be used in Makefile.PLs to write some text to the Makefile at the end.

prefixify

Check a path variable in `$self` from %Config, if it contains a prefix, and replace it with another one.

Takes as arguments an attribute name, a search prefix and a replacement prefix. Changes the attribute in the object.

processPL (o)

Defines targets to run *.PL files.

realclean (o)

Defines the realclean target.

replace_manpage_separator

Takes the name of a package, which may be a nested package, in the form Foo/Bar and replaces the slash with `::`. Returns the replacement.

static (o)

Defines the static target.

static_lib (o)

Defines how to produce the *.a (or equivalent) files.

staticmake (o)

Calls makeaperl.

subdir_x (o)

Helper subroutine for subdirs

subdirs (o)

Defines targets to process subdirectories.

test (o)

Defines the test targets.

test_via_harness (o)

Helper method to write the test targets

test_via_script (o)

Other helper method for test.

tool_autosplit (o)

Defines a simple perl call that runs autosplit. May be deprecated by pm_to_blib soon.

tools_other (o)

Defines SHELL, LD, TOUCH, CP, MV, RM_F, RM_RF, CHMOD, UMASK_NULL in the Makefile. Also defines the perl programs MKPATH, WARN_IF_OLD_PACKLIST, MOD_INSTALL. DOC_INSTALL, and UNINSTALL.

tool_xsubpp (o)

Determines typemaps, xsubpp version, prototype behaviour.

top_targets (o)

Defines the targets all, subdirs, config, and O_FILES writedoc

Obsolete, depecated method. Not used since Version 5.21.

xs_c (o)

Defines the suffix rules to compile XS files to C.

xs_o (o)

Defines suffix rules to go from XS to object files directly. This is only intended for broken make implementations.

perl_archive

This is internal method that returns path to libperl.a equivalent to be linked to dynamic extensions. UNIX does not have one but OS2 and Win32 do.

export_list

This is internal method that returns name of a file that is passed to linker to define symbols to be exported. UNIX does not have one but OS2 and Win32 do.

SEE ALSO

ExtUtils::MakeMaker

NAME

ExtUtils::MM_VMS − methods to override UN*X behaviour in ExtUtils::MakeMaker

SYNOPSIS

```
 use ExtUtils::MM_VMS; # Done internally by ExtUtils::MakeMaker if needed
```

DESCRIPTION

See ExtUtils::MM_Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

Methods always loaded

eliminate_macros

Expands MM[KS]/Make macros in a text string, using the contents of identically named elements of `%$self,` and returns the result as a file specification in Unix syntax.

fixpath

Catchall routine to clean up problem MM[SK]/Make macros. Expands macros in any directory specification, in order to avoid juxtaposing two VMS−syntax directories when MM[SK] is run. Also expands expressions which are all macro, so that we can tell how long the expansion is, and avoid overrunning DCL‘s command buffer when MM[KS] is running.

If optional second argument has a TRUE value, then the return string is a VMS−syntax directory specification, otherwise it is a VMS−syntax file specification.

catdir

Concatenates a list of file specifications, and returns the result as a VMS−syntax directory specification.

catfile

Concatenates a list of file specifications, and returns the result as a VMS−syntax directory specification.

wraplist

Converts a list into a string wrapped at approximately 80 columns.

curdir (override)

Returns a string representing of the current directory.

rootdir (override)

Returns a string representing of the root directory.

updir (override)

Returns a string representing of the parent directory.

SelfLoaded methods

Those methods which override default MM_Unix methods are marked "(override)", while methods unique to MM_VMS are marked "(specific)". For overridden methods, documentation is limited to an explanation of why this method overrides the MM_Unix method; see the ExtUtils::MM_Unix documentation for more details.

guess_name (override)

Try to determine name of extension being built. We begin with the name of the current directory. Since VMS filenames are case−insensitive, however, we look for a ***.pm*** file whose name matches that of the current directory (presumably the ‘main’ ***.pm*** file for this extension), and try to find a `package` statement from which to obtain the Mixed::Case package name.

find_perl (override)

Use VMS file specification syntax and CLI commands to find and invoke Perl images.

path (override)

Translate logical name DCL`$PATH` as a searchlist, rather than trying to `split` string value of `$ENV{'PATH'}.`

maybe_command (override)

Follows VMS naming conventions for executable files. If the name passed in doesn't exactly match an executable file, appends ***.Exe*** (or equivalent) to check for executable image, and ***.Com*** to check for DCL procedure. If this fails, checks directories in DCL`$PATH` and finally ***Sys`$System:`*** for an executable file having the name specified, with or without the ***.Exe***−equivalent suffix.

maybe_command_in_dirs (override)

Uses DCL argument quoting on test command line.

perl_script (override)

If name passed in doesn't specify a readable file, appends ***.com*** or ***.pl*** and tries again, since it's customary to have file types on all files under VMS.

file_name_is_absolute (override)

Checks for VMS directory spec as well as Unix separators.

replace_manpage_separator

Use as separator a character which is legal in a VMS−syntax file name.

init_others (override)

Provide VMS−specific forms of various utility commands, then hand off to the default MM_Unix method.

constants (override)

Fixes up numerous file and directory macros to insure VMS syntax regardless of input syntax. Also adds a few VMS−specific macros and makes lists of files comma−separated.

cflags (override)

Bypass shell script and produce qualifiers for CC directly (but warn user if a shell script for this extension exists). Fold multiple /Defines into one, since some C compilers pay attention to only one instance of this qualifier on the command line.

const_cccmd (override)

Adds directives to point C preprocessor to the right place when handling #include <sys/foo.h> directives. Also constructs CC command line a bit differently than MM_Unix method.

pm_to_blib (override)

DCL *still* accepts a maximum of 255 characters on a command line, so we write the (potentially) long list of file names to a temp file, then persuade Perl to read it instead of the command line to find args.

tool_autosplit (override)

Use VMS−style quoting on command line.

tool_sxubpp (override)

Use VMS−style quoting on xsubpp command line.

xsubpp_version (override)

Test xsubpp exit status according to VMS rules (`$sts` & 1 ==> good) rather than Unix rules (`$sts` == 0 ==> good).

tools_other (override)

Adds a few MM[SK] macros, and shortens some the installatin commands, in order to stay under DCL‘s 255−character limit. Also changes EQUALIZE_TIMESTAMP to set revision date of target file to one second later than source file, since MMK interprets precisely equal revision dates for a source and target file as a sign that the target needs to be updated.

dist (override)

Provide VMSish defaults for some values, then hand off to default MM_Unix method.

c_o (override)

Use VMS syntax on command line. In particular, `$(DEFINE)` and `$(PERL_INC)` have been pulled into `$(CCCMD).` Also use MM[SK] macros.

xs_c (override)

Use MM[SK] macros.

xs_o (override)

Use MM[SK] macros, and VMS command line for C compiler.

top_targets (override)

Use VMS quoting on command line for Version_check.

dlsyms (override)

Create VMS linker options files specifying universal symbols for this extension‘s shareable image, and listing other shareable images or libraries to which it should be linked.

dynamic_lib (override)

Use VMS Link command.

dynamic_bs (override)

Use VMS−style quoting on Mkbootstrap command line.

static_lib (override)

Use VMS commands to manipulate object library.

manifypods (override)

Use VMS−style quoting on command line, and VMS logical name to specify fallback location at build time if we can‘t find pod2man.

processPL (override)

Use VMS−style quoting on command line.

installbin (override)

Stay under DCL‘s 255 character command line limit once again by splitting potentially long list of files across multiple lines in `realclean` target.

subdir_x (override)

Use VMS commands to change default directory.

clean (override)

Split potentially long list of files across multiple commands (in order to stay under the magic command line limit). Also use MM[SK] commands for handling subdirectories.

realclean (override)

Guess what we‘re working around? Also, use MM[SK] for subdirectories.

dist_basics (override)

Use VMS−style quoting on command line.

dist_core (override)

Syntax for invoking ***VMS_Share*** differs from that for Unix ***shar***, so `shdist` target actions are VMS−specific.

dist_dir (override)

Use VMS−style quoting on command line.

dist_test (override)

Use VMS commands to change default directory, and use VMS−style quoting on command line.

install (override)

Work around DCL‘s 255 character limit several times,and use VMS−style command line quoting in a few cases.

perldepend (override)

Use VMS−style syntax for files; it‘s cheaper to just do it directly here than to have the MM_Unix method call `catfile` repeatedly. Also use config.vms as source of original config data if the Perl distribution is available; config.sh is an ancillary file under VMS. Finally, if we have to rebuild Config.pm, use MM[SK] to do it.

makefile (override)

Use VMS commands and quoting.

test (override)

Use VMS commands for handling subdirectories.

test_via_harness (override)

Use VMS−style quoting on command line.

test_via_script (override)

Use VMS−style quoting on command line.

makeaperl (override)

Undertake to build a new set of Perl images using VMS commands. Since VMS does dynamic loading, it‘s not necessary to statically link each extension into the Perl image, so this isn‘t the normal build path. Consequently, it hasn‘t really been tested, and may well be incomplete.

nicetext (override)

Insure that colons marking targets are preceded by space, in order to distinguish the target delimiter from a colon appearing as part of a filespec.

NAME

ExtUtils::MM_Win32 − methods to override UN*X behaviour in ExtUtils::MakeMaker

SYNOPSIS

```
 use ExtUtils::MM_Win32; # Done internally by ExtUtils::MakeMaker if needed
```

DESCRIPTION

See ExtUtils::MM_Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

catfile

Concatenate one or more directory names and a filename to form a complete path ending with a filename

static_lib (o)

Defines how to produce the *.a (or equivalent) files.

dynamic_lib (o)

Defines how to produce the *.so (or equivalent) files.

canonpath

No physical check on the filesystem, but a logical cleanup of a path. On UNIX eliminated successive slashes and successive "/.".

perl_script

Takes one argument, a file name, and returns the file name, if the argument is likely to be a perl script. On MM_Unix this is true for any ordinary, readable file.

pm_to_blib

Defines target that copies all files in the hash PM to their destination and autosplits them. See *ExtUtils::Install/DESCRIPTION*

test_via_harness (o)

Helper method to write the test targets

tool_autosplit (override)

Use Win32 quoting on command line.

tools_other (o)

Win32 overrides.

Defines SHELL, LD, TOUCH, CP, MV, RM_F, RM_RF, CHMOD, UMASK_NULL in the Makefile. Also defines the perl programs MKPATH, WARN_IF_OLD_PACKLIST, MOD_INSTALL. DOC_INSTALL, and UNINSTALL.

manifypods (o)

We don‘t want manpage process. XXX add pod2html support later.

dist_ci (o)

Same as MM_Unix version (changes command−line quoting).

dist_core (o)

Same as MM_Unix version (changes command−line quoting).

pasthru (o)

Defines the string that is passed to recursive make calls in subdirectories.

NAME

ExtUtils::MakeMaker − create an extension Makefile

SYNOPSIS

```
use ExtUtils::MakeMaker;
```

```
WriteMakefile( ATTRIBUTE => VALUE [, ...] );
```

which is really

```
MM−>new(\%att)−>flush;
```

DESCRIPTION

This utility is designed to write a Makefile for an extension module from a Makefile.PL. It is based on the Makefile.SH model provided by Andy Dougherty and the perl5−porters.

It splits the task of generating the Makefile into several subroutines that can be individually overridden. Each subroutine returns the text it wishes to have written to the Makefile.

MakeMaker is object oriented. Each directory below the current directory that contains a Makefile.PL. Is treated as a separate object. This makes it possible to write an unlimited number of Makefiles with a single invocation of `WriteMakefile()`.

How To Write A Makefile.PL

The short answer is: Don‘t.

```
        Always begin with h2xs.
        Always begin with h2xs!
        ALWAYS BEGIN WITH H2XS!
```

even if you‘re not building around a header file, and even if you don‘t have an XS component.

Run h2xs(1) before you start thinking about writing a module. For so called pm−only modules that consist of `*.pm` files only, h2xs has the `−X` switch. This will generate dummy files of all kinds that are useful for the module developer.

The medium answer is:

```
    use ExtUtils::MakeMaker;
    WriteMakefile( NAME => "Foo::Bar" );
```

The long answer is the rest of the manpage :−)

Default Makefile Behaviour

The generated Makefile enables the user of the extension to invoke

```
  perl Makefile.PL # optionally "perl Makefile.PL verbose"
  make
  make test        # optionally set TEST_VERBOSE=1
  make install     # See below
```

The Makefile to be produced may be altered by adding arguments of the form `KEY=VALUE`. E.g.

```
  perl Makefile.PL PREFIX=/tmp/myperl5
```

Other interesting targets in the generated Makefile are

```
  make config     # to check if the Makefile is up−to−date
  make clean      # delete local temp files (Makefile gets renamed)
  make realclean  # delete derived files (including ./blib)
  make ci         # check in all the files in the MANIFEST file
  make dist       # see below the Distribution Support section
```

make test

MakeMaker checks for the existence of a file named ***test.pl*** in the current directory and if it exists it adds commands to the test target of the generated Makefile that will execute the script with the proper set of perl `−I` options.

MakeMaker also checks for any files matching glob("t/*.t"). It will add commands to the test target of the generated Makefile that execute all matching files via the *Test::Harness* module with the `−I` switches set correctly.

make testdb

A useful variation of the above is the target `testdb`. It runs the test under the Perl debugger (see *perldebug*). If the file ***test.pl*** exists in the current directory, it is used for the test.

If you want to debug some other testfile, set `TEST_FILE` variable thusly:

```
  make testdb TEST_FILE=t/mytest.t
```

By default the debugger is called using `−d` option to perl. If you want to specify some other option, set `TESTDB_SW` variable:

```
  make testdb TESTDB_SW=−Dx
```

make install

make alone puts all relevant files into directories that are named by the macros INST_LIB, INST_ARCHLIB, INST_SCRIPT, INST_MAN1DIR, and INST_MAN3DIR. All these default to something below ./blib if you are *not* building below the perl source directory. If you *are* building below the perl source, INST_LIB and INST_ARCHLIB default to
../../lib, and INST_SCRIPT is not defined.

The *install* target of the generated Makefile copies the files found below each of the INST_* directories to their INSTALL* counterparts. Which counterparts are chosen depends on the setting of INSTALLDIRS according to the following table:

```
                              INSTALLDIRS set to
                           perl               site

    INST_ARCHLIB     INSTALLARCHLIB    INSTALLSITEARCH
    INST_LIB         INSTALLPRIVLIB    INSTALLSITELIB
    INST_BIN                   INSTALLBIN
    INST_SCRIPT               INSTALLSCRIPT
    INST_MAN1DIR              INSTALLMAN1DIR
    INST_MAN3DIR              INSTALLMAN3DIR
```

The INSTALL... macros in turn default to their %Config (`$Config{installprivlib}`, `$Config{installarchlib}`, etc.) counterparts.

You can check the values of these variables on your system with

```
    perl '−V:install.*'
```

And to check the sequence in which the library directories are searched by perl, run

```
    perl −le 'print join $/, @INC'
```

PREFIX and LIB attribute

PREFIX and LIB can be used to set several INSTALL* attributes in one go. The quickest way to install a module in a non−standard place might be

```
    perl Makefile.PL LIB=~/lib
```

This will install the module‘s architecture−independent files into ~/lib, the architecture−dependent files into ~/lib/`$archname/auto`.

Another way to specify many INSTALL directories with a single parameter is PREFIX.

```
    perl Makefile.PL PREFIX=~
```

This will replace the string specified by `$Config{prefix}` in all `$Config{install*}` values.

Note, that in both cases the tilde expansion is done by MakeMaker, not by perl by default, nor by make. Conflicts between parmeters LIB, PREFIX and the various INSTALL* arguments are resolved so that XXX

If the user has superuser privileges, and is not working on AFS (Andrew File System) or relatives, then the defaults for INSTALLPRIVLIB, INSTALLARCHLIB, INSTALLSCRIPT, etc. will be appropriate, and this incantation will be the best:

```
    perl Makefile.PL; make; make test
    make install
```

make install per default writes some documentation of what has been done into the file `$(INSTALLARCHLIB)/perllocal.pod.` This feature can be bypassed by calling make pure_install.

AFS users

will have to specify the installation directories as these most probably have changed since perl itself has been installed. They will have to do this by calling

```
    perl Makefile.PL INSTALLSITELIB=/afs/here/today \
        INSTALLSCRIPT=/afs/there/now INSTALLMAN3DIR=/afs/for/manpages
    make
```

Be careful to repeat this procedure every time you recompile an extension, unless you are sure the AFS installation directories are still valid.

Static Linking of a new Perl Binary

An extension that is built with the above steps is ready to use on systems supporting dynamic loading. On systems that do not support dynamic loading, any newly created extension has to be linked together with the available resources. MakeMaker supports the linking process by creating appropriate targets in the Makefile whenever an extension is built. You can invoke the corresponding section of the makefile with

```
    make perl
```

That produces a new perl binary in the current directory with all extensions linked in that can be found in INST_ARCHLIB , SITELIBEXP, and PERL_ARCHLIB. To do that, MakeMaker writes a new Makefile, on UNIX, this is called Makefile.aperl (may be system dependent). If you want to force the creation of a new perl, it is recommended, that you delete this Makefile.aperl, so the directories are searched−through for linkable libraries again.

The binary can be installed into the directory where perl normally resides on your machine with

```
    make inst_perl
```

To produce a perl binary with a different name than `perl`, either say

```
    perl Makefile.PL MAP_TARGET=myperl
    make myperl
    make inst_perl
```

or say

```
    perl Makefile.PL
    make myperl MAP_TARGET=myperl
    make inst_perl MAP_TARGET=myperl
```

In any case you will be prompted with the correct invocation of the `inst_perl` target that installs the new binary into INSTALLBIN.

make inst_perl per default writes some documentation of what has been done into the file `$(INSTALLARCHLIB)/perllocal.pod.` This can be bypassed by calling make pure_inst_perl.

Warning: the inst_perl: target will most probably overwrite your existing perl binary. Use with care!

Sometimes you might want to build a statically linked perl although your system supports dynamic loading. In this case you may explicitly set the linktype with the invocation of the Makefile.PL or make:

```
    perl Makefile.PL LINKTYPE=static    # recommended
```

or

```
    make LINKTYPE=static                # works on most systems
```

Determination of Perl Library and Installation Locations

MakeMaker needs to know, or to guess, where certain things are located. Especially INST_LIB and INST_ARCHLIB (where to put the files during the make(1) run), PERL_LIB and PERL_ARCHLIB (where to read existing modules from), and PERL_INC (header files and `libperl*.*`).

Extensions may be built either using the contents of the perl source directory tree or from the installed perl library. The recommended way is to build extensions after you have run 'make install' on perl itself. You can do that in any directory on your hard disk that is not below the perl source tree. The support for extensions below the ext directory of the perl distribution is only good for the standard extensions that come with perl.

If an extension is being built below the `ext/` directory of the perl source then MakeMaker will set PERL_SRC automatically (e.g., `../..`). If PERL_SRC is defined and the extension is recognized as a standard extension, then other variables default to the following:

```
  PERL_INC     = PERL_SRC
  PERL_LIB     = PERL_SRC/lib
  PERL_ARCHLIB = PERL_SRC/lib
  INST_LIB     = PERL_LIB
  INST_ARCHLIB = PERL_ARCHLIB
```

If an extension is being built away from the perl source then MakeMaker will leave PERL_SRC undefined and default to using the installed copy of the perl library. The other variables default to the following:

```
  PERL_INC     = $archlibexp/CORE
  PERL_LIB     = $privlibexp
  PERL_ARCHLIB = $archlibexp
  INST_LIB     = ./blib/lib
  INST_ARCHLIB = ./blib/arch
```

If perl has not yet been installed then PERL_SRC can be defined on the command line as shown in the previous section.

Which architecture dependent directory?

If you don't want to keep the defaults for the INSTALL* macros, MakeMaker helps you to minimize the typing needed: the usual relationship between INSTALLPRIVLIB and INSTALLARCHLIB is determined by Configure at perl compilation time. MakeMaker supports the user who sets INSTALLPRIVLIB. If INSTALLPRIVLIB is set, but INSTALLARCHLIB not, then MakeMaker defaults the latter to be the same subdirectory of INSTALLPRIVLIB as Configure decided for the counterparts in %Config , otherwise it defaults to INSTALLPRIVLIB. The same relationship holds for INSTALLSITELIB and INSTALLSITEARCH.

MakeMaker gives you much more freedom than needed to configure internal variables and get different results. It is worth to mention, that make(1) also lets you configure most of the variables that are used in the Makefile. But in the majority of situations this will not be necessary, and should only be done, if the author of a package recommends it (or you know what you're doing).

Using Attributes and Parameters

The following attributes can be specified as arguments to `WriteMakefile()` or as NAME=VALUE pairs on the command line:

C Ref to array of *.c file names. Initialised from a directory scan and the values portion of the XS attribute hash. This is not currently used by MakeMaker but may be handy in Makefile.PLs.

CONFIG

Arrayref. E.g. [qw(archname manext)] defines ARCHNAME & MANEXT from config.sh. MakeMaker will add to CONFIG the following values anyway: ar cc cccdlflags ccdlflags dlext dlsrc ld lddlflags ldflags libc lib_ext obj_ext ranlib sitelibexp sitearchexp so

CONFIGURE

CODE reference. The subroutine should return a hash reference. The hash may contain further attributes, e.g. {LIBS => ...}, that have to be determined by some evaluation method.

DEFINE

Something like `"-DHAVE_UNISTD_H"`

DIR

Ref to array of subdirectories containing Makefile.PLs e.g. [‘sdbm’] in ext/SDBM_File

DISTNAME

Your name for distributing the package (by tar file). This defaults to NAME above.

DL_FUNCS

Hashref of symbol names for routines to be made available as universal symbols. Each key/value pair consists of the package name and an array of routine names in that package. Used only under AIX (export lists) and VMS (linker options) at present. The routine names supplied will be expanded in the same way as XSUB names are expanded by the `XS()` macro. Defaults to

```
{"$(NAME)" => ["boot_$(NAME)" ] }
```

e.g.

```
{"RPC" => [qw( boot_rpcb rpcb_gettime getnetconfigent )],
 "NetconfigPtr" => [ ’DESTROY’] }
```

DL_VARS

Array of symbol names for variables to be made available as universal symbols. Used only under AIX (export lists) and VMS (linker options) at present. Defaults to []. (e.g. [qw(Foo_version Foo_numstreams Foo_tree)])

EXCLUDE_EXT

Array of extension names to exclude when doing a static build. This is ignored if INCLUDE_EXT is present. Consult INCLUDE_EXT for more details. (e.g. [qw(Socket POSIX)])

This attribute may be most useful when specified as a string on the commandline: perl Makefile.PL EXCLUDE_EXT=‘Socket Safe’

EXE_FILES

Ref to array of executable files. The files will be copied to the INST_SCRIPT directory. Make realclean will delete them from there again.

NO_VC

In general any generated Makefile checks for the current version of MakeMaker and the version the Makefile was built under. If NO_VC is set, the version check is neglected. Do not write this into your Makefile.PL, use it interactively instead.

FIRST_MAKEFILE

The name of the Makefile to be produced. Defaults to the contents of MAKEFILE, but can be overridden. This is used for the second Makefile that will be produced for the MAP_TARGET.

FULLPERL

Perl binary able to run this extension.

H Ref to array of *.h file names. Similar to C.

INC

Include file dirs eg: `"−I/usr/5include −I/path/to/inc"`

INCLUDE_EXT

Array of extension names to be included when doing a static build. MakeMaker will normally build with all of the installed extensions when doing a static build, and that is usually the desired behavior. If INCLUDE_EXT is present then MakeMaker will build only with those extensions which are explicitly mentioned. (e.g. [qw(Socket POSIX)])

It is not necessary to mention DynaLoader or the current extension when filling in INCLUDE_EXT. If the INCLUDE_EXT is mentioned but is empty then only DynaLoader and the current extension will be included in the build.

This attribute may be most useful when specified as a string on the commandline: perl Makefile.PL INCLUDE_EXT='POSIX Socket Devel::Peek'

INSTALLARCHLIB

Used by 'make install', which copies files from INST_ARCHLIB to this directory if INSTALLDIRS is set to perl.

INSTALLBIN

Directory to install binary files (e.g. tkperl) into.

INSTALLDIRS

Determines which of the two sets of installation directories to choose: installprivlib and installarchlib versus installsitelib and installsitearch. The first pair is chosen with INSTALLDIRS=perl, the second with INSTALLDIRS=site. Default is site.

INSTALLMAN1DIR

This directory gets the man pages at 'make install' time. Defaults to `$Config{installman1dir}.`

INSTALLMAN3DIR

This directory gets the man pages at 'make install' time. Defaults to `$Config{installman3dir}.`

INSTALLPRIVLIB

Used by 'make install', which copies files from INST_LIB to this directory if INSTALLDIRS is set to perl.

INSTALLSCRIPT

Used by 'make install' which copies files from INST_SCRIPT to this directory.

INSTALLSITELIB

Used by 'make install', which copies files from INST_LIB to this directory if INSTALLDIRS is set to site (default).

INSTALLSITEARCH

Used by 'make install', which copies files from INST_ARCHLIB to this directory if INSTALLDIRS is set to site (default).

INST_ARCHLIB

Same as INST_LIB for architecture dependent files.

INST_BIN

Directory to put real binary files during ‘make’. These will be copied to INSTALLBIN during ‘make install’

INST_EXE

Old name for INST_SCRIPT. Deprecated. Please use INST_SCRIPT if you need to use it.

INST_LIB

Directory where we put library files of this extension while building it.

INST_MAN1DIR

Directory to hold the man pages at ‘make’ time

INST_MAN3DIR

Directory to hold the man pages at ‘make’ time

INST_SCRIPT

Directory, where executable files should be installed during ‘make’. Defaults to "./blib/bin", just to have a dummy location during testing. make install will copy the files in INST_SCRIPT to INSTALLSCRIPT.

LDFROM

defaults to "`$(OBJECT)`" and is used in the ld command to specify what files to link/load from (also see dynamic_lib below for how to specify ld flags)

LIBPERL_A

The filename of the perllibrary that will be used together with this extension. Defaults to libperl.a.

LIB

LIB can only be set at `perl Makefile.PL` time. It has the effect of setting both INSTALLPRIVLIB and INSTALLSITELIB to that value regardless any

LIBS

An anonymous array of alternative library specifications to be searched for (in order) until at least one library is found. E.g.

```
'LIBS' => ["-lgdbm", "-ldbm -lfoo", "-L/path -ldbm.nfs"]
```

Mind, that any element of the array contains a complete set of arguments for the ld command. So do not specify

```
'LIBS' => ["-ltcl", "-ltk", "-lX11"]
```

See ODBM_File/Makefile.PL for an example, where an array is needed. If you specify a scalar as in

```
'LIBS' => "-ltcl -ltk -lX11"
```

MakeMaker will turn it into an array with one element.

LINKTYPE

‘static’ or ‘dynamic’ (default unless usedl=undef in config.sh). Should only be used to force static linking (also see linkext below).

MAKEAPERL

Boolean which tells MakeMaker, that it should include the rules to make a perl. This is handled automatically as a switch by MakeMaker. The user normally does not need it.

MAKEFILE

The name of the Makefile to be produced.

MAN1PODS

Hashref of pod−containing files. MakeMaker will default this to all EXE_FILES files that include POD directives. The files listed here will be converted to man pages and installed as was requested at Configure time.

MAN3PODS

Hashref of .pm and .pod files. MakeMaker will default this to all
.pod and any .pm files that include POD directives. The files listed
here will be converted to man pages and installed as was requested at Configure time.

MAP_TARGET

If it is intended, that a new perl binary be produced, this variable may hold a name for that binary. Defaults to perl

MYEXTLIB

If the extension links to a library that it builds set this to the name of the library (see SDBM_File)

NAME

Perl module name for this extension (DBD::Oracle). This will default to the directory name but should be explicitly defined in the Makefile.PL.

NEEDS_LINKING

MakeMaker will figure out, if an extension contains linkable code anywhere down the directory tree, and will set this variable accordingly, but you can speed it up a very little bit, if you define this boolean variable yourself.

NOECHO

Defaults to `@`. By setting it to an empty string you can generate a Makefile that echos all commands. Mainly used in debugging MakeMaker itself.

NORECURS

Boolean. Attribute to inhibit descending into subdirectories.

OBJECT

List of object files, defaults to `‘$(BASEEXT)$(OBJ_EXT)‘,` but can be a long string containing all object files, e.g. "tkpBind.o tkpButton.o tkpCanvas.o"

OPTIMIZE

Defaults to `−O`. Set it to `−g` to turn debugging on. The flag is passed to subdirectory makes.

PERL

Perl binary for tasks that can be done by miniperl

PERLMAINCC

The call to the program that is able to compile perlmain.c. Defaults to `$(CC).`

PERL_ARCHLIB

Same as above for architecture dependent files

PERL_LIB

Directory containing the Perl library to use.

PERL_SRC

Directory containing the Perl source code (use of this should be avoided, it may be undefined)

PL_FILES

Ref to hash of files to be processed as perl programs. MakeMaker will default to any found *.PL file (except Makefile.PL) being keys and the basename of the file being the value. E.g.

```
{'foobar.PL' => 'foobar'}
```

The *.PL files are expected to produce output to the target files themselves.

PM

Hashref of .pm files and *.pl files to be installed. e.g.

```
{'name_of_file.pm' => '$(INST_LIBDIR)/install_as.pm'}
```

By default this will include *.pm and *.pl. If a lib directory exists and is not listed in DIR (above) then any *.pm and *.pl files it contains will also be included by default. Defining PM in the Makefile.PL will override PMLIBDIRS.

PMLIBDIRS

Ref to array of subdirectories containing library files. Defaults to ['lib', `$(BASEEXT)`]. The directories will be scanned and any files they contain will be installed in the corresponding location in the library. A `libscan()` method can be used to alter the behaviour. Defining PM in the Makefile.PL will override PMLIBDIRS.

PREFIX

Can be used to set the three INSTALL* attributes in one go (except for probably INSTALLMAN1DIR, if it is not below PREFIX according to %Config). They will have PREFIX as a common directory node and will branch from that node into lib/, lib/ARCHNAME or whatever Configure decided at the build time of your perl (unless you override one of them, of course).

PREREQ_PM

Hashref: Names of modules that need to be available to run this extension (e.g. Fcntl for SDBM_File) are the keys of the hash and the desired version is the value. If the required version number is 0, we only check if any version is installed already.

SKIP

Arryref. E.g. [qw(name1 name2)] skip (do not write) sections of the Makefile. Caution! Do not use the SKIP attribute for the neglectible speedup. It may seriously damage the resulting Makefile. Only use it, if you really need it.

TYPEMAPS

Ref to array of typemap file names. Use this when the typemaps are in some directory other than the current directory or when they are not named **typemap**. The last typemap in the list takes precedence. A typemap in the current directory has highest precedence, even if it isn't listed in TYPEMAPS. The default system typemap has lowest precedence.

VERSION

Your version number for distributing the package. This defaults to 0.1.

VERSION_FROM

Instead of specifying the VERSION in the Makefile.PL you can let MakeMaker parse a file to determine the version number. The parsing routine requires that the file named by VERSION_FROM contains one single line to compute the version number. The first line in the file that contains the regular expression

```
/\$(([\w\:\']*)\bVERSION)\b.*\=/
```

will be evaluated with `eval()` and the value of the named variable **after** the `eval()` will be assigned to the VERSION attribute of the MakeMaker object. The following lines will be parsed o.k.:

```
$VERSION = '1.00';
( $VERSION ) = '$Revision: 1.211 $ ' =~ /\$Revision:\s+([^\s]+)/;
```

```
    $FOO::VERSION = '1.10';
```

but these will fail:

```
    my $VERSION = '1.01';
    local $VERSION = '1.02';
    local $FOO::VERSION = '1.30';
```

The file named in VERSION_FROM is added as a dependency to Makefile to guarantee, that the Makefile contains the correct VERSION macro after a change of the file.

XS

Hashref of .xs files. MakeMaker will default this. e.g.

```
  {'name_of_file.xs' => 'name_of_file.c'}
```

The .c files will automatically be included in the list of files deleted by a make clean.

XSOPT

String of options to pass to xsubpp. This might include `-C++` or `-extern`. Do not include typemaps here; the TYPEMAP parameter exists for that purpose.

XSPROTOARG

May be set to an empty string, which is identical to `-prototypes`, or `-noprototypes`. See the xsubpp documentation for details. MakeMaker defaults to the empty string.

XS_VERSION

Your version number for the .xs file of this package. This defaults to the value of the VERSION attribute.

Additional lowercase attributes

can be used to pass parameters to the methods which implement that part of the Makefile.

clean

```
  {FILES => "*.xyz foo"}
```

depend

```
  {ANY_TARGET => ANY_DEPENDECY, ...}
```

dist

```
  {TARFLAGS => 'cvfF', COMPRESS => 'gzip', SUFFIX => 'gz',
  SHAR => 'shar -m', DIST_CP => 'ln', ZIP => '/bin/zip',
  ZIPFLAGS => '-rl', DIST_DEFAULT => 'private tardist' }
```

If you specify COMPRESS, then SUFFIX should also be altered, as it is needed to tell make the target file of the compression. Setting DIST_CP to ln can be useful, if you need to preserve the timestamps on your files. DIST_CP can take the values ‘cp‘, which copies the file, ‘ln‘, which links the file, and ‘best’ which copies symbolic links and links the rest. Default is ‘best’.

dynamic_lib

```
  {ARMAYBE => 'ar', OTHERLDFLAGS => '...', INST_DYNAMIC_DEP => '...'}
```

installpm

Deprecated as of MakeMaker 5.23. See *ExtUtils::MM_Unix/pm_to_blib*.

linkext

```
  {LINKTYPE => 'static', 'dynamic' or ''}
```

NB: Extensions that have nothing but *.pm files had to say

```
  {LINKTYPE => ''}
```

with Pre−5.0 MakeMakers. Since version 5.00 of MakeMaker such a line can be deleted safely. MakeMaker recognizes, when there‘s nothing to be linked.

macro

```
{ANY_MACRO => ANY_VALUE, ...}
```

realclean

```
{FILES => '$(INST_ARCHAUTODIR)/*.xyz'}
```

tool_autosplit

```
{MAXLEN =E<gt> 8}
```

Overriding MakeMaker Methods

If you cannot achieve the desired Makefile behaviour by specifying attributes you may define private subroutines in the Makefile.PL. Each subroutines returns the text it wishes to have written to the Makefile. To override a section of the Makefile you can either say:

```
        sub MY::c_o { "new literal text" }
```

or you can edit the default by saying something like:

```
        sub MY::c_o {
            my($inherited) = shift->SUPER::c_o(@_);
            $inherited =~ s/old text/new text/;
            $inherited;
        }
```

If you running experiments with embedding perl as a library into other applications, you might find MakeMaker not sufficient. You‘d better have a look at ExtUtils::embed which is a collection of utilities for embedding.

If you still need a different solution, try to develop another subroutine, that fits your needs and submit the diffs to ***perl5−porters@nicoh.com*** or ***comp.lang.perl.misc*** as appropriate.

For a complete description of all MakeMaker methods see *ExtUtils::MM_Unix*.

Here is a simple example of how to add a new target to the generated Makefile:

```
    sub MY::postamble {
        '
    $(MYEXTLIB): sdbm/Makefile
            cd sdbm && $(MAKE) all
    ';
    }
```

Hintsfile support

MakeMaker.pm uses the architecture specific information from Config.pm. In addition it evaluates architecture specific hints files in a `hints/` directory. The hints files are expected to be named like their counterparts in `PERL_SRC/hints`, but with an `.pl` file name extension (eg. `next_3_2.pl`). They are simply `eval`ed by MakeMaker within the `WriteMakefile()` subroutine, and can be used to execute commands as well as to include special variables. The rules which hintsfile is chosen are the same as in Configure.

The hintsfile is `eval()`ed immediately after the arguments given to WriteMakefile are stuffed into a hash reference `$self` but before this reference becomes blessed. So if you want to do the equivalent to override or create an attribute you would say something like

```
    $self->{LIBS} = ['-ldbm -lucb -lc'];
```

Distribution Support

For authors of extensions MakeMaker provides several Makefile targets. Most of the support comes from the ExtUtils::Manifest module, where additional documentation can be found.

make distcheck

reports which files are below the build directory but not in the MANIFEST file and vice versa. (See `ExtUtils::Manifest::fullcheck()` for details)

make skipcheck

reports which files are skipped due to the entries in the `MANIFEST.SKIP` file (See `ExtUtils::Manifest::skipcheck()` for details)

make distclean

does a realclean first and then the distcheck. Note that this is not needed to build a new distribution as long as you are sure, that the MANIFEST file is ok.

make manifest

rewrites the MANIFEST file, adding all remaining files found (See `ExtUtils::Manifest::mkmanifest()` for details)

make distdir

Copies all the files that are in the MANIFEST file to a newly created directory with the name `$(DISTNAME)−$(VERSION).` If that directory exists, it will be removed first.

make disttest

Makes a distdir first, and runs a `perl Makefile.PL`, a make, and a make test in that directory.

make tardist

First does a distdir. Then a command `$(PREOP)` which defaults to a null command, followed by `$(TOUNIX),` which defaults to a null command under UNIX, and will convert files in distribution directory to UNIX format otherwise. Next it runs `tar` on that directory into a tarfile and deletes the directory. Finishes with a command `$(POSTOP)` which defaults to a null command.

make dist

Defaults to `$(DIST_DEFAULT)` which in turn defaults to tardist.

make uutardist

Runs a tardist first and uuencodes the tarfile.

make shdist

First does a distdir. Then a command `$(PREOP)` which defaults to a null command. Next it runs `shar` on that directory into a sharfile and deletes the intermediate directory again. Finishes with a command `$(POSTOP)` which defaults to a null command. Note: For shdist to work properly a `shar` program that can handle directories is mandatory.

make zipdist

First does a distdir. Then a command `$(PREOP)` which defaults to a null command. Runs `$(ZIP) $(ZIPFLAGS)` on that directory into a zipfile. Then deletes that directory. Finishes with a command `$(POSTOP)` which defaults to a null command.

make ci

Does a `$(CI)` and a `$(RCS_LABEL)` on all files in the MANIFEST file.

Customization of the dist targets can be done by specifying a hash reference to the dist attribute of the WriteMakefile call. The following parameters are recognized:

```
CI            ('ci −u')
COMPRESS      ('compress')
```

```
    POSTOP       ('@ :')
    PREOP        ('@ :')
    TO_UNIX      (depends on the system)
    RCS_LABEL    ('rcs -q -Nv$(VERSION_SYM):')
    SHAR         ('shar')
    SUFFIX       ('Z')
    TAR          ('tar')
    TARFLAGS     ('cvf')
    ZIP          ('zip')
    ZIPFLAGS     ('-r')
```

An example:

```
    WriteMakefile( 'dist' => { COMPRESS=>"gzip", SUFFIX=>"gz" })
```

SEE ALSO

ExtUtils::MM_Unix, ExtUtils::Manifest, ExtUtils::testlib, ExtUtils::Install, ExtUtils::embed

AUTHORS

Andy Dougherty <***dougherа@lafcol.lafayette.edu***, Andreas König <***A.Koenig@franz.ww.TU−Berlin.DE***, Tim Bunce <***Tim.Bunce@ig.co.uk***. VMS support by Charles Bailey <***bailey@genetics.upenn.edu***. OS/2 support by Ilya Zakharevich <***ilya@math.ohio−state.edu***. Contact the makemaker mailing list `mailto:makemaker@franz.ww.tu−berlin.de`, if you have any questions.

NAME

ExtUtils::Manifest − utilities to write and check a MANIFEST file

SYNOPSIS

```
require ExtUtils::Manifest;

ExtUtils::Manifest::mkmanifest;

ExtUtils::Manifest::manicheck;

ExtUtils::Manifest::filecheck;

ExtUtils::Manifest::fullcheck;

ExtUtils::Manifest::skipcheck;

ExtUtild::Manifest::manifind();

ExtUtils::Manifest::maniread($file);

ExtUtils::Manifest::manicopy($read,$target,$how);
```

DESCRIPTION

`Mkmanifest()` writes all files in and below the current directory to a file named in the global variable `$ExtUtils::Manifest::MANIFEST` (which defaults to `MANIFEST`) in the current directory. It works similar to

```
    find . −print
```

but in doing so checks each line in an existing `MANIFEST` file and includes any comments that are found in the existing `MANIFEST` file in the new one. Anything between white space and an end of line within a `MANIFEST` file is considered to be a comment. Filenames and comments are seperated by one or more TAB characters in the output. All files that match any regular expression in a file `MANIFEST.SKIP` (if such a file exists) are ignored.

`Manicheck()` checks if all the files within a `MANIFEST` in the current directory really do exist. It only reports discrepancies and exits silently if MANIFEST and the tree below the current directory are in sync.

`Filecheck()` finds files below the current directory that are not mentioned in the `MANIFEST` file. An optional file `MANIFEST.SKIP` will be consulted. Any file matching a regular expression in such a file will not be reported as missing in the `MANIFEST` file.

`Fullcheck()` does both a `manicheck()` and a `filecheck()`.

`Skipcheck()` lists all the files that are skipped due to your `MANIFEST.SKIP` file.

`Manifind()` retruns a hash reference. The keys of the hash are the files found below the current directory.

Maniread(`$file)` reads a named `MANIFEST` file (defaults to `MANIFEST` in the current directory) and returns a HASH reference with files being the keys and comments being the values of the HASH.

Manicopy(`$read,$target,$how)` copies the files that are the keys in the HASH *%`$read`* to the named target directory. The HASH reference *`$read`* is typically returned by the `maniread()` function. This function is useful for producing a directory tree identical to the intended distribution tree. The third parameter `$how` can be used to specify a different methods of "copying". Valid values are `cp`, which actually copies the files, `ln` which creates hard links, and `best` which mostly links the files but copies any symbolic link to make a tree without any symbolic link. Best is the default.

MANIFEST.SKIP

The file MANIFEST.SKIP may contain regular expressions of files that should be ignored by `mkmanifest()` and `filecheck()`. The regular expressions should appear one on each line. A typical example:

```
        \bRCS\b
        ^MANIFEST\.
        ^Makefile$
        ~$
        \.html$
        \.old$
        ^blib/
        ^MakeMaker−\d
```

EXPORT_OK

`&mkmanifest`, `&manicheck`, `&filecheck`, `&fullcheck`, `&maniread`, and `&manicopy` are exportable.

GLOBAL VARIABLES

`$ExtUtils::Manifest::MANIFEST` defaults to `MANIFEST`. Changing it results in both a different `MANIFEST` and a different `MANIFEST.SKIP` file. This is useful if you want to maintain different distributions for different audiences (say a user version and a developer version including RCS).

`$ExtUtils::Manifest::Quiet` defaults to 0. If set to a true value, all functions act silently.

DIAGNOSTICS

All diagnostic output is sent to `STDERR`.

`Not in MANIFEST:` *file*

is reported if a file is found, that is missing in the `MANIFEST` file which is excluded by a regular expression in the file `MANIFEST.SKIP`.

`No such file:` *file*

is reported if a file mentioned in a `MANIFEST` file does not exist.

`MANIFEST:` *$!*

is reported if `MANIFEST` could not be opened.

`Added to MANIFEST:` *file*

is reported by `mkmanifest()` if `$Verbose` is set and a file is added to MANIFEST. `$Verbose` is set to 1 by default.

SEE ALSO

ExtUtils::MakeMaker which has handy targets for most of the functionality.

AUTHOR

Andreas Koenig <***koenig@franz.ww.TU−Berlin.DE***

NAME

ExtUtils::Mkbootstrap − make a bootstrap file for use by DynaLoader

SYNOPSIS

`mkbootstrap`

DESCRIPTION

Mkbootstrap typically gets called from an extension Makefile.

There is no `*.bs` file supplied with the extension. Instead a `*_BS` file which has code for the special cases, like posix for berkeley db on the NeXT.

This file will get parsed, and produce a maybe empty `@DynaLoader::dl_resolve_using` array for the current architecture. That will be extended by `$BSLOADLIBS,` which was computed by `ExtUtils::Liblist::ext()`. If this array still is empty, we do nothing, else we write a .bs file with an `@DynaLoader::dl_resolve_using` array.

The `*_BS` file can put some code into the generated `*.bs` file by placing it in `$bscode.` This is a handy ‘escape’ mechanism that may prove useful in complex situations.

If @DynaLoader::dl_resolve_using contains `−L*` or `−l*` entries then Mkbootstrap will automatically add a `dl_findfile()` call to the generated `*.bs` file.

NAME

ExtUtils::Mksymlists − write linker options files for dynamic extension

SYNOPSIS

```
use ExtUtils::Mksymlists;
Mksymlists({ NAME     => $name ,
             DL_VARS  => [ $var1, $var2, $var3 ],
             DL_FUNCS => { $pkg1 => [ $func1, $func2 ],
                           $pkg2 => [ $func3 ] });
```

DESCRIPTION

`ExtUtils::Mksymlists` produces files used by the linker under some OSs during the creation of shared libraries for dynamic extensions. It is normally called from a MakeMaker−generated Makefile when the extension is built. The linker option file is generated by calling the function `Mksymlists`, which is exported by default from `ExtUtils::Mksymlists`. It takes one argument, a list of key−value pairs, in which the following keys are recognized:

NAME

This gives the name of the extension (*e.g.* Tk::Canvas) for which the linker option file will be produced.

DL_FUNCS

This is identical to the DL_FUNCS attribute available via MakeMaker, from which it is usually taken. Its value is a reference to an associative array, in which each key is the name of a package, and each value is an a reference to an array of function names which should be exported by the extension. For instance, one might say `DL_FUNCS => { Homer::Iliad   => [ qw(trojans greeks) ], Homer::Odyssey => [ qw(travellers family suitors) ] }`. The function names should be identical to those in the XSUB code; `Mksymlists` will alter the names written to the linker option file to match the changes made by ***xsubpp***. In addition, if none of the functions in a list begin with the string **boot_**, `Mksymlists` will add a bootstrap function for that package, just as xsubpp does. (If a **boot_<pkg>** function is present in the list, it is passed through unchanged.) If DL_FUNCS is not specified, it defaults to the bootstrap function for the extension specified in NAME.

DL_VARS

This is identical to the DL_VARS attribute available via MakeMaker, and, like DL_FUNCS, it is usually specified via MakeMaker. Its value is a reference to an array of variable names which should be exported by the extension.

FILE

This key can be used to specify the name of the linker option file (minus the OS−specific extension), if for some reason you do not want to use the default value, which is the last word of the NAME attribute (*e.g.* for Tk::Canvas, FILE defaults to 'Canvas').

FUNCLIST

This provides an alternate means to specify function names to be exported from the extension. Its value is a reference to an array of function names to be exported by the extension. These names are passed through unaltered to the linker options file.

DLBASE

This item specifies the name by which the linker knows the extension, which may be different from the name of the extension itself (for instance, some linkers add an '_' to the name of the extension). If it is not specified, it is derived from the NAME attribute. It is presently used only by OS2.

When calling `Mksymlists`, one should always specify the NAME attribute. In most cases, this is all that‘s necessary. In the case of unusual extensions, however, the other attributes can be used to provide additional information to the linker.

AUTHOR

Charles Bailey <*bailey@genetics.upenn.edu*>

REVISION

Last revised 14−Feb−1996, for Perl 5.002.

NAME

ExtUtils::testlib − add blib/* directories to @INC

SYNOPSIS

```
use ExtUtils::testlib;
```

DESCRIPTION

After an extension has been built and before it is installed it may be desirable to test it bypassing `make test`. By adding

```
    use ExtUtils::testlib;
```

to a test program the intermediate directories used by `make` are added to @INC.

NAME

xsubpp − compiler to convert Perl XS code into C code

SYNOPSIS

xsubpp [**−v**] [**−C++**] [**−except**] [**−s pattern**] [**−prototypes**] [**−noversioncheck**] [**−typemap typemap**]... file.xs

DESCRIPTION

xsubpp will compile XS code into C code by embedding the constructs necessary to let C functions manipulate Perl values and creates the glue necessary to let Perl access those functions. The compiler uses typemaps to determine how to map C function parameters and variables to Perl values.

The compiler will search for typemap files called *typemap*. It will use the following search path to find default typemaps, with the rightmost typemap taking precedence.

```
        ../../../typemap:../../typemap:../typemap:typemap
```

OPTIONS

−C++ Adds ‘‘extern "C"‘’ to the C code.

−except

Adds exception handling stubs to the C code.

−typemap typemap

Indicates that a user−supplied typemap should take precedence over the default typemaps. This option may be used multiple times, with the last typemap having the highest precedence.

−v Prints the *xsubpp* version number to standard output, then exits.

−prototypes

By default *xsubpp* will not automatically generate prototype code for all xsubs. This flag will enable prototypes.

−noversioncheck

Disables the run time test that determines if the object file (derived from the `.xs` file) and the `.pm` files have the same version number.

ENVIRONMENT

No environment variables are used.

AUTHOR

Larry Wall

MODIFICATION HISTORY

See the file ***changes.pod***.

SEE ALSO

perl(1), perlxs(1), perlxstut(1), perlxs(1)

NAME

Fcntl − load the C Fcntl.h defines

SYNOPSIS

```
    use Fcntl;
    use Fcntl qw(:DEFAULT :flock);
```

DESCRIPTION

This module is just a translation of the C ***fnctl.h*** file. Unlike the old mechanism of requiring a translated ***fnctl.ph*** file, this uses the **h2xs** program (see the Perl source distribution) and your native C compiler. This means that it has a far more likely chance of getting the numbers right.

NOTE

Only `#define` symbols get translated; you must still correctly pack up your own arguments to pass as args for locking functions, etc.

EXPORTED SYMBOLS

By default your system's F_* and O_* constants (eg, F_DUPFD and O_CREAT) are exported into your namespace. You can request that the `flock()` constants (LOCK_SH, LOCK_EX, LOCK_NB and LOCK_UN) be provided by using the tag `:flock`. See *Exporter*.

Please refer to your native `fcntl()` and `open()` documentation to see what constants are implemented in your system.

NAME

fileparse − split a pathname into pieces

basename − extract just the filename from a path

dirname − extract just the directory from a path

SYNOPSIS

```
use File::Basename;

($name,$path,$suffix) = fileparse($fullname,@suffixlist)
fileparse_set_fstype($os_string);
$basename = basename($fullname,@suffixlist);
$dirname = dirname($fullname);

($name,$path,$suffix) = fileparse("lib/File/Basename.pm","\.pm");
fileparse_set_fstype("VMS");
$basename = basename("lib/File/Basename.pm",".pm");
$dirname = dirname("lib/File/Basename.pm");
```

DESCRIPTION

These routines allow you to parse file specifications into useful pieces using the syntax of different operating systems.

fileparse_set_fstype

You select the syntax via the routine `fileparse_set_fstype()`.

If the argument passed to it contains one of the substrings "VMS", "MSDOS", "MacOS", "AmigaOS" or "MSWin32", the file specification syntax of that operating system is used in future calls to `fileparse()`, `basename()`, and `dirname()`. If it contains none of these substrings, UNIX syntax is used. This pattern matching is case−insensitive. If you‘ve selected VMS syntax, and the file specification you pass to one of these routines contains a "/", they assume you are using UNIX emulation and apply the UNIX syntax rules instead, for that function call only.

If the argument passed to it contains one of the substrings "VMS", "MSDOS", "MacOS", "AmigaOS", "os2", "MSWin32" or "RISCOS", then the pattern matching for suffix removal is performed without regard for case, since those systems are not case−sensitive when opening existing files (though some of them preserve case on file creation).

If you haven‘t called `fileparse_set_fstype()`, the syntax is chosen by examining the builtin variable `$^O` according to these rules.

fileparse

The `fileparse()` routine divides a file specification into three parts: a leading **path**, a file **name**, and a **suffix**. The **path** contains everything up to and including the last directory separator in the input file specification. The remainder of the input file specification is then divided into **name** and **suffix** based on the optional patterns you specify in `@suffixlist`. Each element of this list is interpreted as a regular expression, and is matched against the end of **name**. If this succeeds, the matching portion of **name** is removed and prepended to **suffix**. By proper use of `@suffixlist`, you can remove file types or versions for examination.

You are guaranteed that if you concatenate **path**, **name**, and **suffix** together in that order, the result will denote the same file as the input file specification.

EXAMPLES

Using UNIX file syntax:

```
($base,$path,$type) = fileparse('/virgil/aeneid/draft.book7',
                                '\.book\d+');
```

would yield

```
    $base eq 'draft'
    $path eq '/virgil/aeneid/',
    $type eq '.book7'
```

Similarly, using VMS syntax:

```
    ($name,$dir,$type) = fileparse('Doc_Root:[Help]Rhetoric.Rnh',
                                   '\..*');
```

would yield

```
    $name eq 'Rhetoric'
    $dir  eq 'Doc_Root:[Help]'
    $type eq '.Rnh'
```

`basename`

The `basename()` routine returns the first element of the list produced by calling `fileparse()` with the same arguments, except that it always quotes metacharacters in the given suffixes. It is provided for programmer compatibility with the UNIX shell command basename(1).

`dirname`

The `dirname()` routine returns the directory portion of the input file specification. When using VMS or MacOS syntax, this is identical to the second element of the list produced by calling `fileparse()` with the same input file specification. (Under VMS, if there is no directory information in the input file specification, then the current default device and directory are returned.) When using UNIX or MSDOS syntax, the return value conforms to the behavior of the UNIX shell command dirname(1). This is usually the same as the behavior of `fileparse()`, but differs in some cases. For example, for the input file specification ***lib/***, `fileparse()` considers the directory name to be ***lib/***, while `dirname()` considers the directory name to be `.`).

NAME

validate − run many filetest checks on a tree

SYNOPSIS

```
use File::CheckTree;

$warnings += validate( q{
    /vmunix                 −e || die
    /boot                   −e || die
    /bin                    cd
        csh                 −ex
        csh                 !−ug
        sh                  −ex
        sh                  !−ug
    /usr                    −d || warn "What happened to $file?\n"
});
```

DESCRIPTION

The `validate()` routine takes a single multiline string consisting of lines containing a filename plus a file test to try on it. (The file test may also be a "cd", causing subsequent relative filenames to be interpreted relative to that directory.) After the file test you may put `|| die` to make it a fatal error if the file test fails. The default is `|| warn`. The file test may optionally have a "!' prepended to test for the opposite condition. If you do a cd and then list some relative filenames, you may want to indent them slightly for readability. If you supply your own `die()` or `warn()` message, you can use `$file` to interpolate the filename.

Filetests may be bunched: "−rwx" tests for all of `−r`, `−w`, and `−x`. Only the first failed test of the bunch will produce a warning.

The routine returns the number of warnings issued.

NAME

File::Copy − Copy files or filehandles

SYNOPSIS

```
use File::Copy;

copy("file1","file2");
copy("Copy.pm",\*STDOUT);'
move("/dev1/fileA","/dev2/fileB");

use POSIX;
use File::Copy cp;

$n=FileHandle−>new("/dev/null","r");
cp($n,"x");'
```

DESCRIPTION

The File::Copy module provides two basic functions, `copy` and `move`, which are useful for getting the contents of a file from one place to another.

- The `copy` function takes two parameters: a file to copy from and a file to copy to. Either argument may be a string, a FileHandle reference or a FileHandle glob. Obviously, if the first argument is a filehandle of some sort, it will be read from, and if it is a file *name* it will be opened for reading. Likewise, the second argument will be written to (and created if need be).

  Note that passing in files as handles instead of names may lead to loss of information on some operating systems; it is recommended that you use file names whenever possible. Files are opened in binary mode where applicable. To get a consistent behavour when copying from a filehandle to a file, use `binmode` on the filehandle.

  An optional third parameter can be used to specify the buffer size used for copying. This is the number of bytes from the first file, that wil be held in memory at any given time, before being written to the second file. The default buffer size depends upon the file, but will generally be the whole file (up to 2Mb), or 1k for filehandles that do not reference files (eg. sockets).

  You may use the syntax `use File::Copy "cp"` to get at the "cp" alias for this function. The syntax is *exactly* the same.

- The `move` function also takes two parameters: the current name and the intended name of the file to be moved. If the destination already exists and is a directory, and the source is not a directory, then the source file will be renamed into the directory specified by the destination.

  If possible, `move()` will simply rename the file. Otherwise, it copies the file to the new location and deletes the original. If an error occurs during this copy−and−delete process, you may be left with a (possibly partial) copy of the file under the destination name.

  You may use the "mv" alias for this function in the same way that you may use the "cp" alias for `copy`.

File::Copy also provides the `syscopy` routine, which copies the file specified in the first parameter to the file specified in the second parameter, preserving OS−specific attributes and file structure. For Unix systems, this is equivalent to the simple `copy` routine. For VMS systems, this calls the `rmscopy` routine (see below). For OS/2 systems, this calls the `syscopy` XSUB directly.

Special behavior if syscopy is defined (VMS and OS/2)

If both arguments to `copy` are not file handles, then `copy` will perform a "system copy" of the input file to a new output file, in order to preserve file attributes, indexed file structure, *etc.* The buffer size parameter is ignored. If either argument to `copy` is a handle to an opened file, then data is copied using Perl operators, and no effort is made to preserve file attributes or record structure.

The system copy routine may also be called directly under VMS and OS/2 as `File::Copy::syscopy` (or under VMS as `File::Copy::rmscopy`, which is the routine that does the actual work for syscopy).

rmscopy(`$from,$to[,$date_flag]`)

The first and second arguments may be strings, typeglobs, typeglob references, or objects inheriting from IO::Handle; they are used in all cases to obtain the *filespec* of the input and output files, respectively. The name and type of the input file are used as defaults for the output file, if necessary.

A new version of the output file is always created, which inherits the structure and RMS attributes of the input file, except for owner and protections (and possibly timestamps; see below). All data from the input file is copied to the output file; if either of the first two parameters to `rmscopy` is a file handle, its position is unchanged. (Note that this means a file handle pointing to the output file will be associated with an old version of that file after `rmscopy` returns, not the newly created version.)

The third parameter is an integer flag, which tells `rmscopy` how to handle timestamps. If it is < 0, none of the input file‘s timestamps are propagated to the output file. If it is > 0, then it is interpreted as a bitmask: if bit 0 (the LSB) is set, then timestamps other than the revision date are propagated; if bit 1 is set, the revision date is propagated. If the third parameter to `rmscopy` is 0, then it behaves much like the DCL COPY command: if the name or type of the output file was explicitly specified, then no timestamps are propagated, but if they were taken implicitly from the input filespec, then all timestamps other than the revision date are propagated. If this parameter is not supplied, it defaults to 0.

Like `copy`, `rmscopy` returns 1 on success. If an error occurs, it sets `$!,` deletes the output file, and returns 0.

RETURN

All functions return 1 on success, 0 on failure. `$!` will be set if an error was encountered.

AUTHOR

File::Copy was written by Aaron Sherman <*ajs@ajs.com*> in 1995, and updated by Charles Bailey <*bailey@genetics.upenn.edu*> in 1996.

NAME

find − traverse a file tree

finddepth − traverse a directory structure depth−first

SYNOPSIS

```
use File::Find;
find(\&wanted, '/foo','/bar');
sub wanted { ... }

use File::Find;
finddepth(\&wanted, '/foo','/bar');
sub wanted { ... }
```

DESCRIPTION

The `wanted()` function does whatever verifications you want. `$File::Find::dir` contains the current directory name, and `$_` the current filename within that directory. `$File::Find::name` contains `"$File::Find::dir/$_".` You are `chdir()`'d to `$File::Find::dir` when the function is called. The function may set `$File::Find::prune` to prune the tree.

File::Find assumes that you don't alter the `$_` variable. If you do then make sure you return it to its original value before exiting your function.

This library is primarily for the `find2perl` tool, which when fed,

```
find2perl / −name .nfs\* −mtime +7 \
    −exec rm −f {} \; −o −fstype nfs −prune
```

produces something like:

```
sub wanted {
    /^\.nfs.*$/ &&
    (($dev,$ino,$mode,$nlink,$uid,$gid) = lstat($_)) &&
    int(−M _) > 7 &&
    unlink($_)
    ||
    ($nlink || (($dev,$ino,$mode,$nlink,$uid,$gid) = lstat($_))) &&
    $dev < 0 &&
    ($File::Find::prune = 1);
}
```

Set the variable `$File::Find::dont_use_nlink` if you're using AFS, since AFS cheats.

`finddepth` is just like `find`, except that it does a depth−first search.

Here's another interesting wanted function. It will find all symlinks that don't resolve:

```
sub wanted {
    −l && !−e && print "bogus link: $File::Find::name\n";
}
```

NAME

File::Path − create or remove a series of directories

SYNOPSIS

```
use File::Path

mkpath(['/foo/bar/baz', 'blurfl/quux'], 1, 0711);

rmtree(['foo/bar/baz', 'blurfl/quux'], 1, 1);
```

DESCRIPTION

The `mkpath` function provides a convenient way to create directories, even if your `mkdir` kernel call won't create more than one level of directory at a time. `mkpath` takes three arguments:

- the name of the path to create, or a reference to a list of paths to create,
- a boolean value, which if TRUE will cause `mkpath` to print the name of each directory as it is created (defaults to FALSE), and
- the numeric mode to use when creating the directories (defaults to 0777)

It returns a list of all directories (including intermediates, determined using the Unix '/' separator) created.

Similarly, the `rmtree` function provides a convenient way to delete a subtree from the directory structure, much like the Unix command `rm −r`. `rmtree` takes three arguments:

- the root of the subtree to delete, or a reference to a list of roots. All of the files and directories below each root, as well as the roots themselves, will be deleted.
- a boolean value, which if TRUE will cause `rmtree` to print a message each time it examines a file, giving the name of the file, and indicating whether it's using `rmdir` or `unlink` to remove it, or that it's skipping it. (defaults to FALSE)
- a boolean value, which if TRUE will cause `rmtree` to skip any files to which you do not have delete access (if running under VMS) or write access (if running under another OS). This will change in the future when a criterion for 'delete permission' under OSs other than VMS is settled. (defaults to FALSE)

It returns the number of files successfully deleted. Symlinks are treated as ordinary files.

AUTHORS

Tim Bunce <***Tim.Bunce@ig.co.uk***> Charles Bailey <***bailey@genetics.upenn.edu***>

REVISION

This module was last revised 14−Feb−1996, for perl 5.002. `$VERSION` is 1.0101.

NAME

File::stat − by−name interface to Perl‘s built−in `stat()` functions

SYNOPSIS

```
use File::stat;
$st = stat($file) or die "No $file: $!";
if ( ($st−>mode & 0111) && $st−>nlink > 1) ) {
    print "$file is executable with lotsa links\n";
}

use File::stat qw(:FIELDS);
stat($file) or die "No $file: $!";
if ( ($st_mode & 0111) && $st_nlink > 1) ) {
    print "$file is executable with lotsa links\n";
}
```

DESCRIPTION

This module‘s default exports override the core `stat()` and `lstat()` functions, replacing them with versions that return "File::stat" objects. This object has methods that return the similarly named structure field name from the stat(2) function; namely, dev, ino, mode, nlink, uid, gid, rdev, size, atime, mtime, ctime, blksize, and blocks.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your `stat()` and `lstat()` functions.) Access these fields as variables named with a preceding `st_` in front their method names. Thus, `$stat_obj−>dev()` corresponds to `$st_dev` if you import the fields.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built−ins are still available via the `CORE::` pseudo−package.

NOTE

While this class is currently implemented using the Class::Template module to build a struct−like class, you shouldn‘t rely upon this.

AUTHOR

Tom Christiansen

NAME

FileCache − keep more files open than the system permits

SYNOPSIS

```
    cacheout $path;
    print $path @data;
```

DESCRIPTION

The `cacheout` function will make sure that there‘s a filehandle open for writing available as the pathname you give it. It automatically closes and re−opens files if you exceed your system file descriptor maximum.

BUGS

sys/param.h lies with its `NOFILE` define on some systems, so you may have to set `$cacheout::maxopen` yourself.

NAME

FileHandle − supply object methods for filehandles

SYNOPSIS

```
use FileHandle;

$fh = new FileHandle;
if ($fh−>open "< file") {
    print <$fh>;
    $fh−>close;
}

$fh = new FileHandle "> FOO";
if (defined $fh) {
    print $fh "bar\n";
    $fh−>close;
}

$fh = new FileHandle "file", "r";
if (defined $fh) {
    print <$fh>;
    undef $fh;       # automatically closes the file
}

$fh = new FileHandle "file", O_WRONLY|O_APPEND;
if (defined $fh) {
    print $fh "corge\n";
    undef $fh;       # automatically closes the file
}

$pos = $fh−>getpos;
$fh−>setpos($pos);

$fh−>setvbuf($buffer_var, _IOLBF, 1024);

($readfh, $writefh) = FileHandle::pipe;

autoflush STDOUT 1;
```

DESCRIPTION

NOTE: This class is now a front−end to the IO::* classes.

`FileHandle::new` creates a `FileHandle`, which is a reference to a newly created symbol (see the `Symbol` package). If it receives any parameters, they are passed to `FileHandle::open`; if the open fails, the `FileHandle` object is destroyed. Otherwise, it is returned to the caller.

`FileHandle::new_from_fd` creates a `FileHandle` like `new` does. It requires two parameters, which are passed to `FileHandle::fdopen`; if the fdopen fails, the `FileHandle` object is destroyed. Otherwise, it is returned to the caller.

`FileHandle::open` accepts one parameter or two. With one parameter, it is just a front end for the built−in `open` function. With two parameters, the first parameter is a filename that may include whitespace or other special characters, and the second parameter is the open mode, optionally followed by a file permission value.

If `FileHandle::open` receives a Perl mode string ("", "+<", etc.) or a POSIX `fopen()` mode string ("w", "r+", etc.), it uses the basic Perl `open` operator.

If `FileHandle::open` is given a numeric mode, it passes that mode and the optional permissions value to the Perl `sysopen` operator. For convenience, `FileHandle::import` tries to import the O_XXX constants from the Fcntl module. If dynamic loading is not available, this may fail, but the rest of

FileHandle will still work.

`FileHandle::fdopen` is like `open` except that its first parameter is not a filename but rather a file handle name, a FileHandle object, or a file descriptor number.

If the C functions `fgetpos()` and `fsetpos()` are available, then `FileHandle::getpos` returns an opaque value that represents the current position of the FileHandle, and `FileHandle::setpos` uses that value to return to a previously visited position.

If the C function `setvbuf()` is available, then `FileHandle::setvbuf` sets the buffering policy for the FileHandle. The calling sequence for the Perl function is the same as its C counterpart, including the macros `_IOFBF`, `_IOLBF`, and `_IONBF`, except that the buffer parameter specifies a scalar variable to use as a buffer. WARNING: A variable used as a buffer by `FileHandle::setvbuf` must not be modified in any way until the FileHandle is closed or until `FileHandle::setvbuf` is called again, or memory corruption may result!

See *perlfunc* for complete descriptions of each of the following supported `FileHandle` methods, which are just front ends for the corresponding built−in functions:

```
close
fileno
getc
gets
eof
clearerr
seek
tell
```

See *perlvar* for complete descriptions of each of the following supported `FileHandle` methods:

```
autoflush
output_field_separator
output_record_separator
input_record_separator
input_line_number
format_page_number
format_lines_per_page
format_lines_left
format_name
format_top_name
format_line_break_characters
format_formfeed
```

Furthermore, for doing normal I/O you might need these:

$fh−print

See *print*.

$fh−printf

See *printf*.

$fh−getline

This works like <`$fh` described in *I/O Operators in perlop* except that it‘s more readable and can be safely called in an array context but still returns just one line.

$fh−getlines

This works like <`$fh` when called in an array context to read all the remaining lines in a file, except that it‘s more readable. It will also `croak()` if accidentally called in a scalar context.

SEE ALSO

The **IO** extension, *perlfunc*, *I/O Operators in perlop*.

NAME

FindBin − Locate directory of original perl script

SYNOPSIS

```
 use FindBin;
 use lib "$FindBin::Bin/../lib";

 or

 use FindBin qw($Bin);
 use lib "$Bin/../lib";
```

DESCRIPTION

Locates the full path to the script bin directory to allow the use of paths relative to the bin directory.

This allows a user to setup a directory tree for some software with directories <root>/bin and <root>/lib and then the above example will allow the use of modules in the lib directory without knowing where the software tree is installed.

If perl is invoked using the **−e** option or the perl script is read from `STDIN` then FindBin sets both `$Bin` and `$RealBin` to the current directory.

EXPORTABLE VARIABLES

```
 $Bin         − path to bin directory from where script was invoked
 $Script      − basename of script from which perl was invoked
 $RealBin     − $Bin with all links resolved
 $RealScript  − $Script with all links resolved
```

KNOWN BUGS

if perl is invoked as

```
   perl filename
```

and *filename* does not have executable rights and a program called *filename* exists in the users `$ENV{PATH}` which satisfies both **−x** and **−T** then FindBin assumes that it was invoked via the `$ENV{PATH}`.

Workaround is to invoke perl as

```
 perl ./filename
```

AUTHORS

Graham Barr <***bodg@tiuk.ti.com***> Nick Ing−Simmons <***nik@tiuk.ti.com***>

COPYRIGHT

Copyright (c) 1995 Graham Barr & Nick Ing−Simmons. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

REVISION

`$Revision:` 1.4 `$`

NAME

GetOptions − extended processing of command line options

SYNOPSIS

```
    use Getopt::Long;
    $result = GetOptions (...option−descriptions...);
```

DESCRIPTION

The Getopt::Long module implements an extended getopt function called `GetOptions()`. This function adheres to the POSIX syntax for command line options, with GNU extensions. In general, this means that options have long names instead of single letters, and are introduced with a double dash "—". Support for bundling of command line options, as was the case with the more traditional single−letter approach, is provided but not enabled by default. For example, the UNIX "ps" command can be given the command line "option"

```
    −vax
```

which means the combination of **−v**, **−a** and **−x**. With the new syntax **—vax** would be a single option, probably indicating a computer architecture.

Command line options can be used to set values. These values can be specified in one of two ways:

```
    −−size 24
    −−size=24
```

GetOptions is called with a list of option−descriptions, each of which consists of two elements: the option specifier and the option linkage. The option specifier defines the name of the option and, optionally, the value it can take. The option linkage is usually a reference to a variable that will be set when the option is used. For example, the following call to GetOptions:

```
    &GetOptions("size=i" => \$offset);
```

will accept a command line option "size" that must have an integer value. With a command line of "—size 24" this will cause the variable `$offset` to get the value 24.

Alternatively, the first argument to GetOptions may be a reference to a HASH describing the linkage for the options. The following call is equivalent to the example above:

```
    %optctl = ("size" => \$offset);
    &GetOptions(\%optctl, "size=i");
```

Linkage may be specified using either of the above methods, or both. Linkage specified in the argument list takes precedence over the linkage specified in the HASH.

The command line options are taken from array @ARGV. Upon completion of GetOptions, @ARGV will contain the rest (i.e. the non−options) of the command line.

Each option specifier designates the name of the option, optionally followed by an argument specifier. Values for argument specifiers are:

<none>	Option does not take an argument. The option variable will be set to 1.
!	Option does not take an argument and may be negated, i.e. prefixed by "no". E.g. "foo!" will allow **—foo** (with value 1) and **−nofoo** (with value 0). The option variable will be set to 1, or 0 if negated.
=s	Option takes a mandatory string argument. This string will be assigned to the option variable. Note that even if the string argument starts with − or —, it will not be considered an option on itself.

:s	Option takes an optional string argument. This string will be assigned to the option variable. If omitted, it will be assigned "" (an empty string). If the string argument starts with − or —, it will be considered an option on itself.
=i	Option takes a mandatory integer argument. This value will be assigned to the option variable. Note that the value may start with − to indicate a negative value.
:i	Option takes an optional integer argument. This value will be assigned to the option variable. If omitted, the value 0 will be assigned. Note that the value may start with − to indicate a negative value.
=f	Option takes a mandatory real number argument. This value will be assigned to the option variable. Note that the value may start with − to indicate a negative value.
:f	Option takes an optional real number argument. This value will be assigned to the option variable. If omitted, the value 0 will be assigned.

A lone dash − is considered an option, the corresponding option name is the empty string.

A double dash on itself — signals end of the options list.

Linkage specification

The linkage specifier is optional. If no linkage is explicitly specified but a ref HASH is passed, GetOptions will place the value in the HASH. For example:

```
%optctl = ();
&GetOptions (\%optctl, "size=i");
```

will perform the equivalent of the assignment

```
$optctl{"size"} = 24;
```

For array options, a reference to an array is used, e.g.:

```
%optctl = ();
&GetOptions (\%optctl, "sizes=i@");
```

with command line "−sizes 24 −sizes 48" will perform the equivalent of the assignment

```
$optctl{"sizes"} = [24, 48];
```

For hash options (an option whose argument looks like "name=value"), a reference to a hash is used, e.g.:

```
%optctl = ();
&GetOptions (\%optctl, "define=s%");
```

with command line "—define foo=hello —define bar=world" will perform the equivalent of the assignment

```
$optctl{"define"} = {foo=>'hello', bar=>'world')
```

If no linkage is explicitly specified and no ref HASH is passed, GetOptions will put the value in a global variable named after the option, prefixed by "opt_". To yield a usable Perl variable, characters that are not part of the syntax for variables are translated to underscores. For example, "—fpp−struct−return" will set the variable `$opt_fpp_struct_return.` Note that this variable resides in the namespace of the calling program, not necessarily **main**. For example:

```
&GetOptions ("size=i", "sizes=i@");
```

with command line "−size 10 −sizes 24 −sizes 48" will perform the equivalent of the assignments

```
$opt_size = 10;
@opt_sizes = (24, 48);
```

A lone dash − is considered an option, the corresponding Perl identifier is `$opt_` .

The linkage specifier can be a reference to a scalar, a reference to an array, a reference to a hash or a reference to a subroutine.

If a REF SCALAR is supplied, the new value is stored in the referenced variable. If the option occurs more than once, the previous value is overwritten.

If a REF ARRAY is supplied, the new value is appended (pushed) to the referenced array.

If a REF HASH is supplied, the option value should look like "key" or "key=value" (if the "=value" is omitted then a value of 1 is implied). In this case, the element of the referenced hash with the key "key" is assigned "value".

If a REF CODE is supplied, the referenced subroutine is called with two arguments: the option name and the option value. The option name is always the true name, not an abbreviation or alias.

Aliases and abbreviations

The option name may actually be a list of option names, separated by "|"s, e.g. "foo|bar|blech=s". In this example, "foo" is the true name of this option. If no linkage is specified, options "foo", "bar" and "blech" all will set `$opt_foo.`

Option names may be abbreviated to uniqueness, depending on configuration option **auto_abbrev**.

Non−option call−back routine

A special option specifier, <>, can be used to designate a subroutine to handle non−option arguments. GetOptions will immediately call this subroutine for every non−option it encounters in the options list. This subroutine gets the name of the non−option passed. This feature requires configuration option **permute**, see section CONFIGURATION OPTIONS.

See also the examples.

Option starters

On the command line, options can start with − (traditional), –– (POSIX) and + (GNU, now being phased out). The latter is not allowed if the environment variable **POSIXLY_CORRECT** has been defined.

Options that start with "––" may have an argument appended, separated with an "=", e.g. "––foo=bar".

Return value

A return status of 0 (false) indicates that the function detected one or more errors.

COMPATIBILITY

`Getopt::Long::GetOptions()` is the successor of **newgetopt.pl** that came with Perl 4. It is fully upward compatible. In fact, the Perl 5 version of newgetopt.pl is just a wrapper around the module.

If an "@" sign is appended to the argument specifier, the option is treated as an array. Value(s) are not set, but pushed into array @opt_name. If explicit linkage is supplied, this must be a reference to an ARRAY.

If an "%" sign is appended to the argument specifier, the option is treated as a hash. Value(s) of the form "name=value" are set by setting the element of the hash %opt_name with key "name" to "value" (if the "=value" portion is omitted it defaults to 1). If explicit linkage is supplied, this must be a reference to a HASH.

If configuration option **getopt_compat** is set (see section CONFIGURATION OPTIONS), options that start with "+" or "−" may also include their arguments, e.g. "+foo=bar". This is for compatiblity with older implementations of the GNU "getopt" routine.

If the first argument to GetOptions is a string consisting of only non−alphanumeric characters, it is taken to specify the option starter characters. Everything starting with one of these characters from the starter will be considered an option. **Using a starter argument is strongly deprecated.**

For convenience, option specifiers may have a leading − or ––, so it is possible to write:

```
    GetOptions qw(−foo=s −−bar=i −−ar=s);
```

EXAMPLES

If the option specifier is "one:i" (i.e. takes an optional integer argument), then the following situations are handled:

```
−one −two             −> $opt_one = ’’, −two is next option
−one −2               −> $opt_one = −2
```

Also, assume specifiers "foo=s" and "bar:s" :

```
−bar −xxx             −> $opt_bar = ’’, ’−xxx’ is next option
−foo −bar             −> $opt_foo = ’−bar’
−foo −−               −> $opt_foo = ’−−’
```

In GNU or POSIX format, option names and values can be combined:

```
+foo=blech            −> $opt_foo = ’blech’
−−bar=                −> $opt_bar = ’’
−−bar=−−              −> $opt_bar = ’−−’
```

Example of using variable references:

```
$ret = &GetOptions (’foo=s’, \$foo, ’bar=i’, ’ar=s’, \@ar);
```

With command line options "−foo blech −bar 24 −ar xx −ar yy" this will result in:

```
$foo = ’blech’
$opt_bar = 24
@ar = (’xx’,’yy’)
```

Example of using the <> option specifier:

```
@ARGV = qw(−foo 1 bar −foo 2 blech);
&GetOptions("foo=i", \$myfoo, "<>", \&mysub);
```

Results:

```
&mysub("bar") will be called (with $myfoo being 1)
&mysub("blech") will be called (with $myfoo being 2)
```

Compare this with:

```
@ARGV = qw(−foo 1 bar −foo 2 blech);
&GetOptions("foo=i", \$myfoo);
```

This will leave the non−options in @ARGV:

```
$myfoo −> 2
@ARGV −> qw(bar blech)
```

CONFIGURATION OPTIONS

GetOptions can be configured by calling subroutine **Getopt::Long::config**. This subroutine takes a list of quoted strings, each specifying a configuration option to be set, e.g. **ignore_case**. Options can be reset by prefixing with **no_**, e.g. **no_ignore_case**. Case does not matter. Multiple calls to **config** are possible.

Previous versions of Getopt::Long used variables for the purpose of configuring. Although manipulating these variables still work, it is strongly encouraged to use the new **config** routine. Besides, it is much easier.

The following options are available:

default	This option causes all configuration options to be reset to their default values.
auto_abbrev	Allow option names to be abbreviated to uniqueness. Default is set unless environment variable POSIXLY_CORRECT has been set, in which case **auto_abbrev** is reset.

getopt_compat

Allow '+' to start options. Default is set unless environment variable POSIXLY_CORRECT has been set, in which case **getopt_compat** is reset.

require_order

Whether non−options are allowed to be mixed with options. Default is set unless environment variable POSIXLY_CORRECT has been set, in which case b<require_order is reset.

See also **permute**, which is the opposite of **require_order**.

permute

Whether non−options are allowed to be mixed with options. Default is set unless environment variable POSIXLY_CORRECT has been set, in which case **permute** is reset. Note that **permute** is the opposite of **require_order**.

If **permute** is set, this means that

```
−foo arg1 −bar arg2 arg3
```

is equivalent to

```
−foo −bar arg1 arg2 arg3
```

If a non−option call−back routine is specified, @ARGV will always be empty upon succesful return of GetOptions since all options have been processed, except when **−−** is used:

```
−foo arg1 −bar arg2 -- arg3
```

will call the call−back routine for arg1 and arg2, and terminate leaving arg2 in @ARGV.

If **require_order** is set, options processing terminates when the first non−option is encountered.

```
−foo arg1 −bar arg2 arg3
```

is equivalent to

```
−foo -- arg1 −bar arg2 arg3
```

bundling (default: reset)

Setting this variable to a non−zero value will allow single−character options to be bundled. To distinguish bundles from long option names, long options must be introduced with **−−** and single−character options (and bundles) with **−**. For example,

```
ps −vax −−vax
```

would be equivalent to

```
ps −v −a −x −−vax
```

provided "vax", "v", "a" and "x" have been defined to be valid options.

Bundled options can also include a value in the bundle; this value has to be the last part of the bundle, e.g.

```
scale −h24 −w80
```

is equivalent to

```
scale −h 24 −w 80
```

Note: resetting **bundling** also resets **bundling_override**.

bundling_override (default: reset)

If **bundling_override** is set, bundling is enabled as with **bundling** but now long option names override option bundles. In the above example, **−vax** would be interpreted as the option "vax", not the bundle "v", "a", "x".

Note: resetting **bundling_override** also resets **bundling**.

Note: Using option bundling can easily lead to unexpected results, especially when mixing long options and bundles. Caveat emptor.

ignore_case (default: set)

If set, case is ignored when matching options.

Note: resetting **ignore_case** also resets **ignore_case_always**.

ignore_case_always (default: reset)

When bundling is in effect, case is ignored on single−character options also.

Note: resetting **ignore_case_always** also resets **ignore_case**.

pass_through (default: reset)

Unknown options are passed through in @ARGV instead of being flagged as errors. This makes it possible to write wrapper scripts that process only part of the user supplied options, and passes the remaining options to some other program.

This can be very confusing, especially when **permute** is also set.

debug (default: reset)

Enable copious debugging output.

OTHER USEFUL VARIABLES

`$Getopt::Long::VERSION`

The version number of this Getopt::Long implementation in the format `major`.`minor`. This can be used to have Exporter check the version, e.g.

```
    use Getopt::Long 3.00;
```

You can inspect `$Getopt::Long::major_version` and `$Getopt::Long::minor_version` for the individual components.

`$Getopt::Long::error`

Internal error flag. May be incremented from a call−back routine to cause options parsing to fail.

NAME

getopt − Process single−character switches with switch clustering

getopts − Process single−character switches with switch clustering

SYNOPSIS

```
use Getopt::Std;

getopt('oDI');    # −o, −D & −I take arg.  Sets opt_* as a side effect.
getopt('oDI', \%opts);    # −o, −D & −I take arg.  Values in %opts
getopts('oif:');  # −o & −i are boolean flags, −f takes an argument
                  # Sets opt_* as a side effect.
getopts('oif:', \%opts);  # options as above. Values in %opts
```

DESCRIPTION

The `getopt()` functions processes single−character switches with switch clustering. Pass one argument which is a string containing all switches that take an argument. For each switch found, sets `$opt_x` (where x is the switch name) to the value of the argument, or 1 if no argument. Switches which take an argument don‘t care whether there is a space between the switch and the argument.

For those of you who don‘t like additional variables being created, `getopt()` and `getopts()` will also accept a hash reference as an optional second argument. Hash keys will be x (where x is the switch name) with key values the value of the argument or 1 if no argument is specified.

NAME

I18N::Collate − compare 8−bit scalar data according to the current locale

SYNOPSIS

```
    use I18N::Collate;
    setlocale(LC_COLLATE, ’locale−of−your−choice’);
    $s1 = new I18N::Collate "scalar_data_1";
    $s2 = new I18N::Collate "scalar_data_2";
```

DESCRIPTION

This module provides you with objects that will collate according to your national character set, provided that the POSIX `setlocale()` function is supported on your system.

You can compare `$s1` and `$s2` above with

```
    $s1 le $s2
```

to extract the data itself, you‘ll need a dereference: `$$s1`

This module uses `POSIX::setlocale()`. The basic collation conversion is done by `strxfrm()` which terminates at NUL characters being a decent C routine. `collate_xfrm()` handles embedded NUL characters gracefully.

The available locales depend on your operating system; try whether `locale −a` shows them or man pages for "locale" or "nlsinfo" or the direct approach `ls /usr/lib/nls/loc` or `ls /usr/lib/nls` or `ls /usr/lib/locale`. Not all the locales that your vendor supports are necessarily installed: please consult your operating system‘s documentation and possibly your local system administration. The locale names are probably something like `xx_XX.(ISO)?8859−N` or `xx_XX.(ISO)?8859N`, for example `fr_CH.ISO8859−1` is the Swiss (CH) variant of French (fr), ISO Latin (8859) 1 (−1) which is the Western European character set.

NAME

IO::File − supply object methods for filehandles

SYNOPSIS

```
use IO::File;

$fh = new IO::File;
if ($fh−>open("< file")) {
    print <$fh>;
    $fh−>close;
}

$fh = new IO::File "> file";
if (defined $fh) {
    print $fh "bar\n";
    $fh−>close;
}

$fh = new IO::File "file", "r";
if (defined $fh) {
    print <$fh>;
    undef $fh;       # automatically closes the file
}

$fh = new IO::File "file", O_WRONLY|O_APPEND;
if (defined $fh) {
    print $fh "corge\n";

    $pos = $fh−>getpos;
    $fh−>setpos($pos);

    undef $fh;       # automatically closes the file
}

autoflush STDOUT 1;
```

DESCRIPTION

`IO::File` inherits from `IO::Handle` and `IO::Seekable`. It extends these classes with methods that are specific to file handles.

CONSTRUCTOR

new ([ARGS])

Creates a `IO::File`. If it receives any parameters, they are passed to the method `open`; if the open fails, the object is destroyed. Otherwise, it is returned to the caller.

new_tmpfile

Creates an `IO::File` opened for read/write on a newly created temporary file. On systems where this is possible, the temporary file is anonymous (i.e. it is unlinked after creation, but held open). If the temporary file cannot be created or opened, the `IO::File` object is destroyed. Otherwise, it is returned to the caller.

METHODS

open(FILENAME [,MODE [,PERMS]])

`open` accepts one, two or three parameters. With one parameter, it is just a front end for the built−in `open` function. With two parameters, the first parameter is a filename that may include whitespace or other special characters, and the second parameter is the open mode, optionally followed by a file permission value.

If `IO::File::open` receives a Perl mode string (">", "+<", etc.) or a POSIX `fopen()` mode string ("w", "r+", etc.), it uses the basic Perl `open` operator.

If `IO::File::open` is given a numeric mode, it passes that mode and the optional permissions value to the Perl `sysopen` operator. For convenience, `IO::File::import` tries to import the O_XXX constants from the Fcntl module. If dynamic loading is not available, this may fail, but the rest of IO::File will still work.

SEE ALSO

perlfunc, *I/O Operators in perlop*, *IO::Handle IO::Seekable*

HISTORY

Derived from FileHandle.pm by Graham Barr <***bodg@tiuk.ti.com***>.

NAME

IO::Handle − supply object methods for I/O handles

SYNOPSIS

```
    use IO::Handle;

    $fh = new IO::Handle;
    if ($fh−>fdopen(fileno(STDIN),"r")) {
        print $fh−>getline;
        $fh−>close;
    }

    $fh = new IO::Handle;
    if ($fh−>fdopen(fileno(STDOUT),"w")) {
        $fh−>print("Some text\n");
    }

    $fh−>setvbuf($buffer_var, _IOLBF, 1024);

    undef $fh;       # automatically closes the file if it’s open

    autoflush STDOUT 1;
```

DESCRIPTION

`IO::Handle` is the base class for all other IO handle classes. It is not intended that objects of `IO::Handle` would be created directly, but instead `IO::Handle` is inherited from by several other classes in the IO hierarchy.

If you are reading this documentation, looking for a replacement for the `FileHandle` package, then I suggest you read the documentation for `IO::File`

A `IO::Handle` object is a reference to a symbol (see the `Symbol` package)

CONSTRUCTOR

new ( )

Creates a new `IO::Handle` object.

new_from_fd (FD, MODE)

Creates a `IO::Handle` like `new` does. It requires two parameters, which are passed to the method `fdopen`; if the fdopen fails, the object is destroyed. Otherwise, it is returned to the caller.

METHODS

If the C function `setvbuf()` is available, then `IO::Handle::setvbuf` sets the buffering policy for the IO::Handle. The calling sequence for the Perl function is the same as its C counterpart, including the macros `_IOFBF`, `_IOLBF`, and `_IONBF`, except that the buffer parameter specifies a scalar variable to use as a buffer. WARNING: A variable used as a buffer by `IO::Handle::setvbuf` must not be modified in any way until the IO::Handle is closed or until `IO::Handle::setvbuf` is called again, or memory corruption may result!

See *perlfunc* for complete descriptions of each of the following supported `IO::Handle` methods, which are just front ends for the corresponding built−in functions:

```
    close
    fileno
    getc
    eof
    read
    truncate
    stat
```

```
print
printf
sysread
syswrite
```

See *perlvar* for complete descriptions of each of the following supported `IO::Handle` methods:

```
autoflush
output_field_separator
output_record_separator
input_record_separator
input_line_number
format_page_number
format_lines_per_page
format_lines_left
format_name
format_top_name
format_line_break_characters
format_formfeed
format_write
```

Furthermore, for doing normal I/O you might need these:

`$fh`−getline

This works like <`$fh` described in *I/O Operators in perlop* except that it‘s more readable and can be safely called in an array context but still returns just one line.

`$fh`−getlines

This works like <`$fh` when called in an array context to read all the remaining lines in a file, except that it‘s more readable. It will also `croak()` if accidentally called in a scalar context.

`$fh`−fdopen (FD, MODE)

`fdopen` is like an ordinary `open` except that its first parameter is not a filename but rather a file handle name, a IO::Handle object, or a file descriptor number.

`$fh`−write (BUF, LEN [, OFFSET }\])

`write` is like `write` found in C, that is it is the opposite of read. The wrapper for the perl `write` function is called `format_write`.

`$fh`−opened

Returns true if the object is currently a valid file descriptor.

Lastly, a special method for working under **−T** and setuid/gid scripts:

`$fh`−untaint

Marks the object as taint−clean, and as such data read from it will also be considered taint−clean. Note that this is a very trusting action to take, and appropriate consideration for the data source and potential vulnerability should be kept in mind.

NOTE

A `IO::Handle` object is a GLOB reference. Some modules that inherit from `IO::Handle` may want to keep object related variables in the hash table part of the GLOB. In an attempt to prevent modules trampling on each other I propose the that any such module should prefix its variables with its own name separated by _‘s. For example the IO::Socket module keeps a `timeout` variable in ‘io_socket_timeout’.

SEE ALSO

perlfunc, *I/O Operators in perlop*, *IO::File*

BUGS

Due to backwards compatibility, all filehandles resemble objects of class `IO::Handle`, or actually classes derived from that class. They actually aren‘t. Which means you can‘t derive your own class from `IO::Handle` and inherit those methods.

HISTORY

Derived from FileHandle.pm by Graham Barr <***bodg@tiuk.ti.com***>

NAME

IO::pipe − supply object methods for pipes

SYNOPSIS

```
use IO::Pipe;

$pipe = new IO::Pipe;

if($pid = fork()) { # Parent
    $pipe−>reader();

    while(<$pipe> {
        ....
    }

}
elsif(defined $pid) { # Child
    $pipe−>writer();

    print $pipe ....
}

or

$pipe = new IO::Pipe;

$pipe−>reader(qw(ls −l));

while(<$pipe>) {
    ....
}
```

DESCRIPTION

`IO::Pipe` provides an interface to createing pipes between processes.

CONSTRCUTOR

new ([READER, WRITER])

Creates a `IO::Pipe`, which is a reference to a newly created symbol (see the `Symbol` package). `IO::Pipe::new` optionally takes two arguments, which should be objects blessed into `IO::Handle`, or a subclass thereof. These two objects will be used for the system call to `pipe`. If no arguments are given then then method `handles` is called on the new `IO::Pipe` object.

These two handles are held in the array part of the GLOB until either `reader` or `writer` is called.

METHODS

reader ([ARGS])

The object is re−blessed into a sub−class of `IO::Handle`, and becomes a handle at the reading end of the pipe. If ARGS are given then `fork` is called and ARGS are passed to exec.

writer ([ARGS])

The object is re−blessed into a sub−class of `IO::Handle`, and becomes a handle at the writing end of the pipe. If ARGS are given then `fork` is called and ARGS are passed to exec.

handles ()

This method is called during construction by `IO::Pipe::new` on the newly created `IO::Pipe` object. It returns an array of two objects blessed into `IO::Pipe::End`, or a subclass thereof.

SEE ALSO

IO::Handle

AUTHOR

Graham Barr <bodg@tiuk.ti.com

COPYRIGHT

NAME

IO::Seekable − supply seek based methods for I/O objects

SYNOPSIS

```
use IO::Seekable;
package IO::Something;
@ISA = qw(IO::Seekable);
```

DESCRIPTION

`IO::Seekable` does not have a constuctor of its own as is intended to be inherited by other `IO::Handle` based objects. It provides methods which allow seeking of the file descriptors.

If the C functions `fgetpos()` and `fsetpos()` are available, then `IO::File::getpos` returns an opaque value that represents the current position of the IO::File, and `IO::File::setpos` uses that value to return to a previously visited position.

See *perlfunc* for complete descriptions of each of the following supported `IO::Seekable` methods, which are just front ends for the corresponding built−in functions:

```
seek
tell
```

SEE ALSO

perlfunc, *I/O Operators in perlop*, *IO::Handle IO::File*

HISTORY

Derived from FileHandle.pm by Graham Barr <bodg@tiuk.ti.com>

NAME

IO::Select − OO interface to the select system call

SYNOPSIS

```
use IO::Select;

$s = IO::Select->new();

$s->add(\*STDIN);
$s->add($some_handle);

@ready = $s->can_read($timeout);

@ready = IO::Select->new(@handles)->read(0);
```

DESCRIPTION

The `IO::Select` package implements an object approach to the system `select` function call. It allows the user to see what IO handles, see *IO::Handle*, are ready for reading, writing or have an error condition pending.

CONSTRUCTOR

new ([HANDLES])

The constructor creates a new object and optionally initialises it with a set of handles.

METHODS

add (HANDLES)

Add the list of handles to the `IO::Select` object. It is these values that will be returned when an event occurs. `IO::Select` keeps these values in a cache which is indexed by the `fileno` of the handle, so if more than one handle with the same `fileno` is specified then only the last one is cached.

Each handle can be an `IO::Handle` object, an integer or an array reference where the first element is a `IO::Handle` or an integer.

remove (HANDLES)

Remove all the given handles from the object. This method also works by the `fileno` of the handles. So the exact handles that were added need not be passed, just handles that have an equivalent `fileno`

exists (HANDLE)

Returns a true value (actually the handle itself) if it is present. Returns undef otherwise.

handles

Return an array of all registered handles.

can_read ([TIMEOUT])

Return an array of handles that are ready for reading. `TIMEOUT` is the maximum amount of time to wait before returning an empty list. If `TIMEOUT` is not given and any handles are registered then the call will block.

can_write ([TIMEOUT])

Same as `can_read` except check for handles that can be written to.

has_error ([TIMEOUT])

Same as `can_read` except check for handles that have an error condition, for example EOF.

count ( )

Returns the number of handles that the object will check for when one of the `can_` methods is called or the object is passed to the `select` static method.

bits()

Return the bit string suitable as argument to the core `select()` call.

`bits()`

Return the bit string suitable as argument to the core `select()` call.

select (READ, WRITE, ERROR [, TIMEOUT])

`select` is a static method, that is you call it with the package name like `new`. `READ`, `WRITE` and `ERROR` are either `undef` or `IO::Select` objects. `TIMEOUT` is optional and has the same effect as for the core select call.

The result will be an array of 3 elements, each a reference to an array which will hold the handles that are ready for reading, writing and have error conditions respectively. Upon error an empty array is returned.

EXAMPLE

Here is a short example which shows how `IO::Select` could be used to write a server which communicates with several sockets while also listening for more connections on a listen socket

```
use IO::Select;
use IO::Socket;

$lsn = new IO::Socket::INET(Listen => 1, LocalPort => 8080);
$sel = new IO::Select( $lsn );

while(@ready = $sel->can_read) {
    foreach $fh (@ready) {
        if($fh == $lsn) {
            # Create a new socket
            $new = $lsn->accept;
            $sel->add($new);
        }
        else {
            # Process socket

            # Maybe we have finished with the socket
            $sel->remove($fh);
            $fh->close;
        }
    }
}
```

AUTHOR

Graham Barr <***Graham.Barr@tiuk.ti.com***>

COPYRIGHT

Copyright (c) 1995 Graham Barr. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

IO::Socket − Object interface to socket communications

SYNOPSIS

```
use IO::Socket;
```

DESCRIPTION

`IO::Socket` provides an object interface to creating and using sockets. It is built upon the *IO::Handle* interface and inherits all the methods defined by *IO::Handle*.

`IO::Socket` only defines methods for those operations which are common to all types of socket. Operations which are specified to a socket in a particular domain have methods defined in sub classes of `IO::Socket`

`IO::Socket` will export all functions (and constants) defined by *Socket*.

CONSTRUCTOR

new ([ARGS])

Creates an `IO::Socket`, which is a reference to a newly created symbol (see the `Symbol` package). `new` optionally takes arguments, these arguments are in key−value pairs. `new` only looks for one key `Domain` which tells new which domain the socket will be in. All other arguments will be passed to the configuration method of the package for that domain, See below.

METHODS

See *perlfunc* for complete descriptions of each of the following supported `IO::Socket` methods, which are just front ends for the corresponding built−in functions:

```
socket
socketpair
bind
listen
accept
send
recv
peername (getpeername)
sockname (getsockname)
```

Some methods take slightly different arguments to those defined in *perlfunc* in attempt to make the interface more flexible. These are

accept([PKG])

perform the system call `accept` on the socket and return a new object. The new object will be created in the same class as the listen socket, unless `PKG` is specified. This object can be used to communicate with the client that was trying to connect. In a scalar context the new socket is returned, or undef upon failure. In an array context a two−element array is returned containing the new socket and the peer address, the list will be empty upon failure.

Additional methods that are provided are

timeout([VAL])

Set or get the timeout value associated with this socket. If called without any arguments then the current setting is returned. If called with an argument the current setting is changed and the previous value returned.

sockopt(OPT [, VAL])

Unified method to both set and get options in the SOL_SOCKET level. If called with one argument then getsockopt is called, otherwise setsockopt is called.

sockdomain

Returns the numerical number for the socket domain type. For example, for a AF_INET socket the value of `&AF_INET` will be returned.

socktype

Returns the numerical number for the socket type. For example, for a SOCK_STREAM socket the value of `&SOCK_STREAM` will be returned.

protocol

Returns the numerical number for the protocol being used on the socket, if known. If the protocol is unknown, as with an AF_UNIX socket, zero is returned.

SUB−CLASSES

IO::Socket::INET

`IO::Socket::INET` provides a constructor to create an AF_INET domain socket and some related methods. The constructor can take the following options

```
PeerAddr    Remote host address          <hostname>[:<port>]
PeerPort    Remote port or service       <service>[(<no>)] | <no>
LocalAddr   Local host bind address      hostname[:port]
LocalPort   Local host bind port         <service>[(<no>)] | <no>
Proto       Protocol name                "tcp" | "udp" | ...
Type        Socket type                  SOCK_STREAM | SOCK_DGRAM | ...
Listen      Queue size for listen
Reuse       Set SO_REUSEADDR before binding
Timeout     Timeout value for various operations
```

If `Listen` is defined then a listen socket is created, else if the socket type, which is derived from the protocol, is SOCK_STREAM then `connect()` is called.

The `PeerAddr` can be a hostname or the IP−address on the "xx.xx.xx.xx" form. The `PeerPort` can be a number or a symbolic service name. The service name might be followed by a number in parenthesis which is used if the service is not known by the system. The `PeerPort` specification can also be embedded in the `PeerAddr` by preceding it with a ":".

Only one of `Type` or `Proto` needs to be specified, one will be assumed from the other. If you specify a symbolic `PeerPort` port, then the constructor will try to derive `Type` and `Proto` from the service name.

Examples:

```
$sock = IO::Socket::INET−>new(PeerAddr => ’www.perl.org’,
                              PeerPort => http(80),
                              Proto    => ’tcp’);

$sock = IO::Socket::INET−>new(PeerAddr => ’localhost:smtp(25)’);

$sock = IO::Socket::INET−>new(Listen    => 5,
                              LocalAddr => ’localhost’,
                              LocalPort => 9000,
                              Proto     => ’tcp’);
```

METHODS

sockaddr ()

Return the address part of the sockaddr structure for the socket

sockport ()

Return the port number that the socket is using on the local host

sockhost ( )

Return the address part of the sockaddr structure for the socket in a text form xx.xx.xx.xx

peeraddr ( )

Return the address part of the sockaddr structure for the socket on the peer host

peerport ( )

Return the port number for the socket on the peer host.

peerhost ( )

Return the address part of the sockaddr structure for the socket on the peer host in a text form xx.xx.xx.xx

IO::Socket::UNIX

`IO::Socket::UNIX` provides a constructor to create an AF_UNIX domain socket and some related methods. The constructor can take the following options

```
    Type        Type of socket (eg SOCK_STREAM or SOCK_DGRAM)
    Local       Path to local fifo
    Peer        Path to peer fifo
    Listen      Create a listen socket
```

METHODS

`hostpath()`

Returns the pathname to the fifo at the local end

`peerpath()`

Returns the pathanme to the fifo at the peer end

SEE ALSO

Socket, *IO::Handle*

AUTHOR

Graham Barr <***Graham.Barr@tiuk.ti.com***>

COPYRIGHT

Copyright (c) 1996 Graham Barr. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

IPC::Open2, open2 − open a process for both reading and writing

SYNOPSIS

```
use IPC::Open2;
$pid = open2(\*RDR, \*WTR, 'some cmd and args');
  # or
$pid = open2(\*RDR, \*WTR, 'some', 'cmd', 'and', 'args');
```

DESCRIPTION

The `open2()` function spawns the given `$cmd` and connects `$rdr` for reading and `$wtr` for writing. It's what you think should work when you try

```
open(HANDLE, "|cmd args|");
```

The write filehandle will have autoflush turned on.

If `$rdr` is a string (that is, a bareword filehandle rather than a glob or a reference) and it begins with `"&"`, then the child will send output directly to that file handle. If `$wtr` is a string that begins with `"<&"`, then WTR will be closed in the parent, and the child will read from it directly. In both cases, there will be a dup(2) instead of a pipe(2) made.

`open2()` returns the process ID of the child process. It doesn't return on failure: it just raises an exception matching `/^open2:/`.

WARNING

It will not create these file handles for you. You have to do this yourself. So don't pass it empty variables expecting them to get filled in for you.

Additionally, this is very dangerous as you may block forever. It assumes it's going to talk to something like **bc**, both writing to it and reading from it. This is presumably safe because you "know" that commands like **bc** will read a line at a time and output a line at a time. Programs like **sort** that read their entire input stream first, however, are quite apt to cause deadlock.

The big problem with this approach is that if you don't have control over source code being run in the the child process, you can't control what it does with pipe buffering. Thus you can't just open a pipe to `cat -v` and continually read and write a line from it.

SEE ALSO

See *IPC::Open3* for an alternative that handles STDERR as well. This function is really just a wrapper around `open3()`.

NAME

IPC::Open3, open3 − open a process for reading, writing, and error handling

SYNOPSIS

```
    $pid = open3(\*WTRFH, \*RDRFH, \*ERRFH
                    'some cmd and args', 'optarg', ...);
```

DESCRIPTION

Extremely similar to `open2()`, `open3()` spawns the given `$cmd` and connects RDRFH for reading, WTRFH for writing, and ERRFH for errors. If ERRFH is ‘’, or the same as RDRFH, then STDOUT and STDERR of the child are on the same file handle. The WTRFH will have autoflush turned on.

If WTRFH begins with "<&`",` then WTRFH will be closed in the parent, and the child will read from it directly. If RDRFH or ERRFH begins with ">&`",` then the child will send output directly to that file handle. In both cases, there will be a dup(2) instead of a pipe(2) made.

If you try to read from the child‘s stdout writer and their stderr writer, you‘ll have problems with blocking, which means you‘ll want to use `select()`, which means you‘ll have to use `sysread()` instead of normal stuff.

`open3()` returns the process ID of the child process. It doesn‘t return on failure: it just raises an exception matching `/^open3:/`.

WARNING

It will not create these file handles for you. You have to do this yourself. So don‘t pass it empty variables expecting them to get filled in for you.

Additionally, this is very dangerous as you may block forever. It assumes it‘s going to talk to something like **bc**, both writing to it and reading from it. This is presumably safe because you "know" that commands like **bc** will read a line at a time and output a line at a time. Programs like **sort** that read their entire input stream first, however, are quite apt to cause deadlock.

The big problem with this approach is that if you don‘t have control over source code being run in the the child process, you can‘t control what it does with pipe buffering. Thus you can‘t just open a pipe to `cat -v` and continually read and write a line from it.

NAME

Math::BigFloat − Arbitrary length float math package

SYNOPSIS

```
use Math::BogFloat;
$f = Math::BigFloat−>new($string);

$f−>fadd(NSTR) return NSTR            addition
$f−>fsub(NSTR) return NSTR            subtraction
$f−>fmul(NSTR) return NSTR            multiplication
$f−>fdiv(NSTR[,SCALE]) returns NSTR   division to SCALE places
$f−>fneg() return NSTR                negation
$f−>fabs() return NSTR                absolute value
$f−>fcmp(NSTR) return CODE            compare undef,<0,=0,>0
$f−>fround(SCALE) return NSTR         round to SCALE digits
$f−>ffround(SCALE) return NSTR        round at SCALEth place
$f−>fnorm() return (NSTR)             normalize
$f−>fsqrt([SCALE]) return NSTR        sqrt to SCALE places
```

DESCRIPTION

All basic math operations are overloaded if you declare your big floats as

```
$float = new Math::BigFloat "2.123123123123123123123123123123123";
```

number format

canonical strings have the form /[+−]\d+E[+−]\d+/ . Input values can have inbedded whitespace.

Error returns ‘NaN’

An input parameter was "Not a Number" or divide by zero or sqrt of negative number.

Division is computed to

`max($div_scale,length(dividend)+length(divisor))` digits by default. Also used for default sqrt scale.

BUGS

The current version of this module is a preliminary version of the real thing that is currently (as of perl5.002) under development.

AUTHOR

Mark Biggar

NAME

Math::BigInt − Arbitrary size integer math package

SYNOPSIS

```
use Math::BigInt;
$i = Math::BigInt−>new($string);

$i−>bneg return BINT               negation
$i−>babs return BINT               absolute value
$i−>bcmp(BINT) return CODE         compare numbers (undef,<0,=0,>0)
$i−>badd(BINT) return BINT         addition
$i−>bsub(BINT) return BINT         subtraction
$i−>bmul(BINT) return BINT         multiplication
$i−>bdiv(BINT) return (BINT,BINT)  division (quo,rem) just quo if scalar
$i−>bmod(BINT) return BINT         modulus
$i−>bgcd(BINT) return BINT         greatest common divisor
$i−>bnorm return BINT              normalization
```

DESCRIPTION

All basic math operations are overloaded if you declare your big integers as

```
$i = new Math::BigInt ’123 456 789 123 456 789’;
```

Canonical notation

Big integer value are strings of the form `/^[+−]\d+$/` with leading zeros suppressed.

Input

Input values to these routines may be strings of the form `/^\s*[+−]?[\d\s]+$/`.

Output

Output values always always in canonical form

Actual math is done in an internal format consisting of an array whose first element is the sign (/^[+−]$/) and whose remaining elements are base 100000 digits with the least significant digit first. The string ‘NaN’ is used to represent the result when input arguments are not numbers, as well as the result of dividing by zero.

EXAMPLES

```
 ’+0’                            canonical zero value
 ’   −123 123 123’               canonical value ’−123123123’
 ’1 23 456 7890’                 canonical value ’+1234567890’
```

BUGS

The current version of this module is a preliminary version of the real thing that is currently (as of perl5.002) under development.

AUTHOR

Mark Biggar, overloaded interface by Ilya Zakharevich.

NAME

Math::Complex − complex numbers and associated mathematical functions

SYNOPSIS

```
use Math::Complex;
$z = Math::Complex−>make(5, 6);
$t = 4 − 3*i + $z;
$j = cplxe(1, 2*pi/3);
```

DESCRIPTION

This package lets you create and manipulate complex numbers. By default, *Perl* limits itself to real numbers, but an extra `use` statement brings full complex support, along with a full set of mathematical functions typically associated with and/or extended to complex numbers.

If you wonder what complex numbers are, they were invented to be able to solve the following equation:

```
x*x = −1
```

and by definition, the solution is noted *i* (engineers use *j* instead since *i* usually denotes an intensity, but the name does not matter). The number *i* is a pure *imaginary* number.

The arithmetics with pure imaginary numbers works just like you would expect it with real numbers... you just have to remember that

```
i*i = −1
```

so you have:

```
5i + 7i = i * (5 + 7) = 12i
4i − 3i = i * (4 − 3) = i
4i * 2i = −8
6i / 2i = 3
1 / i = −i
```

Complex numbers are numbers that have both a real part and an imaginary part, and are usually noted:

```
a + bi
```

where `a` is the *real* part and `b` is the *imaginary* part. The arithmetic with complex numbers is straightforward. You have to keep track of the real and the imaginary parts, but otherwise the rules used for real numbers just apply:

```
(4 + 3i) + (5 − 2i) = (4 + 5) + i(3 − 2) = 9 + i
(2 + i) * (4 − i) = 2*4 + 4i −2i −i*i = 8 + 2i + 1 = 9 + 2i
```

A graphical representation of complex numbers is possible in a plane (also called the *complex plane*, but it‘s really a 2D plane). The number

```
z = a + bi
```

is the point whose coordinates are (a, b). Actually, it would be the vector originating from (0, 0) to (a, b). It follows that the addition of two complex numbers is a vectorial addition.

Since there is a bijection between a point in the 2D plane and a complex number (i.e. the mapping is unique and reciprocal), a complex number can also be uniquely identified with polar coordinates:

```
[rho, theta]
```

where `rho` is the distance to the origin, and `theta` the angle between the vector and the *x* axis. There is a notation for this using the exponential form, which is:

```
rho * exp(i * theta)
```

where *i* is the famous imaginary number introduced above. Conversion between this form and the cartesian form `a + bi` is immediate:

```
        a = rho * cos(theta)
        b = rho * sin(theta)
```

which is also expressed by this formula:

```
        z = rho * exp(i * theta) = rho * (cos theta + i * sin theta)
```

In other words, it‘s the projection of the vector onto the *x* and *y* axes. Mathematicians call *rho* the *norm* or *modulus* and *theta* the *argument* of the complex number. The *norm* of `z` will be noted `abs(z)`.

The polar notation (also known as the trigonometric representation) is much more handy for performing multiplications and divisions of complex numbers, whilst the cartesian notation is better suited for additions and substractions. Real numbers are on the *x* axis, and therefore *theta* is zero.

All the common operations that can be performed on a real number have been defined to work on complex numbers as well, and are merely *extensions* of the operations defined on real numbers. This means they keep their natural meaning when there is no imaginary part, provided the number is within their definition set.

For instance, the `sqrt` routine which computes the square root of its argument is only defined for positive real numbers and yields a positive real number (it is an application from **R+** to **R+**). If we allow it to return a complex number, then it can be extended to negative real numbers to become an application from **R** to **C** (the set of complex numbers):

```
        sqrt(x) = x >= 0 ? sqrt(x) : sqrt(−x)*i
```

It can also be extended to be an application from **C** to **C**, whilst its restriction to **R** behaves as defined above by using the following definition:

```
        sqrt(z = [r,t]) = sqrt(r) * exp(i * t/2)
```

Indeed, a negative real number can be noted `[x,pi]` (the modulus *x* is always positive, so `[x,pi]` is really `−x`, a negative number) and the above definition states that

```
        sqrt([x,pi]) = sqrt(x) * exp(i*pi/2) = [sqrt(x),pi/2] = sqrt(x)*i
```

which is exactly what we had defined for negative real numbers above.

All the common mathematical functions defined on real numbers that are extended to complex numbers share that same property of working *as usual* when the imaginary part is zero (otherwise, it would not be called an extension, would it?).

A *new* operation possible on a complex number that is the identity for real numbers is called the *conjugate*, and is noted with an horizontal bar above the number, or `~z` here.

```
         z = a + bi
        ~z = a − bi
```

Simple... Now look:

```
        z * ~z = (a + bi) * (a − bi) = a*a + b*b
```

We saw that the norm of `z` was noted `abs(z)` and was defined as the distance to the origin, also known as:

```
        rho = abs(z) = sqrt(a*a + b*b)
```

so

```
        z * ~z = abs(z) ** 2
```

If z is a pure real number (i.e. `b == 0`), then the above yields:

```
        a * a = abs(a) ** 2
```

which is true (`abs` has the regular meaning for real number, i.e. stands for the absolute value). This example explains why the norm of `z` is noted `abs(z)`: it extends the `abs` function to complex numbers, yet is the regular `abs` we know when the complex number actually has no imaginary part... This justifies *a posteriori* our use of the `abs` notation for the norm.

OPERATIONS

Given the following notations:

```
z1 = a + bi = r1 * exp(i * t1)
z2 = c + di = r2 * exp(i * t2)
z = <any complex or real number>
```

the following (overloaded) operations are supported on complex numbers:

```
z1 + z2 = (a + c) + i(b + d)
z1 − z2 = (a − c) + i(b − d)
z1 * z2 = (r1 * r2) * exp(i * (t1 + t2))
z1 / z2 = (r1 / r2) * exp(i * (t1 − t2))
z1 ** z2 = exp(z2 * log z1)
~z1 = a − bi
abs(z1) = r1 = sqrt(a*a + b*b)
sqrt(z1) = sqrt(r1) * exp(i * t1/2)
exp(z1) = exp(a) * exp(i * b)
log(z1) = log(r1) + i*t1
sin(z1) = 1/2i (exp(i * z1) − exp(−i * z1))
cos(z1) = 1/2 (exp(i * z1) + exp(−i * z1))
abs(z1) = r1
atan2(z1, z2) = atan(z1/z2)
```

The following extra operations are supported on both real and complex numbers:

```
Re(z) = a
Im(z) = b
arg(z) = t

cbrt(z) = z ** (1/3)
log10(z) = log(z) / log(10)
logn(z, n) = log(z) / log(n)

tan(z) = sin(z) / cos(z)

 csc(z) = 1 / sin(z)
 sec(z) = 1 / cos(z)
cot(z) = 1 / tan(z)

asin(z) = −i * log(i*z + sqrt(1−z*z))
acos(z) = −i * log(z + sqrt(z*z−1))
atan(z) = i/2 * log((i+z) / (i−z))

 acsc(z) = asin(1 / z)
 asec(z) = acos(1 / z)
acot(z) = −i/2 * log((i+z) / (z−i))

sinh(z) = 1/2 (exp(z) − exp(−z))
cosh(z) = 1/2 (exp(z) + exp(−z))
tanh(z) = sinh(z) / cosh(z) = (exp(z) − exp(−z)) / (exp(z) + exp(−z))

 csch(z) = 1 / sinh(z)
 sech(z) = 1 / cosh(z)
coth(z) = 1 / tanh(z)
```

```
asinh(z) = log(z + sqrt(z*z+1))
acosh(z) = log(z + sqrt(z*z−1))
atanh(z) = 1/2 * log((1+z) / (1−z))

 acsch(z) = asinh(1 / z)
 asech(z) = acosh(1 / z)
acoth(z) = atanh(1 / z) = 1/2 * log((1+z) / (z−1))
```

log, *csc*, *cot*, *acsc*, *acot*, *csch*, *coth*, *acosech*, *acotanh*, have aliases *ln*, *cosec*, *cotan*, *acosec*, *acotan*, *cosech*, *cotanh*, *acosech*, *acotanh*, respectively.

The *root* function is available to compute all the *n* roots of some complex, where *n* is a strictly positive integer. There are exactly *n* such roots, returned as a list. Getting the number mathematicians call `j` such that:

```
1 + j + j*j = 0;
```

is a simple matter of writing:

```
$j = ((root(1, 3))[1];
```

The *k*th root for `z = [r,t]` is given by:

```
(root(z, n))[k] = r**(1/n) * exp(i * (t + 2*k*pi)/n)
```

The *spaceship* comparison operator is also defined. In order to ensure its restriction to real numbers is conform to what you would expect, the comparison is run on the real part of the complex number first, and imaginary parts are compared only when the real parts match.

CREATION

To create a complex number, use either:

```
$z = Math::Complex−>make(3, 4);
$z = cplx(3, 4);
```

if you know the cartesian form of the number, or

```
$z = 3 + 4*i;
```

if you like. To create a number using the trigonometric form, use either:

```
$z = Math::Complex−>emake(5, pi/3);
$x = cplxe(5, pi/3);
```

instead. The first argument is the modulus, the second is the angle (in radians, the full circle is 2*pi). (Mnmemonic: `e` is used as a notation for complex numbers in the trigonometric form).

It is possible to write:

```
$x = cplxe(−3, pi/4);
```

but that will be silently converted into `[3,−3pi/4]`, since the modulus must be positive (it represents the distance to the origin in the complex plane).

STRINGIFICATION

When printed, a complex number is usually shown under its cartesian form *a+bi*, but there are legitimate cases where the polar format *[r,t]* is more appropriate.

By calling the routine `Math::Complex::display_format` and supplying either `"polar"` or `"cartesian"`, you override the default display format, which is `"cartesian"`. Not supplying any argument returns the current setting.

This default can be overridden on a per−number basis by calling the `display_format` method instead. As before, not supplying any argument returns the current display format for this number. Otherwise whatever you specify will be the new display format for *this* particular number.

For instance:

```
use Math::Complex;

Math::Complex::display_format('polar');
$j = ((root(1, 3))[1];
print "j = $j\n";               # Prints "j = [1,2pi/3]
$j->display_format('cartesian');
print "j = $j\n";               # Prints "j = -0.5+0.866025403784439i"
```

The polar format attempts to emphasize arguments like *k*pi/n* (where *n* is a positive integer and *k* an integer within [−9,+9]).

USAGE

Thanks to overloading, the handling of arithmetics with complex numbers is simple and almost transparent.

Here are some examples:

```
use Math::Complex;

$j = cplxe(1, 2*pi/3);  # $j ** 3 == 1
print "j = $j, j**3 = ", $j ** 3, "\n";
print "1 + j + j**2 = ", 1 + $j + $j**2, "\n";

$z = -16 + 0*i;                 # Force it to be a complex
print "sqrt($z) = ", sqrt($z), "\n";

$k = exp(i * 2*pi/3);
print "$j - $k = ", $j - $k, "\n";
```

BUGS

Saying `use Math::Complex;` exports many mathematical routines in the caller environment. This is construed as a feature by the Author, actually... ;−)

The code is not optimized for speed, although we try to use the cartesian form for addition−like operators and the trigonometric form for all multiplication−like operators.

The `arg()` routine does not ensure the angle is within the range [−pi,+pi] (a side effect caused by multiplication and division using the trigonometric representation).

All routines expect to be given real or complex numbers. Don‘t attempt to use BigFloat, since Perl has currently no rule to disambiguate a '+' operation (for instance) between two overloaded entities.

AUTHORS

```
Raphael Manfredi <F<Raphael_Manfredi@grenoble.hp.com>>
Jarkko Hietaniemi <F<jhi@iki.fi>>
```

NAME

NDBM_File − Tied access to ndbm files

SYNOPSIS

```
use NDBM_File;

tie(%h, 'NDBM_File', 'Op.dbmx', O_RDWR|O_CREAT, 0640);

untie %h;
```

DESCRIPTION

See *tie*

NAME

Net::Ping − check a remote host for reachability

SYNOPSIS

```
use Net::Ping;

$p = Net::Ping−>new();
print "$host is alive.\n" if $p−>ping($host);
$p−>close();

$p = Net::Ping−>new("icmp");
foreach $host (@host_array)
{
    print "$host is ";
    print "NOT " unless $p−>ping($host, 2);
    print "reachable.\n";
    sleep(1);
}
$p−>close();

$p = Net::Ping−>new("tcp", 2);
while ($stop_time > time())
{
    print "$host not reachable ", scalar(localtime()), "\n"
        unless $p−>ping($host);
    sleep(300);
}
undef($p);

# For backward compatibility
print "$host is alive.\n" if pingecho($host);
```

DESCRIPTION

This module contains methods to test the reachability of remote hosts on a network. A ping object is first created with optional parameters, a variable number of hosts may be pinged multiple times and then the connection is closed.

You may choose one of three different protocols to use for the ping. With the "tcp" protocol the `ping()` method attempts to establish a connection to the remote host‘s echo port. If the connection is successfully established, the remote host is considered reachable. No data is actually echoed. This protocol does not require any special privileges but has higher overhead than the other two protocols.

Specifying the "udp" protocol causes the `ping()` method to send a udp packet to the remote host‘s echo port. If the echoed packet is received from the remote host and the received packet contains the same data as the packet that was sent, the remote host is considered reachable. This protocol does not require any special privileges.

If the "icmp" protocol is specified, the `ping()` method sends an icmp echo message to the remote host, which is what the UNIX ping program does. If the echoed message is received from the remote host and the echoed information is correct, the remote host is considered reachable. Specifying the "icmp" protocol requires that the program be run as root or that the program be setuid to root.

Functions

Net::Ping−new([`$proto` [, `$def_timeout` [, `$bytes`]]]);

Create a new ping object. All of the parameters are optional. `$proto` specifies the protocol to use when doing a ping. The current choices are "tcp", "udp" or "icmp". The default is "udp".

If a default timeout (`$def_timeout`) in seconds is provided, it is used when a timeout is not given to the `ping()` method (below). The timeout must be greater than 0 and the default, if not specified, is 5 seconds.

If the number of data bytes (`$bytes)` is given, that many data bytes are included in the ping packet sent to the remote host. The number of data bytes is ignored if the protocol is "tcp". The minimum (and default) number of data bytes is 1 if the protocol is "udp" and 0 otherwise. The maximum number of data bytes that can be specified is 1024.

$p−ping($host [, `$timeout]);`

Ping the remote host and wait for a response. `$host` can be either the hostname or the IP number of the remote host. The optional timeout must be greater than 0 seconds and defaults to whatever was specified when the ping object was created. If the hostname cannot be found or there is a problem with the IP number, undef is returned. Otherwise, 1 is returned if the host is reachable and 0 if it is not. For all practical purposes, undef and 0 and can be treated as the same case.

$p−close();

Close the network connection for this ping object. The network connection is also closed by "undef `$p".` The network connection is automatically closed if the ping object goes out of scope (e.g. `$p` is local to a subroutine and you leave the subroutine).

pingecho(`$host` [, `$timeout]);`

To provide backward compatibility with the previous version of Net::Ping, a `pingecho()` subroutine is available with the same functionality as before. `pingecho()` uses the tcp protocol. The return values and parameters are the same as described for the `ping()` method. This subroutine is obsolete and may be removed in a future version of Net::Ping.

WARNING

`pingecho()` or a ping object with the tcp protocol use `alarm()` to implement the timeout. So, don‘t use `alarm()` in your program while you are using `pingecho()` or a ping object with the tcp protocol. The udp and icmp protocols do not use `alarm()` to implement the timeout.

NOTES

There will be less network overhead (and some efficiency in your program) if you specify either the udp or the icmp protocol. The tcp protocol will generate 2.5 times or more traffic for each ping than either udp or icmp. If many hosts are pinged frequently, you may wish to implement a small wait (e.g. 25ms or more) between each ping to avoid flooding your network with packets.

The icmp protocol requires that the program be run as root or that it be setuid to root. The tcp and udp protocols do not require special privileges, but not all network devices implement the echo protocol for tcp or udp.

Local hosts should normally respond to pings within milliseconds. However, on a very congested network it may take up to 3 seconds or longer to receive an echo packet from the remote host. If the timeout is set too low under these conditions, it will appear that the remote host is not reachable (which is almost the truth).

Reachability doesn‘t necessarily mean that the remote host is actually functioning beyond its ability to echo packets.

Because of a lack of anything better, this module uses its own routines to pack and unpack ICMP packets. It would be better for a separate module to be written which understands all of the different kinds of ICMP packets.

NAME

Net::hostent − by−name interface to Perl‘s built−in `gethost*()` functions

SYNOPSIS

```
 use Net::hostnet;
```

DESCRIPTION

This module‘s default exports override the core `gethostbyname()` and `gethostbyaddr()` functions, replacing them with versions that return "Net::hostent" objects. This object has methods that return the similarly named structure field name from the C‘s hostent structure from ***netdb.h***; namely name, aliases, addrtype, length, and addresses. The aliases and addresses methods return array reference, the rest scalars. The addr method is equivalent to the zeroth element in the addresses array reference.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `h_`. Thus, `$host_obj−>name()` corresponds to `$h_name` if you import the fields. Array references are available as regular array variables, so for example `@{ $host_obj−>aliases() }` would be simply @h_aliases.

The `gethost()` funtion is a simple front−end that forwards a numeric argument to `gethostbyaddr()` by way of Socket::inet_aton, and the rest to `gethostbyname()`.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built−ins are still available via the `CORE::` pseudo−package.

EXAMPLES

```
 use Net::hostent;
 use Socket;

 @ARGV = ('netscape.com') unless @ARGV;

 for $host ( @ARGV ) {

    unless ($h = gethost($host)) {
        warn "$0: no such host: $host\n";
        next;
    }

    printf "\n%s is %s%s\n",
            $host,
            lc($h−>name) eq lc($host) ? "" : "*really* ",
            $h−>name;

    print "\taliases are ", join(", ", @{$h−>aliases}), "\n"
                if @{$h−>aliases};

    if ( @{$h−>addr_list} > 1 ) {
        my $i;
        for $addr ( @{$h−>addr_list} ) {
            printf "\taddr #%d is [%s]\n", $i++, inet_ntoa($addr);
        }
    } else {
        printf "\taddress is [%s]\n", inet_ntoa($h−>addr);
    }

    if ($h = gethostbyaddr($h−>addr)) {
        if (lc($h−>name) ne lc($host)) {
            printf "\tThat addr reverses to host %s!\n", $h−>name;
            $host = $h−>name;
```

```
                    redo;
                }
            }
        }
```

NOTE

While this class is currently implemented using the Class::Template module to build a struct−like class, you shouldn‘t rely upon this.

AUTHOR

Tom Christiansen

NAME

Net::netent − by−name interface to Perl‘s built−in `getnet*()` functions

SYNOPSIS

```
use Net::netent qw(:FIELDS);
getnetbyname("loopback")               or die "bad net";
printf "%s is %08X\n", $n_name, $n_net;

use Net::netent;

$n = getnetbyname("loopback")          or die "bad net";
{ # there’s gotta be a better way, eh?
    @bytes = unpack("C4", pack("N", $n−>net));
    shift @bytes while @bytes && $bytes[0] == 0;
}
printf "%s is %08X [%d.%d.%d.%d]\n", $n−>name, $n−>net, @bytes;
```

DESCRIPTION

This module‘s default exports override the core `getnetbyname()` and `getnetbyaddr()` functions, replacing them with versions that return "Net::netent" objects. This object has methods that return the similarly named structure field name from the C‘s netent structure from ***netdb.h***; namely name, aliases, addrtype, and net. The aliases method returns an array reference, the rest scalars.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `n_`. Thus, `$net_obj->name()` corresponds to `$n_name` if you import the fields. Array references are available as regular array variables, so for example `@{ $net_obj->aliases() }` would be simply @n_aliases.

The `getnet()` funtion is a simple front−end that forwards a numeric argument to `getnetbyaddr()`, and the rest to `getnetbyname()`.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built−ins are still available via the `CORE::` pseudo−package.

EXAMPLES

The `getnet()` functions do this in the Perl core:

```
    sv_setiv(sv, (I32)nent−>n_net);
```

The `gethost()` functions do this in the Perl core:

```
    sv_setpvn(sv, hent−>h_addr, len);
```

That means that the address comes back in binary for the host functions, and as a regular perl integer for the net ones. This seems a bug, but here‘s how to deal with it:

```
use strict;
use Socket;
use Net::netent;

@ARGV = (’loopback’) unless @ARGV;

my($n, $net);

for $net ( @ARGV ) {

    unless ($n = getnetbyname($net)) {
       warn "$0: no such net: $net\n";
       next;
    }
```

```
            printf "\n%s is %s%s\n",
                   $net,
                   lc($n->name) eq lc($net) ? "" : "*really* ",
                   $n->name;

            print "\taliases are ", join(", ", @{$n->aliases}), "\n"
                       if @{$n->aliases};

            # this is stupid; first, why is this not in binary?
            # second, why am i going through these convolutions
            # to make it looks right
            {
               my @a = unpack("C4", pack("N", $n->net));
               shift @a while @a && $a[0] == 0;
               printf "\taddr is %s [%d.%d.%d.%d]\n", $n->net, @a;
            }

            if ($n = getnetbyaddr($n->net)) {
               if (lc($n->name) ne lc($net)) {
                   printf "\tThat addr reverses to net %s!\n", $n->name;
                   $net = $n->name;
                   redo;
               }
            }
        }
```

NOTE

While this class is currently implemented using the Class::Template module to build a struct−like class, you shouldn‘t rely upon this.

AUTHOR

Tom Christiansen

NAME

Net::protoent − by−name interface to Perl‘s built−in `getproto*()` functions

SYNOPSIS

```
use Net::protoent;
$p = getprotobyname(shift || ’tcp’) || die "no proto";
printf "proto for %s is %d, aliases are %s\n",
   $p−>name, $p−>proto, "@{$p−>aliases}";

use Net::protoent qw(:FIELDS);
getprotobyname(shift || ’tcp’) || die "no proto";
print "proto for $p_name is $p_proto, aliases are @p_aliases\n";
```

DESCRIPTION

This module‘s default exports override the core `getprotoent()`, `getprotobyname()`, and `getnetbyport()` functions, replacing them with versions that return "Net::protoent" objects. They take default second arguments of "tcp". This object has methods that return the similarly named structure field name from the C‘s protoent structure from ***netdb.h***; namely name, aliases, and proto. The aliases method returns an array reference, the rest scalars.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `p_`. Thus, `$proto_obj−>name()` corresponds to `$p_name` if you import the fields. Array references are available as regular array variables, so for example `@{ $proto_obj−>aliases() }` would be simply @p_aliases.

The `getproto()` function is a simple front−end that forwards a numeric argument to `getprotobyport()`, and the rest to `getprotobyname()`.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built−ins are still available via the `CORE::` pseudo−package.

NOTE

While this class is currently implemented using the Class::Template module to build a struct−like class, you shouldn‘t rely upon this.

AUTHOR

Tom Christiansen

NAME

Net::servent − by−name interface to Perl‘s built−in `getserv*()` functions

SYNOPSIS

```
use Net::servent;
$s = getservbyname(shift || ’ftp’) || die "no service";
printf "port for %s is %s, aliases are %s\n",
   $s−>name, $s−>port, "@{$s−>aliases}";

use Net::servent qw(:FIELDS);
getservbyname(shift || ’ftp’) || die "no service";
print "port for $s_name is $s_port, aliases are @s_aliases\n";
```

DESCRIPTION

This module‘s default exports override the core `getservent()`, `getservbyname()`, and `getnetbyport()` functions, replacing them with versions that return "Net::servent" objects. They take default second arguments of "tcp". This object has methods that return the similarly named structure field name from the C‘s servent structure from ***netdb.h***; namely name, aliases, port, and proto. The aliases method returns an array reference, the rest scalars.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `n_`. Thus, `$serv_obj−>name()` corresponds to `$s_name` if you import the fields. Array references are available as regular array variables, so for example `@{ $serv_obj−>aliases() }` would be simply @s_aliases.

The `getserv()` function is a simple front−end that forwards a numeric argument to `getservbyport()`, and the rest to `getservbyname()`.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built−ins are still available via the `CORE::` pseudo−package.

EXAMPLES

```
use Net::servent qw(:FIELDS);

while (@ARGV) {
    my ($service, $proto) = ((split m!/!, shift), ’tcp’);
    my $valet = getserv($service, $proto);
    unless ($valet) {
        warn "$0: No service: $service/$proto\n"
        next;
    }
    printf "service $service/$proto is port %d\n", $valet−>port;
    print "alias are @s_aliases\n" if @s_aliases;
}
```

NOTE

While this class is currently implemented using the Class::Template module to build a struct−like class, you shouldn‘t rely upon this.

AUTHOR

Tom Christiansen

NAME

ODBM_File − Tied access to odbm files

SYNOPSIS

```
 use ODBM_File;

 tie(%h, 'ODBM_File', 'Op.dbmx', O_RDWR|O_CREAT, 0640);

 untie %h;
```

DESCRIPTION

See *tie*

NAME

Opcode − Disable named opcodes when compiling perl code

SYNOPSIS

```
    use Opcode;
```

DESCRIPTION

Perl code is always compiled into an internal format before execution.

Evaluating perl code (e.g. via "eval" or "do 'file'") causes the code to be compiled into an internal format and then, provided there was no error in the compilation, executed. The internal format is based on many distinct *opcodes*.

By default no opmask is in effect and any code can be compiled.

The Opcode module allow you to define an *operator mask* to be in effect when perl *next* compiles any code. Attempting to compile code which contains a masked opcode will cause the compilation to fail with an error. The code will not be executed.

NOTE

The Opcode module is not usually used directly. See the ops pragma and Safe modules for more typical uses.

WARNING

The authors make **no warranty**, implied or otherwise, about the suitability of this software for safety or security purposes.

The authors shall not in any case be liable for special, incidental, consequential, indirect or other similar damages arising from the use of this software.

Your mileage will vary. If in any doubt **do not use it**.

Operator Names and Operator Lists

The canonical list of operator names is the contents of the array op_name defined and initialised in file ***opcode.h*** of the Perl source distribution (and installed into the perl library).

Each operator has both a terse name (its opname) and a more verbose or recognisable descriptive name. The opdesc function can be used to return a list of descriptions for a list of operators.

Many of the functions and methods listed below take a list of operators as parameters. Most operator lists can be made up of several types of element. Each element can be one of

an operator name (opname)

Operator names are typically small lowercase words like enterloop, leaveloop, last, next, redo etc. Sometimes they are rather cryptic like gv2cv, i_ncmp and ftsvtx.

an operator tag name (optag)

Operator tags can be used to refer to groups (or sets) of operators. Tag names always being with a colon. The Opcode module defines several optags and the user can define others using the define_optag function.

a negated opname or optag

An opname or optag can be prefixed with an exclamation mark, e.g., !mkdir. Negating an opname or optag means remove the corresponding ops from the accumulated set of ops at that point.

an operator set (opset)

An *opset* as a binary string of approximately 43 bytes which holds a set or zero or more operators.

The opset and opset_to_ops functions can be used to convert from a list of operators to an opset and *vice versa*.

Wherever a list of operators can be given you can use one or more opsets. See also Manipulating Opsets below.

Opcode Functions

The Opcode package contains functions for manipulating operator names tags and sets. All are available for export by the package.

opcodes
: In a scalar context opcodes returns the number of opcodes in this version of perl (around 340 for perl5.002).

  In a list context it returns a list of all the operator names. (Not yet implemented, use @names = opset_to_ops(full_opset).)

opset (OP, ...)
: Returns an opset containing the listed operators.

opset_to_ops (OPSET)
: Returns a list of operator names corresponding to those operators in the set.

opset_to_hex (OPSET)
: Returns a string representation of an opset. Can be handy for debugging.

full_opset
: Returns an opset which includes all operators.

empty_opset
: Returns an opset which contains no operators.

invert_opset (OPSET)
: Returns an opset which is the inverse set of the one supplied.

verify_opset (OPSET, ...)
: Returns true if the supplied opset looks like a valid opset (is the right length etc) otherwise it returns false. If an optional second parameter is true then verify_opset will croak on an invalid opset instead of returning false.

  Most of the other Opcode functions call verify_opset automatically and will croak if given an invalid opset.

define_optag (OPTAG, OPSET)
: Define OPTAG as a symbolic name for OPSET. Optag names always start with a colon `:`.

  The optag name used must not be defined already (define_optag will croak if it is already defined). Optag names are global to the perl process and optag definitions cannot be altered or deleted once defined.

  It is strongly recommended that applications using Opcode should use a leading capital letter on their tag names since lowercase names are reserved for use by the Opcode module. If using Opcode within a module you should prefix your tags names with the name of your module to ensure uniqueness and thus avoid clashes with other modules.

opmask_add (OPSET)
: Adds the supplied opset to the current opmask. Note that there is currently *no* mechanism for unmasking ops once they have been masked. This is intentional.

opmask
: Returns an opset corresponding to the current opmask.

opdesc (OP, ...)
: This takes a list of operator names and returns the corresponding list of operator descriptions.

opdump (PAT)

Dumps to STDOUT a two column list of op names and op descriptions. If an optional pattern is given then only lines which match the (case insensitive) pattern will be output.

It‘s designed to be used as a handy command line utility:

```
perl −MOpcode=opdump −e opdump
perl −MOpcode=opdump −e ’opdump Eval’
```

Manipulating Opsets

Opsets may be manipulated using the perl bit vector operators & (and), | (or), ^ (xor) and ~ (negate/invert).

However you should never rely on the numerical position of any opcode within the opset. In other words both sides of a bit vector operator should be opsets returned from Opcode functions.

Also, since the number of opcodes in your current version of perl might not be an exact multiple of eight, there may be unused bits in the last byte of an upset. This should not cause any problems (Opcode functions ignore those extra bits) but it does mean that using the ~ operator will typically not produce the same ‘physical’ opset ‘string’ as the invert_opset function.

TO DO (maybe)

```
$bool = opset_eq($opset1, $opset2)  true if opsets are logically eqiv

$yes = opset_can($opset, @ops)      true if $opset has all @ops set

@diff = opset_diff($opset1, $opset2) => (’foo’, ’!bar’, ...)
```

Predefined Opcode Tags

:base_core

```
null stub scalar pushmark wantarray const defined undef

rv2sv sassign

rv2av aassign aelem aelemfast aslice av2arylen

rv2hv helem hslice each values keys exists delete

preinc i_preinc predec i_predec postinc i_postinc postdec i_postdec
int hex oct abs pow multiply i_multiply divide i_divide
modulo i_modulo add i_add subtract i_subtract

left_shift right_shift bit_and bit_xor bit_or negate i_negate
not complement

lt i_lt gt i_gt le i_le ge i_ge eq i_eq ne i_ne ncmp i_ncmp
slt sgt sle sge seq sne scmp

substr vec stringify study pos length index rindex ord chr

ucfirst lcfirst uc lc quotemeta trans chop schop chomp schomp

match split

list lslice splice push pop shift unshift reverse

cond_expr flip flop andassign orassign and or xor

warn die lineseq nextstate unstack scope enter leave

rv2cv anoncode prototype

entersub leavesub return method −− XXX loops via recursion?

leaveeval −− needed for Safe to operate, is safe without entereval
```

:base_mem

These memory related ops are not included in :base_core because they can easily be used to implement a resource attack (e.g., consume all available memory).

```
concat repeat join range

anonlist anonhash
```

Note that despite the existance of this optag a memory resource attack may still be possible using only :base_core ops.

Disabling these ops is a *very* heavy handed way to attempt to prevent a memory resource attack. It‘s probable that a specific memory limit mechanism will be added to perl in the near future.

:base_loop

These loop ops are not included in :base_core because they can easily be used to implement a resource attack (e.g., consume all available CPU time).

```
grepstart grepwhile
mapstart mapwhile
enteriter iter
enterloop leaveloop
last next redo
goto
```

:base_io

These ops enable *filehandle* (rather than filename) based input and output. These are safe on the assumption that only pre−existing filehandles are available for use. To create new filehandles other ops such as open would need to be enabled.

```
readline rcatline getc read

formline enterwrite leavewrite

print sysread syswrite send recv eof tell seek

readdir telldir seekdir rewinddir
```

:base_orig

These are a hotchpotch of opcodes still waiting to be considered

```
gvsv gv gelem

padsv padav padhv padany

rv2gv refgen srefgen ref

bless −− could be used to change ownership of objects (reblessing)

pushre regcmaybe regcomp subst substcont

sprintf prtf −− can core dump

crypt

tie untie

dbmopen dbmclose
sselect select
pipe_op sockpair

getppid getpgrp setpgrp getpriority setpriority localtime gmtime

entertry leavetry −− can be used to ’hide’ fatal errors
```

:base_math

These ops are not included in :base_core because of the risk of them being used to generate floating point exceptions (which would have to be caught using a `$SIG{FPE}` handler).

```
atan2 sin cos exp log sqrt
```

These ops are not included in :base_core because they have an effect beyond the scope of the compartment.

```
rand srand
```

:default

A handy tag name for a *reasonable* default set of ops. (The current ops allowed are unstable while development continues. It will change.)

```
:base_core :base_mem :base_loop :base_io :base_orig
```

If safety matters to you (and why else would you be using the Opcode module?) then you should not rely on the definition of this, or indeed any other, optag!

:filesys_read

```
stat lstat readlink

ftatime ftblk ftchr ftctime ftdir fteexec fteowned fteread
ftewrite ftfile ftis ftlink ftmtime ftpipe ftrexec ftrowned
ftrread ftsgid ftsize ftsock ftsuid fttty ftzero ftrwrite ftsvtx

fttext ftbinary

fileno
```

:sys_db

```
ghbyname ghbyaddr ghostent shostent ehostent      -- hosts
gnbyname gnbyaddr gnetent snetent enetent         -- networks
gpbyname gpbynumber gprotoent sprotoent eprotoent -- protocols
gsbyname gsbyport gservent sservent eservent      -- services

gpwnam gpwuid gpwent spwent epwent getlogin       -- users
ggrnam ggrgid ggrent sgrent egrent                -- groups
```

:browse

A handy tag name for a *reasonable* default set of ops beyond the :default optag. Like :default (and indeed all the other optags) its current definition is unstable while development continues. It will change.

The :browse tag represents the next step beyond :default. It it a superset of the :default ops and adds :filesys_read the :sys_db. The intent being that scripts can access more (possibly sensitive) information about your system but not be able to change it.

```
:default :filesys_read :sys_db
```

:filesys_open

```
sysopen open close
umask binmode

open_dir closedir -- other dir ops are in :base_io
```

:filesys_write

```
link unlink rename symlink truncate

mkdir rmdir
```

```
utime chmod chown

fcntl -- not strictly filesys related, but possibly as dangerous?
```

:subprocess

```
backtick system

fork

wait waitpid

glob -- access to Cshell via <‘rm *‘>
```

:ownprocess

```
exec exit kill

time tms -- could be used for timing attacks (paranoid?)
```

:others

This tag holds groups of assorted specialist opcodes that don‘t warrant having optags defined for them.

SystemV Interprocess Communications:

```
msgctl msgget msgrcv msgsnd

semctl semget semop

shmctl shmget shmread shmwrite
```

:still_to_be_decided

```
chdir
flock ioctl

socket getpeername ssockopt
bind connect listen accept shutdown gsockopt getsockname

sleep alarm -- changes global timer state and signal handling
sort -- assorted problems including core dumps
tied -- can be used to access object implementing a tie
pack unpack -- can be used to create/use memory pointers

entereval -- can be used to hide code from initial compile
require dofile

caller -- get info about calling environment and args

reset

dbstate -- perl -d version of nextstate(ment) opcode
```

:dangerous

This tag is simply a bucket for opcodes that are unlikely to be used via a tag name but need to be tagged for completness and documentation.

```
syscall dump chroot
```

SEE ALSO

ops(3) — perl pragma interface to Opcode module.

Safe(3) — Opcode and namespace limited execution compartments

AUTHORS

Originally designed and implemented by Malcolm Beattie, mbeattie@sable.ox.ac.uk as part of Safe version 1.

Split out from Safe module version 1, named opcode tags and other changes added by Tim Bunce <***Tim.Bunce@ig.co.uk***>.

NAME

ops − Perl pragma to restrict unsafe operations when compiling

SYNOPSIS

```
  perl −Mops=:default ...    # only allow reasonably safe operations

  perl −M−ops=system ...     # disable the ’system’ opcode
```

DESCRIPTION

Since the ops pragma currently has an irreversable global effect, it is only of significant practical use with the `−M` option on the command line.

See the *Opcode* module for information about opcodes, optags, opmasks and important information about safety.

SEE ALSO

Opcode(3), Safe(3), perlrun(3)

NAME

POSIX − Perl interface to IEEE Std 1003.1

SYNOPSIS

```
use POSIX;
use POSIX qw(setsid);
use POSIX qw(:errno_h :fcntl_h);

printf "EINTR is %d\n", EINTR;

$sess_id = POSIX::setsid();

$fd = POSIX::open($path, O_CREAT|O_EXCL|O_WRONLY, 0644);
    # note: that’s a filedescriptor, *NOT* a filehandle
```

DESCRIPTION

The POSIX module permits you to access all (or nearly all) the standard POSIX 1003.1 identifiers. Many of these identifiers have been given Perl−ish interfaces. Things which are `#defines` in C, like EINTR or O_NDELAY, are automatically exported into your namespace. All functions are only exported if you ask for them explicitly. Most likely people will prefer to use the fully−qualified function names.

This document gives a condensed list of the features available in the POSIX module. Consult your operating system‘s manpages for general information on most features. Consult *perlfunc* for functions which are noted as being identical to Perl‘s builtin functions.

The first section describes POSIX functions from the 1003.1 specification. The second section describes some classes for signal objects, TTY objects, and other miscellaneous objects. The remaining sections list various constants and macros in an organization which roughly follows IEEE Std 1003.1b−1993.

NOTE

The POSIX module is probably the most complex Perl module supplied with the standard distribution. It incorporates autoloading, namespace games, and dynamic loading of code that‘s in Perl, C, or both. It‘s a great source of wisdom.

CAVEATS

A few functions are not implemented because they are C specific. If you attempt to call these, they will print a message telling you that they aren‘t implemented, and suggest using the Perl equivalent should one exist. For example, trying to access the `setjmp()` call will elicit the message "`setjmp()` is C−specific: use eval {} instead".

Furthermore, some evil vendors will claim 1003.1 compliance, but in fact are not so: they will not pass the PCTS (POSIX Compliance Test Suites). For example, one vendor may not define EDEADLK, or the semantics of the errno values set by open(2) might not be quite right. Perl does not attempt to verify POSIX compliance. That means you can currently successfully say "use POSIX", and then later in your program you find that your vendor has been lax and there‘s no usable ICANON macro after all. This could be construed to be a bug.

FUNCTIONS

_exit
This is identical to the C function `_exit()`.

abort
This is identical to the C function `abort()`.

abs
This is identical to Perl‘s builtin `abs()` function.

access
Determines the accessibility of a file.

```
if( POSIX::access( "/", &POSIX::R_OK ) ){
        print "have read permission\n";
}
```

Returns `undef` on failure.

acos	This is identical to the C function `acos()`.
alarm	This is identical to Perl's builtin `alarm()` function.
asctime	This is identical to the C function `asctime()`.
asin	This is identical to the C function `asin()`.
assert	Unimplemented.
atan	This is identical to the C function `atan()`.
atan2	This is identical to Perl's builtin `atan2()` function.
atexit	`atexit()` is C−specific: use END {} instead.
atof	`atof()` is C−specific.
atoi	`atoi()` is C−specific.
atol	`atol()` is C−specific.
bsearch	`bsearch()` not supplied.
calloc	`calloc()` is C−specific.
ceil	This is identical to the C function `ceil()`.
chdir	This is identical to Perl's builtin `chdir()` function.
chmod	This is identical to Perl's builtin `chmod()` function.
chown	This is identical to Perl's builtin `chown()` function.
clearerr	Use method `IO::Handle::clearerr()` instead.
clock	This is identical to the C function `clock()`.

close
Close the file. This uses file descriptors such as those obtained by calling `POSIX::open`.

```
        $fd = POSIX::open( "foo", &POSIX::O_RDONLY );
        POSIX::close( $fd );
```

Returns `undef` on failure.

closedir
This is identical to Perl's builtin `closedir()` function.

cos
This is identical to Perl's builtin `cos()` function.

cosh
This is identical to the C function `cosh()`.

creat
Create a new file. This returns a file descriptor like the ones returned by `POSIX::open`. Use `POSIX::close` to close the file.

```
        $fd = POSIX::creat( "foo", 0611 );
        POSIX::close( $fd );
```

ctermid
Generates the path name for the controlling terminal.

```
        $path = POSIX::ctermid();
```

ctime
This is identical to the C function `ctime()`.

cuserid
Get the character login name of the user.

```
        $name = POSIX::cuserid();
```

difftime This is identical to the C function `difftime()`.

div `div()` is C−specific.

dup This is similar to the C function `dup()`.

This uses file descriptors such as those obtained by calling `POSIX::open`.

Returns `undef` on failure.

dup2 This is similar to the C function `dup2()`.

This uses file descriptors such as those obtained by calling `POSIX::open`.

Returns `undef` on failure.

errno Returns the value of errno.

```
        $errno = POSIX::errno();
```

execl `execl()` is C−specific.

execle `execle()` is C−specific.

execlp `execlp()` is C−specific.

execv `execv()` is C−specific.

execve `execve()` is C−specific.

execvp `execvp()` is C−specific.

exit This is identical to Perl‘s builtin `exit()` function.

exp This is identical to Perl‘s builtin `exp()` function.

fabs This is identical to Perl‘s builtin `abs()` function.

fclose Use method `IO::Handle::close()` instead.

fcntl This is identical to Perl‘s builtin `fcntl()` function.

fdopen Use method `IO::Handle::new_from_fd()` instead.

feof Use method `IO::Handle::eof()` instead.

ferror Use method `IO::Handle::error()` instead.

fflush Use method `IO::Handle::flush()` instead.

fgetc Use method `IO::Handle::getc()` instead.

fgetpos Use method `IO::Seekable::getpos()` instead.

fgets Use method `IO::Handle::gets()` instead.

fileno Use method `IO::Handle::fileno()` instead.

floor This is identical to the C function `floor()`.

fmod This is identical to the C function `fmod()`.

fopen Use method `IO::File::open()` instead.

fork This is identical to Perl‘s builtin `fork()` function.

fpathconf Retrieves the value of a configurable limit on a file or directory. This uses file descriptors such as those obtained by calling `POSIX::open`.

The following will determine the maximum length of the longest allowable pathname on the filesystem which holds `/tmp/foo`.

```
        $fd = POSIX::open( "/tmp/foo", &POSIX::O_RDONLY );
        $path_max = POSIX::fpathconf( $fd, &POSIX::_PC_PATH_MAX );
```

Returns `undef` on failure.

fprintf `fprintf()` is C−specific—use printf instead.

fputc `fputc()` is C−specific—use print instead.

fputs `fputs()` is C−specific—use print instead.

fread `fread()` is C−specific—use read instead.

free `free()` is C−specific.

freopen `freopen()` is C−specific—use open instead.

frexp Return the mantissa and exponent of a floating−point number.

```
        ($mantissa, $exponent) = POSIX::frexp( 3.14 );
```

fscanf `fscanf()` is C−specific—use < and regular expressions instead.

fseek Use method `IO::Seekable::seek()` instead.

fsetpos Use method `IO::Seekable::setpos()` instead.

fstat Get file status. This uses file descriptors such as those obtained by calling `POSIX::open`. The data returned is identical to the data from Perl‘s builtin `stat` function.

```
        $fd = POSIX::open( "foo", &POSIX::O_RDONLY );
        @stats = POSIX::fstat( $fd );
```

ftell Use method `IO::Seekable::tell()` instead.

fwrite `fwrite()` is C−specific—use print instead.

getc This is identical to Perl‘s builtin `getc()` function.

getchar Returns one character from STDIN.

getcwd Returns the name of the current working directory.

getegid Returns the effective group id.

getenv Returns the value of the specified enironment variable.

geteuid Returns the effective user id.

getgid Returns the user‘s real group id.

getgrgid This is identical to Perl‘s builtin `getgrgid()` function.

getgrnam This is identical to Perl‘s builtin `getgrnam()` function.

getgroups
Returns the ids of the user‘s supplementary groups.

getlogin This is identical to Perl‘s builtin `getlogin()` function.

getpgrp This is identical to Perl‘s builtin `getpgrp()` function.

getpid Returns the process‘s id.

getppid This is identical to Perl‘s builtin `getppid()` function.

getpwnam
This is identical to Perl‘s builtin `getpwnam()` function.

getpwuid
: This is identical to Perl‘s builtin `getpwuid()` function.

gets
: Returns one line from STDIN.

getuid
: Returns the user‘s id.

gmtime
: This is identical to Perl‘s builtin `gmtime()` function.

isalnum
: This is identical to the C function, except that it can apply to a single character or to a whole string.

isalpha
: This is identical to the C function, except that it can apply to a single character or to a whole string.

isatty
: Returns a boolean indicating whether the specified filehandle is connected to a tty.

iscntrl
: This is identical to the C function, except that it can apply to a single character or to a whole string.

isdigit
: This is identical to the C function, except that it can apply to a single character or to a whole string.

isgraph
: This is identical to the C function, except that it can apply to a single character or to a whole string.

islower
: This is identical to the C function, except that it can apply to a single character or to a whole string.

isprint
: This is identical to the C function, except that it can apply to a single character or to a whole string.

ispunct
: This is identical to the C function, except that it can apply to a single character or to a whole string.

isspace
: This is identical to the C function, except that it can apply to a single character or to a whole string.

isupper
: This is identical to the C function, except that it can apply to a single character or to a whole string.

isxdigit
: This is identical to the C function, except that it can apply to a single character or to a whole string.

kill
: This is identical to Perl‘s builtin `kill()` function.

labs
: `labs()` is C−specific, use abs instead.

ldexp
: This is identical to the C function `ldexp()`.

ldiv
: `ldiv()` is C−specific, use / and int instead.

link
: This is identical to Perl‘s builtin `link()` function.

localeconv
: Get numeric formatting information. Returns a reference to a hash containing the current locale formatting values.

The database for the **de** (Deutsch or German) locale.

```
        $loc = POSIX::setlocale( &POSIX::LC_ALL, "de" );
        print "Locale = $loc\n";
        $lconv = POSIX::localeconv();
        print "decimal_point    = ", $lconv->{decimal_point},   "\n";
        print "thousands_sep    = ", $lconv->{thousands_sep},   "\n";
        print "grouping = ", $lconv->{grouping},"\n";
        print "int_curr_symbol  = ", $lconv->{int_curr_symbol}, "\n";
        print "currency_symbol  = ", $lconv->{currency_symbol}, "\n";
        print "mon_decimal_point = ", $lconv->{mon_decimal_point}, "\n";
        print "mon_thousands_sep = ", $lconv->{mon_thousands_sep}, "\n";
        print "mon_grouping     = ", $lconv->{mon_grouping}, "\n";
        print "positive_sign = ", $lconv->{positive_sign}, "\n";
        print "negative_sign    = ", $lconv->{negative_sign}, "\n";
        print "int_frac_digits  = ", $lconv->{int_frac_digits}, "\n";
        print "frac_digits = ", $lconv->{frac_digits}, "\n";
        print "p_cs_precedes    = ", $lconv->{p_cs_precedes},   "\n";
        print "p_sep_by_space   = ", $lconv->{p_sep_by_space},  "\n";
        print "n_cs_precedes    = ", $lconv->{n_cs_precedes},   "\n";
        print "n_sep_by_space   = ", $lconv->{n_sep_by_space},  "\n";
        print "p_sign_posn = ", $lconv->{p_sign_posn}, "\n";
        print "n_sign_posn = ", $lconv->{n_sign_posn}, "\n";
```

localtime
This is identical to Perl's builtin `localtime()` function.

log
This is identical to Perl's builtin `log()` function.

log10
This is identical to the C function `log10()`.

longjmp
`longjmp()` is C−specific: use die instead.

lseek
Move the read/write file pointer. This uses file descriptors such as those obtained by calling `POSIX::open`.

```
        $fd = POSIX::open( "foo", &POSIX::O_RDONLY );
        $off_t = POSIX::lseek( $fd, 0, &POSIX::SEEK_SET );
```

Returns `undef` on failure.

malloc
`malloc()` is C−specific.

mblen
This is identical to the C function `mblen()`.

mbstowcs
This is identical to the C function `mbstowcs()`.

mbtowc
This is identical to the C function `mbtowc()`.

memchr
`memchr()` is C−specific, use `index()` instead.

memcmp
`memcmp()` is C−specific, use eq instead.

memcpy
`memcpy()` is C−specific, use = instead.

memmove
`memmove()` is C−specific, use = instead.

memset
`memset()` is C−specific, use x instead.

mkdir
This is identical to Perl's builtin `mkdir()` function.

mkfifo
This is similar to the C function `mkfifo()`.

Returns `undef` on failure.

mktime
Convert date/time info to a calendar time.

Synopsis:

```
        mktime(sec, min, hour, mday, mon, year, wday = 0, yday = 0, isdst = 0
```

The month (`mon`), weekday (`wday`), and yearday (`yday`) begin at zero. I.e. January is 0, not 1; Sunday is 0, not 1; January 1st is 0, not 1. The year (`year`) is given in years since 1900. I.e. The year 1995 is 95; the year 2001 is 101. Consult your system's `mktime()` manpage for details about these and the other arguments.

Calendar time for December 12, 1995, at 10:30 am.

```
        $time_t = POSIX::mktime( 0, 30, 10, 12, 11, 95 );
        print "Date = ", POSIX::ctime($time_t);
```

Returns `undef` on failure.

modf
: Return the integral and fractional parts of a floating−point number.

```
        ($fractional, $integral) = POSIX::modf( 3.14 );
```

nice
: This is similar to the C function `nice()`.

Returns `undef` on failure.

offsetof
: `offsetof()` is C−specific.

open
: Open a file for reading for writing. This returns file descriptors, not Perl filehandles. Use `POSIX::close` to close the file.

Open a file read−only with mode 0666.

```
        $fd = POSIX::open( "foo" );
```

Open a file for read and write.

```
        $fd = POSIX::open( "foo", &POSIX::O_RDWR );
```

Open a file for write, with truncation.

```
        $fd = POSIX::open( "foo", &POSIX::O_WRONLY | &POSIX::O_TRUNC );
```

Create a new file with mode 0640. Set up the file for writing.

```
        $fd = POSIX::open( "foo", &POSIX::O_CREAT | &POSIX::O_WRONLY, 0640 );
```

Returns `undef` on failure.

opendir
: Open a directory for reading.

```
        $dir = POSIX::opendir( "/tmp" );
        @files = POSIX::readdir( $dir );
        POSIX::closedir( $dir );
```

Returns `undef` on failure.

pathconf
: Retrieves the value of a configurable limit on a file or directory.

The following will determine the maximum length of the longest allowable pathname on the filesystem which holds `/tmp`.

```
        $path_max = POSIX::pathconf( "/tmp", &POSIX::_PC_PATH_MAX );
```

Returns `undef` on failure.

pause
: This is similar to the C function `pause()`.

Returns `undef` on failure.

perror
: This is identical to the C function `perror()`.

pipe
: Create an interprocess channel. This returns file descriptors like those returned by `POSIX::open`.

```
($fd0, $fd1) = POSIX::pipe();
POSIX::write( $fd0, "hello", 5 );
POSIX::read( $fd1, $buf, 5 );
```

pow Computes `$x` raised to the power `$exponent.`

```
$ret = POSIX::pow( $x, $exponent );
```

printf Prints the specified arguments to STDOUT.

putc `putc()` is C−specific—use print instead.

putchar `putchar()` is C−specific—use print instead.

puts `puts()` is C−specific—use print instead.

qsort `qsort()` is C−specific, use sort instead.

raise Sends the specified signal to the current process.

rand `rand()` is non−portable, use Perl‘s rand instead.

read Read from a file. This uses file descriptors such as those obtained by calling `POSIX::open`. If the buffer `$buf` is not large enough for the read then Perl will extend it to make room for the request.

```
$fd = POSIX::open( "foo", &POSIX::O_RDONLY );
$bytes = POSIX::read( $fd, $buf, 3 );
```

Returns `undef` on failure.

readdir This is identical to Perl‘s builtin `readdir()` function.

realloc `realloc()` is C−specific.

remove This is identical to Perl‘s builtin `unlink()` function.

rename This is identical to Perl‘s builtin `rename()` function.

rewind Seeks to the beginning of the file.

rewinddir This is identical to Perl‘s builtin `rewinddir()` function.

rmdir This is identical to Perl‘s builtin `rmdir()` function.

scanf `scanf()` is C−specific—use < and regular expressions instead.

setgid Sets the real group id for this process.

setjmp `setjmp()` is C−specific: use eval {} instead.

setlocale Modifies and queries program‘s locale.

The following will set the traditional UNIX system locale behavior (the second argument `"C"`).

```
$loc = POSIX::setlocale( &POSIX::LC_ALL, "C" );
```

The following will query (the missing second argument) the current LC_CTYPE category.

```
$loc = POSIX::setlocale( &POSIX::LC_CTYPE);
```

The following will set the LC_CTYPE behaviour according to the locale environment variables (the second argument `""`). Please see your systems *setlocale(3)* documentation for the locale environment variables’ meaning or consult *perllocale*.

```
$loc = POSIX::setlocale( &POSIX::LC_CTYPE, "");
```

The following will set the LC_COLLATE behaviour to Argentinian Spanish. **NOTE**: The naming and availability of locales depends on your operating system. Please consult *perllocale* for how to find out which locales are available in your system.

```
        $loc = POSIX::setlocale( &POSIX::LC_ALL, "es_AR.ISO8859−1" );
```

setpgid This is similar to the C function `setpgid()`.

Returns `undef` on failure.

setsid This is identical to the C function `setsid()`.

setuid Sets the real user id for this process.

sigaction Detailed signal management. This uses `POSIX::SigAction` objects for the `action` and `oldaction` arguments. Consult your system‘s `sigaction` manpage for details.

Synopsis:

```
        sigaction(sig, action, oldaction = 0)
```

Returns `undef` on failure.

siglongjmp

`siglongjmp()` is C−specific: use die instead.

sigpending

Examine signals that are blocked and pending. This uses `POSIX::SigSet` objects for the `sigset` argument. Consult your system‘s `sigpending` manpage for details.

Synopsis:

```
        sigpending(sigset)
```

Returns `undef` on failure.

sigprocmask

Change and/or examine calling process‘s signal mask. This uses `POSIX::SigSet` objects for the `sigset` and `oldsigset` arguments. Consult your system‘s `sigprocmask` manpage for details.

Synopsis:

```
        sigprocmask(how, sigset, oldsigset = 0)
```

Returns `undef` on failure.

sigsetjmp `sigsetjmp()` is C−specific: use eval {} instead.

sigsuspend

Install a signal mask and suspend process until signal arrives. This uses `POSIX::SigSet` objects for the `signal_mask` argument. Consult your system‘s `sigsuspend` manpage for details.

Synopsis:

```
        sigsuspend(signal_mask)
```

Returns `undef` on failure.

sin This is identical to Perl‘s builtin `sin()` function.

sinh This is identical to the C function `sinh()`.

sleep This is identical to Perl‘s builtin `sleep()` function.

sprintf This is identical to Perl‘s builtin `sprintf()` function.

sqrt
: This is identical to Perl‘s builtin `sqrt()` function.

srand
: `srand()`.

sscanf
: `sscanf()` is C−specific—use regular expressions instead.

stat
: This is identical to Perl‘s builtin `stat()` function.

strcat
: `strcat()` is C−specific, use .= instead.

strchr
: `strchr()` is C−specific, use `index()` instead.

strcmp
: `strcmp()` is C−specific, use eq instead.

strcoll
: This is identical to the C function `strcoll()`.

strcpy
: `strcpy()` is C−specific, use = instead.

strcspn
: `strcspn()` is C−specific, use regular expressions instead.

strerror
: Returns the error string for the specified errno.

strftime
: Convert date and time information to string. Returns the string.

Synopsis:

```
        strftime(fmt, sec, min, hour, mday, mon, year, wday = 0, yday = 0, is
```

The month (`mon`), weekday (`wday`), and yearday (`yday`) begin at zero. I.e. January is 0, not 1; Sunday is 0, not 1; January 1st is 0, not 1. The year (`year`) is given in years since 1900. I.e. The year 1995 is 95; the year 2001 is 101. Consult your system‘s `strftime()` manpage for details about these and the other arguments.

The string for Tuesday, December 12, 1995.

```
        $str = POSIX::strftime( "%A, %B %d, %Y", 0, 0, 0, 12, 11, 95, 2 );
        print "$str\n";
```

strlen
: `strlen()` is C−specific, use length instead.

strncat
: `strncat()` is C−specific, use .= instead.

strncmp
: `strncmp()` is C−specific, use eq instead.

strncpy
: `strncpy()` is C−specific, use = instead.

stroul
: `stroul()` is C−specific.

strpbrk
: `strpbrk()` is C−specific.

strrchr
: `strrchr()` is C−specific, use `rindex()` instead.

strspn
: `strspn()` is C−specific.

strstr
: This is identical to Perl‘s builtin `index()` function.

strtod
: String to double translation. Returns the parsed number and the number of characters in the unparsed portion of the string. Truly POSIX−compliant systems set `$!` (`$ERRNO`) to indicate a translation error, so clear `$!` before calling strtod. However, non−POSIX systems may not check for overflow, and therefore will never set `$!.`

strtod should respect any POSIX *`setlocale()`* settings.

To parse a string `$str` as a floating point number use

```
    $! = 0;
    ($num, $n_unparsed) = POSIX::strtod($str);
```

The second returned item and `$!` can be used to check for valid input:

```
    if (($str eq '') || ($n_unparsed != 0) || !$!) {
        die "Non-numeric input $str" . $! ? ": $!\n" : "\n";
    }
```

When called in a scalar context strtod returns the parsed number.

strtok `strtok()` is C−specific.

strtol String to (long) integer translation. Returns the parsed number and the number of characters in the unparsed portion of the string. Truly POSIX−compliant systems set `$!` (`$ERRNO`) to indicate a translation error, so clear `$!` before calling strtol. However, non−POSIX systems may not check for overflow, and therefore will never set `$!`.

strtol should respect any POSIX *setlocale()* settings.

To parse a string `$str` as a number in some base `$base` use

```
    $! = 0;
    ($num, $n_unparsed) = POSIX::strtol($str, $base);
```

The base should be zero or between 2 and 36, inclusive. When the base is zero or omitted strtol will use the string itself to determine the base: a leading "0x" or "0X" means hexadecimal; a leading "0" means octal; any other leading characters mean decimal. Thus, "1234" is parsed as a decimal number, "01234" as an octal number, and "0x1234" as a hexadecimal number.

The second returned item and `$!` can be used to check for valid input:

```
    if (($str eq '') || ($n_unparsed != 0) || !$!) {
        die "Non-numeric input $str" . $! ? ": $!\n" : "\n";
    }
```

When called in a scalar context strtol returns the parsed number.

strtoul String to unsigned (long) integer translation. strtoul is identical to strtol except that strtoul only parses unsigned integers. See *strtol* for details.

Note: Some vendors supply strtod and strtol but not strtoul. Other vendors that do suply strtoul parse "−1" as a valid value.

strxfrm String transformation. Returns the transformed string.

```
        $dst = POSIX::strxfrm( $src );
```

sysconf Retrieves values of system configurable variables.

The following will get the machine's clock speed.

```
        $clock_ticks = POSIX::sysconf( &POSIX::_SC_CLK_TCK );
```

Returns `undef` on failure.

system This is identical to Perl's builtin `system()` function.

tan This is identical to the C function `tan()`.

tanh This is identical to the C function `tanh()`.

tcdrain This is similar to the C function `tcdrain()`.

Returns `undef` on failure.

tcflow This is similar to the C function `tcflow()`.

Returns `undef` on failure.

tcflush
: This is similar to the C function `tcflush()`.

  Returns `undef` on failure.

tcgetpgrp
: This is identical to the C function `tcgetpgrp()`.

tcsendbreak
: This is similar to the C function `tcsendbreak()`.

  Returns `undef` on failure.

tcsetpgrp
: This is similar to the C function `tcsetpgrp()`.

  Returns `undef` on failure.

time
: This is identical to Perl‘s builtin `time()` function.

times
: The `times()` function returns elapsed realtime since some point in the past (such as system startup), user and system times for this process, and user and system times used by child processes. All times are returned in clock ticks.

```
    ($realtime, $user, $system, $cuser, $csystem) = POSIX::times();
```

  Note: Perl‘s builtin `times()` function returns four values, measured in seconds.

tmpfile
: Use method `IO::File::new_tmpfile()` instead.

tmpnam
: Returns a name for a temporary file.

```
        $tmpfile = POSIX::tmpnam();
```

tolower
: This is identical to Perl‘s builtin `lc()` function.

toupper
: This is identical to Perl‘s builtin `uc()` function.

ttyname
: This is identical to the C function `ttyname()`.

tzname
: Retrieves the time conversion information from the `tzname` variable.

```
        POSIX::tzset();
        ($std, $dst) = POSIX::tzname();
```

tzset
: This is identical to the C function `tzset()`.

umask
: This is identical to Perl‘s builtin `umask()` function.

uname
: Get name of current operating system.

```
        ($sysname, $nodename, $release, $version, $machine ) = POSIX::uname()
```

ungetc
: Use method `IO::Handle::ungetc()` instead.

unlink
: This is identical to Perl‘s builtin `unlink()` function.

utime
: This is identical to Perl‘s builtin `utime()` function.

vfprintf
: `vfprintf()` is C−specific.

vprintf
: `vprintf()` is C−specific.

vsprintf
: `vsprintf()` is C−specific.

wait
: This is identical to Perl‘s builtin `wait()` function.

waitpid
: Wait for a child process to change state. This is identical to Perl‘s builtin `waitpid()` function.

```
        $pid = POSIX::waitpid( −1, &POSIX::WNOHANG );
        print "status = ", ($? / 256), "\n";
```

wcstombs
: This is identical to the C function `wcstombs()`.

wctomb
: This is identical to the C function `wctomb()`.

write
: Write to a file. This uses file descriptors such as those obtained by calling `POSIX::open`.

```
        $fd = POSIX::open( "foo", &POSIX::O_WRONLY );
        $buf = "hello";
        $bytes = POSIX::write( $b, $buf, 5 );
```

Returns `undef` on failure.

CLASSES

POSIX::SigAction

new
: Creates a new `POSIX::SigAction` object which corresponds to the C `struct sigaction`. This object will be destroyed automatically when it is no longer needed. The first parameter is the fully−qualified name of a sub which is a signal−handler. The second parameter is a `POSIX::SigSet` object, it defaults to the empty set. The third parameter contains the `sa_flags`, it defaults to 0.

```
        $sigset = POSIX::SigSet->new(SIGINT, SIGQUIT);
        $sigaction = POSIX::SigAction->new( 'main::handler', $sigset, &POSIX:
```

This `POSIX::SigAction` object should be used with the `POSIX::sigaction()` function.

POSIX::SigSet

new
: Create a new SigSet object. This object will be destroyed automatically when it is no longer needed. Arguments may be supplied to initialize the set.

Create an empty set.

```
        $sigset = POSIX::SigSet->new;
```

Create a set with SIGUSR1.

```
        $sigset = POSIX::SigSet->new( &POSIX::SIGUSR1 );
```

addset
: Add a signal to a SigSet object.

```
        $sigset->addset( &POSIX::SIGUSR2 );
```

Returns `undef` on failure.

delset
: Remove a signal from the SigSet object.

```
        $sigset->delset( &POSIX::SIGUSR2 );
```

Returns `undef` on failure.

emptyset
: Initialize the SigSet object to be empty.

```
        $sigset->emptyset();
```

Returns `undef` on failure.

fillset
: Initialize the SigSet object to include all signals.

```
        $sigset->fillset();
```

Returns `undef` on failure.

ismember

Tests the SigSet object to see if it contains a specific signal.

```
        if( $sigset->ismember( &POSIX::SIGUSR1 ) ){
                print "contains SIGUSR1\n";
        }
```

POSIX::Termios

new
Create a new Termios object. This object will be destroyed automatically when it is no longer needed.

```
        $termios = POSIX::Termios->new;
```

getattr
Get terminal control attributes.

Obtain the attributes for stdin.

```
        $termios->getattr()
```

Obtain the attributes for stdout.

```
        $termios->getattr( 1 )
```

Returns `undef` on failure.

getcc
Retrieve a value from the c_cc field of a termios object. The c_cc field is an array so an index must be specified.

```
        $c_cc[1] = $termios->getcc(1);
```

getcflag
Retrieve the c_cflag field of a termios object.

```
        $c_cflag = $termios->getcflag;
```

getiflag
Retrieve the c_iflag field of a termios object.

```
        $c_iflag = $termios->getiflag;
```

getispeed

Retrieve the input baud rate.

```
        $ispeed = $termios->getispeed;
```

getlflag
Retrieve the c_lflag field of a termios object.

```
        $c_lflag = $termios->getlflag;
```

getoflag
Retrieve the c_oflag field of a termios object.

```
        $c_oflag = $termios->getoflag;
```

getospeed

Retrieve the output baud rate.

```
        $ospeed = $termios->getospeed;
```

setattr
Set terminal control attributes.

Set attributes immediately for stdout.

```
        $termios->setattr( 1, &POSIX::TCSANOW );
```

Returns `undef` on failure.

setcc
Set a value in the c_cc field of a termios object. The c_cc field is an array so an index must be specified.

```
$termios->setcc( &POSIX::VEOF, 1 );
```

setcflag Set the c_cflag field of a termios object.

```
$termios->setcflag( &POSIX::CLOCAL );
```

setiflag Set the c_iflag field of a termios object.

```
$termios->setiflag( &POSIX::BRKINT );
```

setispeed Set the input baud rate.

```
$termios->setispeed( &POSIX::B9600 );
```

Returns `undef` on failure.

setlflag Set the c_lflag field of a termios object.

```
$termios->setlflag( &POSIX::ECHO );
```

setoflag Set the c_oflag field of a termios object.

```
$termios->setoflag( &POSIX::OPOST );
```

setospeed
Set the output baud rate.

```
$termios->setospeed( &POSIX::B9600 );
```

Returns `undef` on failure.

Baud rate values
B38400 B75 B200 B134 B300 B1800 B150 B0 B19200 B1200 B9600 B600 B4800 B50 B2400 B110

Terminal interface values
TCSADRAIN TCSANOW TCOON TCIOFLUSH TCOFLUSH TCION TCIFLUSH TCSAFLUSH TCIOFF TCOOFF

c_cc field values
VEOF VEOL VERASE VINTR VKILL VQUIT VSUSP VSTART VSTOP VMIN VTIME NCCS

c_cflag field values
CLOCAL CREAD CSIZE CS5 CS6 CS7 CS8 CSTOPB HUPCL PARENB PARODD

c_iflag field values
BRKINT ICRNL IGNBRK IGNCR IGNPAR INLCR INPCK ISTRIP IXOFF IXON PARMRK

c_lflag field values
ECHO ECHOE ECHOK ECHONL ICANON IEXTEN ISIG NOFLSH TOSTOP

c_oflag field values
OPOST

PATHNAME CONSTANTS

Constants
_PC_CHOWN_RESTRICTED _PC_LINK_MAX _PC_MAX_CANON _PC_MAX_INPUT _PC_NAME_MAX _PC_NO_TRUNC _PC_PATH_MAX _PC_PIPE_BUF _PC_VDISABLE

POSIX CONSTANTS

Constants
_POSIX_ARG_MAX _POSIX_CHILD_MAX _POSIX_CHOWN_RESTRICTED

_POSIX_JOB_CONTROL _POSIX_LINK_MAX _POSIX_MAX_CANON _POSIX_MAX_INPUT _POSIX_NAME_MAX _POSIX_NGROUPS_MAX _POSIX_NO_TRUNC _POSIX_OPEN_MAX _POSIX_PATH_MAX _POSIX_PIPE_BUF _POSIX_SAVED_IDS _POSIX_SSIZE_MAX _POSIX_STREAM_MAX _POSIX_TZNAME_MAX _POSIX_VDISABLE _POSIX_VERSION

SYSTEM CONFIGURATION

Constants

_SC_ARG_MAX _SC_CHILD_MAX _SC_CLK_TCK _SC_JOB_CONTROL _SC_NGROUPS_MAX _SC_OPEN_MAX _SC_SAVED_IDS _SC_STREAM_MAX _SC_TZNAME_MAX _SC_VERSION

ERRNO

Constants

E2BIG EACCES EADDRINUSE EADDRNOTAVAIL EAFNOSUPPORT EAGAIN EALREADY EBADF EBUSY ECHILD ECONNABORTED ECONNREFUSED ECONNRESET EDEADLK EDESTADDRREQ EDOM EDQUOT EEXIST EFAULT EFBIG EHOSTDOWN EHOSTUNREACH EINPROGRESS EINTR EINVAL EIO EISCONN EISDIR ELOOP EMFILE EMLINK EMSGSIZE ENAMETOOLONG ENETDOWN ENETRESET ENETUNREACH ENFILE ENOBUFS ENODEV ENOENT ENOEXEC ENOLCK ENOMEM ENOPROTOOPT ENOSPC ENOSYS ENOTBLK ENOTCONN ENOTDIR ENOTEMPTY ENOTSOCK ENOTTY ENXIO EOPNOTSUPP EPERM EPFNOSUPPORT EPIPE EPROCLIM EPROTONOSUPPORT EPROTOTYPE ERANGE EREMOTE ERESTART EROFS ESHUTDOWN ESOCKTNOSUPPORT ESPIPE ESRCH ESTALE ETIMEDOUT ETOOMANYREFS ETXTBSY EUSERS EWOULDBLOCK EXDEV

FCNTL

Constants

FD_CLOEXEC F_DUPFD F_GETFD F_GETFL F_GETLK F_OK F_RDLCK F_SETFD F_SETFL F_SETLK F_SETLKW F_UNLCK F_WRLCK O_ACCMODE O_APPEND O_CREAT O_EXCL O_NOCTTY O_NONBLOCK O_RDONLY O_RDWR O_TRUNC O_WRONLY

FLOAT

Constants

DBL_DIG DBL_EPSILON DBL_MANT_DIG DBL_MAX DBL_MAX_10_EXP DBL_MAX_EXP DBL_MIN DBL_MIN_10_EXP DBL_MIN_EXP FLT_DIG FLT_EPSILON FLT_MANT_DIG FLT_MAX FLT_MAX_10_EXP FLT_MAX_EXP FLT_MIN FLT_MIN_10_EXP FLT_MIN_EXP FLT_RADIX FLT_ROUNDS LDBL_DIG LDBL_EPSILON LDBL_MANT_DIG LDBL_MAX LDBL_MAX_10_EXP LDBL_MAX_EXP LDBL_MIN LDBL_MIN_10_EXP LDBL_MIN_EXP

LIMITS

Constants

ARG_MAX CHAR_BIT CHAR_MAX CHAR_MIN CHILD_MAX INT_MAX INT_MIN LINK_MAX LONG_MAX LONG_MIN MAX_CANON MAX_INPUT MB_LEN_MAX NAME_MAX NGROUPS_MAX OPEN_MAX PATH_MAX PIPE_BUF SCHAR_MAX SCHAR_MIN SHRT_MAX SHRT_MIN SSIZE_MAX STREAM_MAX TZNAME_MAX UCHAR_MAX UINT_MAX ULONG_MAX USHRT_MAX

LOCALE

Constants

LC_ALL LC_COLLATE LC_CTYPE LC_MONETARY LC_NUMERIC LC_TIME

MATH

Constants
HUGE_VAL

SIGNAL

Constants
SA_NOCLDSTOP SA_NOCLDWAIT SA_NODEFER SA_ONSTACK SA_RESETHAND SA_RESTART SA_SIGINFO SIGABRT SIGALRM SIGCHLD SIGCONT SIGFPE SIGHUP SIGILL SIGINT SIGKILL SIGPIPE SIGQUIT SIGSEGV SIGSTOP SIGTERM SIGTSTP SIGTTIN SIGTTOU SIGUSR1 SIGUSR2 SIG_BLOCK SIG_DFL SIG_ERR SIG_IGN SIG_SETMASK SIG_UNBLOCK

STAT

Constants
S_IRGRP S_IROTH S_IRUSR S_IRWXG S_IRWXO S_IRWXU S_ISGID S_ISUID S_IWGRP S_IWOTH S_IWUSR S_IXGRP S_IXOTH S_IXUSR

Macros S_ISBLK S_ISCHR S_ISDIR S_ISFIFO S_ISREG

STDLIB

Constants
EXIT_FAILURE EXIT_SUCCESS MB_CUR_MAX RAND_MAX

STDIO

Constants
BUFSIZ EOF FILENAME_MAX L_ctermid L_cuserid L_tmpname TMP_MAX

TIME

Constants
CLK_TCK CLOCKS_PER_SEC

UNISTD

Constants
R_OK SEEK_CUR SEEK_END SEEK_SET STDIN_FILENO STDOUT_FILENO STRERR_FILENO W_OK X_OK

WAIT

Constants
WNOHANG WUNTRACED

Macros WIFEXITED WEXITSTATUS WIFSIGNALED WTERMSIG WIFSTOPPED WSTOPSIG

CREATION

This document generated by ./mkposixman.PL version 19960129.

NAME

SDBM_File − Tied access to sdbm files

SYNOPSIS

```
 use SDBM_File;

 tie(%h, 'SDBM_File', 'Op.dbmx', O_RDWR|O_CREAT, 0640);

 untie %h;
```

DESCRIPTION

See *tie*

NAME

Safe − Compile and execute code in restricted compartments

SYNOPSIS

```
use Safe;

$compartment = new Safe;

$compartment−>permit(qw(time sort :browse));

$result = $compartment−>reval($unsafe_code);
```

DESCRIPTION

The Safe extension module allows the creation of compartments in which perl code can be evaluated. Each compartment has

a new namespace

The "root" of the namespace (i.e. "main::") is changed to a different package and code evaluated in the compartment cannot refer to variables outside this namespace, even with run−time glob lookups and other tricks.

Code which is compiled outside the compartment can choose to place variables into (or *share* variables with) the compartment‘s namespace and only that data will be visible to code evaluated in the compartment.

By default, the only variables shared with compartments are the "underscore" variables `$_` and @_ (and, technically, the less frequently used %_, the _ filehandle and so on). This is because otherwise perl operators which default to `$_` will not work and neither will the assignment of arguments to @_ on subroutine entry.

an operator mask

Each compartment has an associated "operator mask". Recall that perl code is compiled into an internal format before execution. Evaluating perl code (e.g. via "eval" or "do ‘file‘") causes the code to be compiled into an internal format and then, provided there was no error in the compilation, executed. Code evaulated in a compartment compiles subject to the compartment‘s operator mask. Attempting to evaulate code in a compartment which contains a masked operator will cause the compilation to fail with an error. The code will not be executed.

The default operator mask for a newly created compartment is the ‘:default’ optag.

It is important that you read the Opcode(3) module documentation for more information, especially for detailed definitions of opnames, optags and opsets.

Since it is only at the compilation stage that the operator mask applies, controlled access to potentially unsafe operations can be achieved by having a handle to a wrapper subroutine (written outside the compartment) placed into the compartment. For example,

```
$cpt = new Safe;
sub wrapper {
    # vet arguments and perform potentially unsafe operations
}
$cpt−>share(’&wrapper’);
```

WARNING

The authors make **no warranty**, implied or otherwise, about the suitability of this software for safety or security purposes.

The authors shall not in any case be liable for special, incidental, consequential, indirect or other similar damages arising from the use of this software.

Your mileage will vary. If in any doubt **do not use it**.

RECENT CHANGES

The interface to the Safe module has changed quite dramatically since version 1 (as supplied with Perl5.002). Study these pages carefully if you have code written to use Safe version 1 because you will need to makes changes.

Methods in class Safe

To create a new compartment, use

```
    $cpt = new Safe;
```

Optional argument is (NAMESPACE), where NAMESPACE is the root namespace to use for the compartment (defaults to "Safe::Root0", incremented for each new compartment).

Note that version 1.00 of the Safe module supported a second optional parameter, MASK. That functionality has been withdrawn pending deeper consideration. Use the permit and deny methods described below.

The following methods can then be used on the compartment object returned by the above constructor. The object argument is implicit in each case.

permit (OP, ...)

Permit the listed operators to be used when compiling code in the compartment (in *addition* to any operators already permitted).

permit_only (OP, ...)

Permit *only* the listed operators to be used when compiling code in the compartment (*no* other operators are permitted).

deny (OP, ...)

Deny the listed operators from being used when compiling code in the compartment (other operators may still be permitted).

deny_only (OP, ...)

Deny *only* the listed operators from being used when compiling code in the compartment (*all* other operators will be permitted).

trap (OP, ...)
untrap (OP, ...)

The trap and untrap methods are synonyms for deny and permit respectfully.

share (NAME, ...)

This shares the variable(s) in the argument list with the compartment. This is almost identical to exporting variables using the *Exporter(3)* module.

Each NAME must be the **name** of a variable, typically with the leading type identifier included. A bareword is treated as a function name.

Examples of legal names are '`$foo`' for a scalar, '@foo' for an array, '%foo' for a hash, '`&foo`' or 'foo' for a subroutine and '*foo' for a glob (i.e. all symbol table entries associated with "foo", including scalar, array, hash, sub and filehandle).

Each NAME is assumed to be in the calling package. See share_from for an alternative method (which share uses).

share_from (PACKAGE, ARRAYREF)

This method is similar to `share()` but allows you to explicitly name the package that symbols should be shared from. The symbol names (including type characters) are supplied as an array reference.

```
    $safe−>share_from('main', [ '$foo', '%bar', 'func' ]);
```

varglob (VARNAME)

This returns a glob reference for the symbol table entry of VARNAME in the package of the compartment. VARNAME must be the **name** of a variable without any leading type marker. For example,

```
    $cpt = new Safe 'Root';
    $Root::foo = "Hello world";
    # Equivalent version which doesn't need to know $cpt's package name:
    ${$cpt->varglob('foo')} = "Hello world";
```

reval (STRING)

This evaluates STRING as perl code inside the compartment.

The code can only see the compartment‘s namespace (as returned by the **root** method). The compartment‘s root package appears to be the `main::` package to the code inside the compartment.

Any attempt by the code in STRING to use an operator which is not permitted by the compartment will cause an error (at run−time of the main program but at compile−time for the code in STRING). The error is of the form "%s trapped by operation mask operation...".

If an operation is trapped in this way, then the code in STRING will not be executed. If such a trapped operation occurs or any other compile−time or return error, then `$@` is set to the error message, just as with an `eval()`.

If there is no error, then the method returns the value of the last expression evaluated, or a return statement may be used, just as with subroutines and **`eval()`**. The context (list or scalar) is determined by the caller as usual.

This behaviour differs from the beta distribution of the Safe extension where earlier versions of perl made it hard to mimic the return behaviour of the `eval()` command and the context was always scalar.

Some points to note:

If the entereval op is permitted then the code can use eval "..." to ‘hide’ code which might use denied ops. This is not a major problem since when the code tries to execute the eval it will fail because the opmask is still in effect. However this technique would allow clever, and possibly harmful, code to ‘probe’ the boundaries of what is possible.

Any string eval which is executed by code executing in a compartment, or by code called from code executing in a compartment, will be eval‘d in the namespace of the compartment. This is potentially a serious problem.

Consider a function `foo()` in package pkg compiled outside a compartment but shared with it. Assume the compartment has a root package called ‘Root’. If `foo()` contains an eval statement like eval ‘`$foo` = 1’ then, normally, `$pkg::foo` will be set to 1. If `foo()` is called from the compartment (by whatever means) then instead of setting `$pkg::foo,` the eval will actually set `$Root::pkg::foo.`

This can easily be demonstrated by using a module, such as the Socket module, which uses eval "..." as part of an AUTOLOAD function. You can ‘use’ the module outside the compartment and share an (autoloaded) function with the compartment. If an autoload is triggered by code in the compartment, or by any code anywhere that is called by any means from the compartment, then the eval in the Socket module‘s AUTOLOAD function happens in the namespace of the compartment. Any variables created or used by the eval‘d code are now under the control of the code in the compartment.

A similar effect applies to *all* runtime symbol lookups in code called from a compartment but not compiled within it.

rdo (FILENAME)

This evaluates the contents of file FILENAME inside the compartment. See above documentation on the **reval** method for further details.

root (NAMESPACE)

This method returns the name of the package that is the root of the compartment‘s namespace.

Note that this behaviour differs from version 1.00 of the Safe module where the root module could be used to change the namespace. That functionality has been withdrawn pending deeper consideration.

mask (MASK)

This is a get−or−set method for the compartment‘s operator mask.

With no MASK argument present, it returns the current operator mask of the compartment.

With the MASK argument present, it sets the operator mask for the compartment (equivalent to calling the deny_only method).

Some Safety Issues

This section is currently just an outline of some of the things code in a compartment might do (intentionally or unintentionally) which can have an effect outside the compartment.

Memory Consuming all (or nearly all) available memory.

CPU Causing infinite loops etc.

Snooping Copying private information out of your system. Even something as simple as your user name is of value to others. Much useful information could be gleaned from your environment variables for example.

Signals Causing signals (especially SIGFPE and SIGALARM) to affect your process.

Setting up a signal handler will need to be carefully considered and controlled. What mask is in effect when a signal handler gets called? If a user can get an imported function to get an exception and call the user‘s signal handler, does that user‘s restricted mask get re−instated before the handler is called? Does an imported handler get called with its original mask or the user‘s one?

State Changes

Ops such as chdir obviously effect the process as a whole and not just the code in the compartment. Ops such as rand and srand have a similar but more subtle effect.

AUTHOR

Originally designed and implemented by Malcolm Beattie, mbeattie@sable.ox.ac.uk.

Reworked to use the Opcode module and other changes added by Tim Bunce <***Tim.Bunce@ig.co.uk***>.

NAME

Pod::Text − convert POD data to formatted ASCII text

SYNOPSIS

```
use Pod::Text;

pod2text("perlfunc.pod");
```

Also:

```
pod2text < input.pod
```

DESCRIPTION

Pod::Text is a module that can convert documentation in the POD format (such as can be found throughout the Perl distribution) into formatted ASCII. Termcap is optionally supported for boldface/underline, and can enabled via `$Pod::Text::termcap=1.` If termcap has not been enabled, then backspaces will be used to simulate bold and underlined text.

A separate ***pod2text*** program is included that is primarily a wrapper for Pod::Text.

The single function `pod2text()` can take one or two arguments. The first should be the name of a file to read the pod from, or "<`&STDIN"` to read from STDIN. A second argument, if provided, should be a filehandle glob where output should be sent.

AUTHOR

Tom Christiansen <***tchrist@mox.perl.com***>

TODO

Cleanup work. The input and output locations need to be more flexible, termcap shouldn‘t be a global variable, and the terminal speed needs to be properly calculated.

NAME

Search::Dict, look − search for key in dictionary file

SYNOPSIS

```
use Search::Dict;
look *FILEHANDLE, $key, $dict, $fold;
```

DESCRIPTION

Sets file position in FILEHANDLE to be first line greater than or equal (stringwise) to *$key*. Returns the new file position, or −1 if an error occurs.

The flags specify dictionary order and case folding:

If *$dict* is true, search by dictionary order (ignore anything but word characters and whitespace).

If *$fold* is true, ignore case.

NAME

SelectSaver − save and restore selected file handle

SYNOPSIS

```
use SelectSaver;

{
   my $saver = new SelectSaver(FILEHANDLE);
   # FILEHANDLE is selected
}
# previous handle is selected

{
   my $saver = new SelectSaver;
   # new handle may be selected, or not
}
# previous handle is selected
```

DESCRIPTION

A `SelectSaver` object contains a reference to the file handle that was selected when it was created. If its `new` method gets an extra parameter, then that parameter is selected; otherwise, the selected file handle remains unchanged.

When a `SelectSaver` is destroyed, it re−selects the file handle that was selected when it was created.

NAME

SelfLoader − load functions only on demand

SYNOPSIS

```
package FOOBAR;
use SelfLoader;

... (initializing code)

__DATA__
sub {....
```

DESCRIPTION

This module tells its users that functions in the FOOBAR package are to be autoloaded from after the `__DATA__` token. See also *Autoloading in perlsub*.

The __DATA__ token

The `__DATA__` token tells the perl compiler that the perl code for compilation is finished. Everything after the `__DATA__` token is available for reading via the filehandle FOOBAR::DATA, where FOOBAR is the name of the current package when the `__DATA__` token is reached. This works just the same as `__END__` does in package 'main', but for other modules data after `__END__` is not automatically retreivable , whereas data after `__DATA__` is. The `__DATA__` token is not recognized in versions of perl prior to 5.001m.

Note that it is possible to have `__DATA__` tokens in the same package in multiple files, and that the last `__DATA__` token in a given package that is encountered by the compiler is the one accessible by the filehandle. This also applies to `__END__` and main, i.e. if the 'main' program has an `__END__`, but a module 'require'd (_not_ 'use'd) by that program has a 'package main;' declaration followed by an '`__DATA__`', then the `DATA` filehandle is set to access the data after the `__DATA__` in the module, _not_ the data after the `__END__` token in the 'main' program, since the compiler encounters the 'require'd file later.

SelfLoader autoloading

The **SelfLoader** works by the user placing the `__DATA__` token *after* perl code which needs to be compiled and run at 'require' time, but *before* subroutine declarations that can be loaded in later − usually because they may never be called.

The **SelfLoader** will read from the FOOBAR::DATA filehandle to load in the data after `__DATA__`, and load in any subroutine when it is called. The costs are the one−time parsing of the data after `__DATA__`, and a load delay for the _first_ call of any autoloaded function. The benefits (hopefully) are a speeded up compilation phase, with no need to load functions which are never used.

The **SelfLoader** will stop reading from `__DATA__` if it encounters the `__END__` token − just as you would expect. If the `__END__` token is present, and is followed by the token DATA, then the **SelfLoader** leaves the FOOBAR::DATA filehandle open on the line after that token.

The **SelfLoader** exports the `AUTOLOAD` subroutine to the package using the **SelfLoader**, and this loads the called subroutine when it is first called.

There is no advantage to putting subroutines which will _always_ be called after the `__DATA__` token.

Autoloading and package lexicals

A 'my `$pack_lexical'` statement makes the variable `$pack_lexical` local _only_ to the file up to the `__DATA__` token. Subroutines declared elsewhere _cannot_ see these types of variables, just as if you declared subroutines in the package but in another file, they cannot see these variables.

So specifically, autoloaded functions cannot see package lexicals (this applies to both the **SelfLoader** and the Autoloader). The `vars` pragma provides an alternative to defining package−level globals that will be visible to autoloaded routines. See the documentation on **vars** in the pragma section of *perlmod*.

SelfLoader and AutoLoader

The **SelfLoader** can replace the AutoLoader − just change 'use AutoLoader' to 'use SelfLoader' (though note that the **SelfLoader** exports the AUTOLOAD function − but if you have your own AUTOLOAD and are using the AutoLoader too, you probably know what you're doing), and the `__END__` token to `__DATA__`. You will need perl version 5.001m or later to use this (version 5.001 with all patches up to patch m).

There is no need to inherit from the **SelfLoader**.

The **SelfLoader** works similarly to the AutoLoader, but picks up the subs from after the `__DATA__` instead of in the 'lib/auto' directory. There is a maintainance gain in not needing to run AutoSplit on the module at installation, and a runtime gain in not needing to keep opening and closing files to load subs. There is a runtime loss in needing to parse the code after the `__DATA__`. Details of the **AutoLoader** and another view of these distinctions can be found in that module's documentation.

__DATA__, __END__, and the FOOBAR::DATA filehandle.

This section is only relevant if you want to use the `FOOBAR::DATA` together with the **SelfLoader**.

Data after the `__DATA__` token in a module is read using the FOOBAR::DATA filehandle. `__END__` can still be used to denote the end of the `__DATA__` section if followed by the token DATA − this is supported by the **SelfLoader**. The `FOOBAR::DATA` filehandle is left open if an `__END__` followed by a DATA is found, with the filehandle positioned at the start of the line after the `__END__` token. If no `__END__` token is present, or an `__END__` token with no DATA token on the same line, then the filehandle is closed.

The **SelfLoader** reads from wherever the current position of the `FOOBAR::DATA` filehandle is, until the EOF or `__END__`. This means that if you want to use that filehandle (and ONLY if you want to), you should either

1. Put all your subroutine declarations immediately after the `__DATA__` token and put your own data after those declarations, using the `__END__` token to mark the end of subroutine declarations. You must also ensure that the **SelfLoader** reads first by calling '`SelfLoader->load_stubs()`;', or by using a function which is selfloaded;

or

2. You should read the `FOOBAR::DATA` filehandle first, leaving the handle open and positioned at the first line of subroutine declarations.

You could conceivably do both.

Classes and inherited methods.

For modules which are not classes, this section is not relevant. This section is only relevant if you have methods which could be inherited.

A subroutine stub (or forward declaration) looks like

```
  sub stub;
```

i.e. it is a subroutine declaration without the body of the subroutine. For modules which are not classes, there is no real need for stubs as far as autoloading is concerned.

For modules which ARE classes, and need to handle inherited methods, stubs are needed to ensure that the method inheritance mechanism works properly. You can load the stubs into the module at 'require' time, by adding the statement '`SelfLoader->load_stubs()`;' to the module to do this.

The alternative is to put the stubs in before the `__DATA__` token BEFORE releasing the module, and for this purpose the `Devel::SelfStubber` module is available. However this does require the extra step of ensuring that the stubs are in the module. If this is done I strongly recommend that this is done BEFORE releasing the module − it should NOT be done at install time in general.

Multiple packages and fully qualified subroutine names

Subroutines in multiple packages within the same file are supported − but you should note that this requires exporting the `SelfLoader::AUTOLOAD` to every package which requires it. This is done automatically by the **SelfLoader** when it first loads the subs into the cache, but you should really specify it in the initialization before the `__DATA__` by putting a ‘use SelfLoader’ statement in each package.

Fully qualified subroutine names are also supported. For example,

```
__DATA__
sub foo::bar {23}
package baz;
sub dob {32}
```

will all be loaded correctly by the **SelfLoader**, and the **SelfLoader** will ensure that the packages ‘foo’ and ‘baz’ correctly have the **SelfLoader** `AUTOLOAD` method when the data after `__DATA__` is first parsed.

NAME

Shell − run shell commands transparently within perl

SYNOPSIS

See below.

DESCRIPTION

```
  Date: Thu, 22 Sep 94 16:18:16 −0700
  Message−Id: <9409222318.AA17072@scalpel.netlabs.com>
  To: perl5−porters@isu.edu
  From: Larry Wall <lwall@scalpel.netlabs.com>
  Subject: a new module I just wrote
```

Here‘s one that‘ll whack your mind a little out.

```
    #!/usr/bin/perl

    use Shell;

    $foo = echo("howdy", "<funny>", "world");
    print $foo;

    $passwd = cat("</etc/passwd");
    print $passwd;

    sub ps;
    print ps −ww;

    cp("/etc/passwd", "/tmp/passwd");
```

That‘s maybe too gonzo. It actually exports an AUTOLOAD to the current package (and uncovered a bug in Beta 3, by the way). Maybe the usual usage should be

```
    use Shell qw(echo cat ps cp);
```

Larry

AUTHOR

Larry Wall

NAME

Socket, sockaddr_in, sockaddr_un, inet_aton, inet_ntoa − load the C socket.h defines and structure manipulators

SYNOPSIS

```
use Socket;

$proto = getprotobyname('udp');
socket(Socket_Handle, PF_INET, SOCK_DGRAM, $proto);
$iaddr = gethostbyname('hishost.com');
$port = getservbyname('time', 'udp');
$sin = sockaddr_in($port, $iaddr);
send(Socket_Handle, 0, 0, $sin);

$proto = getprotobyname('tcp');
socket(Socket_Handle, PF_INET, SOCK_STREAM, $proto);
$port = getservbyname('smtp');
$sin = sockaddr_in($port,inet_aton("127.1"));
$sin = sockaddr_in(7,inet_aton("localhost"));
$sin = sockaddr_in(7,INADDR_LOOPBACK);
connect(Socket_Handle,$sin);

($port, $iaddr) = sockaddr_in(getpeername(Socket_Handle));
$peer_host = gethostbyaddr($iaddr, AF_INET);
$peer_addr = inet_ntoa($iaddr);

$proto = getprotobyname('tcp');
socket(Socket_Handle, PF_UNIX, SOCK_STREAM, $proto);
unlink('/tmp/usock');
$sun = sockaddr_un('/tmp/usock');
connect(Socket_Handle,$sun);
```

DESCRIPTION

This module is just a translation of the C ***socket.h*** file. Unlike the old mechanism of requiring a translated ***socket.ph*** file, this uses the **h2xs** program (see the Perl source distribution) and your native C compiler. This means that it has a far more likely chance of getting the numbers right. This includes all of the commonly used pound−defines like AF_INET, SOCK_STREAM, etc.

In addition, some structure manipulation functions are available:

inet_aton HOSTNAME

Takes a string giving the name of a host, and translates that to the 4−byte string (structure). Takes arguments of both the 'rtfm.mit.edu' type and '18.181.0.24'. If the host name cannot be resolved, returns undef. For multi−homed hosts (hosts with more than one address), the first address found is returned.

inet_ntoa IP_ADDRESS

Takes a four byte ip address (as returned by `inet_aton()`) and translates it into a string of the form 'd.d.d.d' where the 'd's are numbers less than 256 (the normal readable four dotted number notation for internet addresses).

INADDR_ANY

Note: does not return a number, but a packed string.

Returns the 4−byte wildcard ip address which specifies any of the hosts ip addresses. (A particular machine can have more than one ip address, each address corresponding to a particular network interface. This wildcard address allows you to bind to all of them simultaneously.) Normally equivalent to inet_aton('0.0.0.0').

INADDR_BROADCAST

Note: does not return a number, but a packed string.

Returns the 4−byte 'this−lan' ip broadcast address. This can be useful for some protocols to solicit information from all servers on the same LAN cable. Normally equivalent to inet_aton('255.255.255.255').

INADDR_LOOPBACK

Note − does not return a number.

Returns the 4−byte loopback address. Normally equivalent to inet_aton('localhost').

INADDR_NONE

Note − does not return a number.

Returns the 4−byte 'invalid' ip address. Normally equivalent to inet_aton('255.255.255.255').

sockaddr_in PORT, ADDRESS
sockaddr_in SOCKADDR_IN

In an array context, unpacks its SOCKADDR_IN argument and returns an array consisting of (PORT, ADDRESS). In a scalar context, packs its (PORT, ADDRESS) arguments as a SOCKADDR_IN and returns it. If this is confusing, use `pack_sockaddr_in()` and `unpack_sockaddr_in()` explicitly.

pack_sockaddr_in PORT, IP_ADDRESS

Takes two arguments, a port number and a 4 byte IP_ADDRESS (as returned by `inet_aton()`). Returns the sockaddr_in structure with those arguments packed in with AF_INET filled in. For internet domain sockets, this structure is normally what you need for the arguments in `bind()`, `connect()`, and `send()`, and is also returned by `getpeername()`, `getsockname()` and `recv()`.

unpack_sockaddr_in SOCKADDR_IN

Takes a sockaddr_in structure (as returned by `pack_sockaddr_in()`) and returns an array of two elements: the port and the 4−byte ip−address. Will croak if the structure does not have AF_INET in the right place.

sockaddr_un PATHNAME
sockaddr_un SOCKADDR_UN

In an array context, unpacks its SOCKADDR_UN argument and returns an array consisting of (PATHNAME). In a scalar context, packs its PATHNAME arguments as a SOCKADDR_UN and returns it. If this is confusing, use `pack_sockaddr_un()` and `unpack_sockaddr_un()` explicitly. These are only supported if your system has <***sys/un.h***>.

pack_sockaddr_un PATH

Takes one argument, a pathname. Returns the sockaddr_un structure with that path packed in with AF_UNIX filled in. For unix domain sockets, this structure is normally what you need for the arguments in `bind()`, `connect()`, and `send()`, and is also returned by `getpeername()`, `getsockname()` and `recv()`.

unpack_sockaddr_un SOCKADDR_UN

Takes a sockaddr_un structure (as returned by `pack_sockaddr_un()`) and returns the pathname. Will croak if the structure does not have AF_UNIX in the right place.

NAME

Symbol − manipulate Perl symbols and their names

SYNOPSIS

```
use Symbol;

$sym = gensym;
open($sym, "filename");
$_ = <$sym>;
# etc.

ungensym $sym;      # no effect

print qualify("x"), "\n";              # "Test::x"
print qualify("x", "FOO"), "\n"        # "FOO::x"
print qualify("BAR::x"), "\n";         # "BAR::x"
print qualify("BAR::x", "FOO"), "\n";  # "BAR::x"
print qualify("STDOUT", "FOO"), "\n";  # "main::STDOUT" (global)
print qualify(\*x), "\n";              # returns \*x
print qualify(\*x, "FOO"), "\n";       # returns \*x

use strict refs;
print { qualify_to_ref $fh } "foo!\n";
$ref = qualify_to_ref $name, $pkg;
```

DESCRIPTION

`Symbol::gensym` creates an anonymous glob and returns a reference to it. Such a glob reference can be used as a file or directory handle.

For backward compatibility with older implementations that didn‘t support anonymous globs, `Symbol::ungensym` is also provided. But it doesn‘t do anything.

`Symbol::qualify` turns unqualified symbol names into qualified variable names (e.g. "myvar" −> "MyPackage::myvar"). If it is given a second parameter, `qualify` uses it as the default package; otherwise, it uses the package of its caller. Regardless, global variable names (e.g. "STDOUT", "ENV", "SIG") are always qualfied with "main::".

Qualification applies only to symbol names (strings). References are left unchanged under the assumption that they are glob references, which are qualified by their nature.

`Symbol::qualify_to_ref` is just like `Symbol::qualify` except that it returns a glob ref rather than a symbol name, so you can use the result even if `use strict ‘refs’` is in effect.

NAME

Sys::Hostname − Try every conceivable way to get hostname

SYNOPSIS

```
    use Sys::Hostname;
    $host = hostname;
```

DESCRIPTION

Attempts several methods of getting the system hostname and then caches the result. It tries `syscall(SYS_gethostname)`, `‘hostname‘`, `‘uname −n‘`, and the file ***/com/host***. If all that fails it `croak`s.

All nulls, returns, and newlines are removed from the result.

AUTHOR

David Sundstrom <***sunds@asictest.sc.ti.com***>

Texas Instruments

NAME

Sys::Syslog, openlog, closelog, setlogmask, syslog − Perl interface to the UNIX syslog(3) calls

SYNOPSIS

```
use Sys::Syslog;

openlog $ident, $logopt, $facility;
syslog $priority, $format, @args;
$oldmask = setlogmask $mask_priority;
closelog;
```

DESCRIPTION

Sys::Syslog is an interface to the UNIX `syslog(3)` program. Call `syslog()` with a string priority and a list of `printf()` args just like `syslog(3)`.

Syslog provides the functions:

openlog `$ident, $logopt, $facility`

`$ident` is prepended to every message. *`$logopt`* contains one or more of the words *pid*, *ndelay*, *cons*, *nowait*. *`$facility`* specifies the part of the system

syslog `$priority, $format, @args`

If *`$priority`* permits, logs *(`$format`, @args)* printed as by `printf(3V)`, with the addition that *%m* is replaced with `"$!"` (the latest error message).

setlogmask `$mask_priority`

Sets log mask *`$mask_priority`* and returns the old mask.

closelog

Closes the log file.

Note that `openlog` now takes three arguments, just like `openlog(3)`.

EXAMPLES

```
openlog($program, 'cons,pid', 'user');
syslog('info', 'this is another test');
syslog('mail|warning', 'this is a better test: %d', time);
closelog();

syslog('debug', 'this is the last test');
openlog("$program $$", 'ndelay', 'user');
syslog('notice', 'fooprogram: this is really done');

$! = 55;
syslog('info', 'problem was %m'); # %m == $! in syslog(3)
```

DEPENDENCIES

Sys::Syslog needs ***syslog.ph***, which can be created with `h2ph`.

SEE ALSO

syslog(3)

AUTHOR

Tom Christiansen <***tchrist@perl.com***> and Larry Wall <***larry@wall.org***>

NAME

Term::Cap − Perl termcap interface

SYNOPSIS

```
require Term::Cap;
$terminal = Tgetent Term::Cap { TERM => undef, OSPEED => $ospeed };
$terminal−>Trequire(qw/ce ku kd/);
$terminal−>Tgoto('cm', $col, $row, $FH);
$terminal−>Tputs('dl', $count, $FH);
$terminal−>Tpad($string, $count, $FH);
```

DESCRIPTION

These are low−level functions to extract and use capabilities from a terminal capability (termcap) database.

The **Tgetent** function extracts the entry of the specified terminal type *TERM* (defaults to the environment variable *TERM*) from the database.

It will look in the environment for a *TERMCAP* variable. If found, and the value does not begin with a slash, and the terminal type name is the same as the environment string *TERM*, the *TERMCAP* string is used instead of reading a termcap file. If it does begin with a slash, the string is used as a path name of the termcap file to search. If *TERMCAP* does not begin with a slash and name is different from *TERM*, **Tgetent** searches the files ***$HOME/.termcap***, ***/etc/termcap***, and ***/usr/share/misc/termcap***, in that order, unless the environment variable *TERMPATH* exists, in which case it specifies a list of file pathnames (separated by spaces or colons) to be searched **instead**. Whenever multiple files are searched and a tc field occurs in the requested entry, the entry it names must be found in the same file or one of the succeeding files. If there is a `:tc=...:` in the *TERMCAP* environment variable string it will continue the search in the files as above.

OSPEED is the terminal output bit rate (often mistakenly called the baud rate). *OSPEED* can be specified as either a POSIX termios/SYSV termio speeds (where 9600 equals 9600) or an old BSD−style speeds (where 13 equals 9600).

Tgetent returns a blessed object reference which the user can then use to send the control strings to the terminal using **Tputs** and **Tgoto**. It calls `croak` on failure.

Tgoto decodes a cursor addressing string with the given parameters.

The output strings for **Tputs** are cached for counts of 1 for performance. **Tgoto** and **Tpad** do not cache. `$self−>{_xx}` is the raw termcap data and `$self−>{xx}` is the cached version.

```
print $terminal−>Tpad($self−>{_xx}, 1);
```

Tgoto, **Tputs**, and **Tpad** return the string and will also output the string to `$FH` if specified.

The extracted termcap entry is available in the object as `$self−>{TERMCAP}.`

EXAMPLES

```
# Get terminal output speed
require POSIX;
my $termios = new POSIX::Termios;
$termios−>getattr;
my $ospeed = $termios−>getospeed;

# Old−style ioctl code to get ospeed:
#     require 'ioctl.pl';
#     ioctl(TTY,$TIOCGETP,$sgtty);
#     ($ispeed,$ospeed) = unpack('cc',$sgtty);

# allocate and initialize a terminal structure
$terminal = Tgetent Term::Cap { TERM => undef, OSPEED => $ospeed };
```

```
# require certain capabilities to be available
$terminal−>Trequire(qw/ce ku kd/);

# Output Routines, if $FH is undefined these just return the string

# Tgoto does the % expansion stuff with the given args
$terminal−>Tgoto('cm', $col, $row, $FH);

# Tputs doesn't do any % expansion.
$terminal−>Tputs('dl', $count = 1, $FH);
```

NAME

Term::Complete − Perl word completion module

SYNOPSIS

```
$input = complete('prompt_string', \@completion_list);
$input = complete('prompt_string', @completion_list);
```

DESCRIPTION

This routine provides word completion on the list of words in the array (or array ref).

The tty driver is put into raw mode using the system command `stty raw −echo` and restored using `stty −raw echo`.

The following command characters are defined:

<tab>

Attempts word completion. Cannot be changed.

^D Prints completion list. Defined by *`$Term::Complete::complete`*.

^U Erases the current input. Defined by *`$Term::Complete::kill`*.

<del>, <bs>

Erases one character. Defined by *`$Term::Complete::erase1`* and *`$Term::Complete::erase2`*.

DIAGNOSTICS

Bell sounds when word completion fails.

BUGS

The completion charater <tab> cannot be changed.

AUTHOR

Wayne Thompson

NAME

Term::ReadLine − Perl interface to various `readline` packages. If no real package is found, substitutes stubs instead of basic functions.

SYNOPSIS

```
    use Term::ReadLine;
    $term = new Term::ReadLine 'Simple Perl calc';
    $prompt = "Enter your arithmetic expression: ";
    $OUT = $term−>OUT || STDOUT;
    while ( defined ($_ = $term−>readline($prompt)) ) {
      $res = eval($_), "\n";
      warn $@ if $@;
      print $OUT $res, "\n" unless $@;
      $term−>addhistory($_) if /\S/;
    }
```

DESCRIPTION

This package is just a front end to some other packages. At the moment this description is written, the only such package is Term−ReadLine, available on CPAN near you. The real target of this stub package is to set up a common interface to whatever Readline emerges with time.

Minimal set of supported functions

All the supported functions should be called as methods, i.e., either as

```
  $term = new Term::ReadLine 'name';
```

or as

```
  $term−>addhistory('row');
```

where `$term` is a return value of Term::ReadLine−>Init.

`ReadLine` returns the actual package that executes the commands. Among possible values are `Term::ReadLine::Gnu`, `Term::ReadLine::Perl`, `Term::ReadLine::Stub Exporter`.

`new` returns the handle for subsequent calls to following functions. Argument is the name of the application. Optionally can be followed by two arguments for `IN` and `OUT` filehandles. These arguments should be globs.

`readline` gets an input line, *possibly* with actual `readline` support. Trailing newline is removed. Returns `undef` on `EOF`.

`addhistory` adds the line to the history of input, from where it can be used if the actual `readline` is present.

`IN`, `$OUT` return the filehandles for input and output or `undef` if `readline` input and output cannot be used for Perl.

`MinLine` If argument is specified, it is an advice on minimal size of line to be included into history. `undef` means do not include anything into history. Returns the old value.

`findConsole` returns an array with two strings that give most appropriate names for files for input and output using conventions `"<$in", ">out"`.

Attribs returns a reference to a hash which describes internal configuration of the package. Names of keys in this hash conform to standard conventions with the leading `rl_` stripped.

`Features` Returns a reference to a hash with keys being features present in current implementation. Several optional features are used in the minimal interface: `appname` should be present if the first argument to `new` is recognized, and `minline` should be present if `MinLine` method is not dummy. `autohistory` should be present if lines are put into history automatically (maybe subject to `MinLine`), and `addhistory` if `addhistory` method is not dummy.

If `Features` method reports a feature `attribs` as present, the method `Attribs` is not dummy.

Additional supported functions

Actually `Term::ReadLine` can use some other package, that will support reacher set of commands.

All these commands are callable via method interface and have names which conform to standard conventions with the leading `rl_` stripped.

EXPORTS

None

ENVIRONMENT

The variable `PERL_RL` governs which ReadLine clone is loaded. If the value is false, a dummy interface is used. If the value is true, it should be tail of the name of the package to use, such as `Perl` or `Gnu`.

If the variable is not set, the best available package is loaded.

NAME

Test::Harness − run perl standard test scripts with statistics

SYNOPSIS

use Test::Harness;

runtests(@tests);

DESCRIPTION

Perl test scripts print to standard output `"ok N"` for each single test, where `N` is an increasing sequence of integers. The first line output by a standard test script is `"1..M"` with `M` being the number of tests that should be run within the test script. Test::Harness::runtests(@tests) runs all the testscripts named as arguments and checks standard output for the expected `"ok N"` strings.

After all tests have been performed, `runtests()` prints some performance statistics that are computed by the Benchmark module.

The test script output

Any output from the testscript to standard error is ignored and bypassed, thus will be seen by the user. Lines written to standard output containing `/^(not\s+)?ok\b/` are interpreted as feedback for `runtests()`. All other lines are discarded.

It is tolerated if the test numbers after `ok` are omitted. In this case Test::Harness maintains temporarily its own counter until the script supplies test numbers again. So the following test script

```
print <<END;
1..6
not ok
ok
not ok
ok
ok
END
```

will generate

```
FAILED tests 1, 3, 6
Failed 3/6 tests, 50.00% okay
```

The global variable `$Test::Harness::verbose` is exportable and can be used to let `runtests()` display the standard output of the script without altering the behavior otherwise.

EXPORT

`&runtests` is exported by Test::Harness per default.

DIAGNOSTICS

`All tests successful.\nFiles=%d,  Tests=%d, %s`

If all tests are successful some statistics about the performance are printed.

`FAILED tests %s\n\tFailed %d/%d tests, %.2f%% okay.`

For any single script that has failing subtests statistics like the above are printed.

`Test returned status %d (wstat %d)`

Scripts that return a non−zero exit status, both `$? >> 8` and `$?` are printed in a message similar to the above.

`Failed 1 test, %.2f%% okay. %s`
`Failed %d/%d tests, %.2f%% okay. %s`

If not all tests were successful, the script dies with one of the above messages.

SEE ALSO

See *Benchmark* for the underlying timing routines.

AUTHORS

Either Tim Bunce or Andreas Koenig, we don‘t know. What we know for sure is, that it was inspired by Larry Wall‘s TEST script that came with perl distributions for ages. Numerous anonymous contributors exist. Current maintainer is Andreas Koenig.

BUGS

Test::Harness uses `$^X` to determine the perl binary to run the tests with. Test scripts running via the shebang (`#!`) line may not be portable because `$^X` is not consistent for shebang scripts across platforms. This is no problem when Test::Harness is run with an absolute path to the perl binary or when `$^X` can be found in the path.

NAME

abbrev − create an abbreviation table from a list

SYNOPSIS

```
use Text::Abbrev;
abbrev $hashref, LIST
```

DESCRIPTION

Stores all unambiguous truncations of each element of LIST as keys key in the associative array referenced to by `$hashref.` The values are the original list elements.

EXAMPLE

```
$hashref = abbrev qw(list edit send abort gripe);

%hash = abbrev qw(list edit send abort gripe);

abbrev $hashref, qw(list edit send abort gripe);

abbrev(*hash, qw(list edit send abort gripe));
```

NAME

Text::ParseWords − parse text into an array of tokens

SYNOPSIS

```
    use Text::ParseWords;
    @words = &quotewords($delim, $keep, @lines);
    @words = &shellwords(@lines);
    @words = &old_shellwords(@lines);
```

DESCRIPTION

`&quotewords()` accepts a delimiter (which can be a regular expression) and a list of lines and then breaks those lines up into a list of words ignoring delimiters that appear inside quotes.

The `$keep` argument is a boolean flag. If true, the quotes are kept with each word, otherwise quotes are stripped in the splitting process. `$keep` also defines whether unprotected backslashes are retained.

A `&shellwords()` replacement is included to demonstrate the new package. This version differs from the original in that it will _NOT_ default to using `$_` if no arguments are given. I personally find the old behavior to be a mis−feature.

`&quotewords()` works by simply jamming all of @lines into a single string in `$_` and then pulling off words a bit at a time until `$_` is exhausted.

AUTHORS

Hal Pomeranz (pomeranz@netcom.com), 23 March 1994

Basically an update and generalization of the old shellwords.pl. Much code shamelessly stolen from the old version (author unknown).

NAME

Text::Soundex − Implementation of the Soundex Algorithm as Described by Knuth

SYNOPSIS

```
use Text::Soundex;

$code = soundex $string;            # get soundex code for a string
@codes = soundex @list;             # get list of codes for list of strings

# set value to be returned for strings without soundex code

$soundex_nocode = ’Z000’;
```

DESCRIPTION

This module implements the soundex algorithm as described by Donald Knuth in Volume 3 of **The Art of Computer Programming**. The algorithm is intended to hash words (in particular surnames) into a small space using a simple model which approximates the sound of the word when spoken by an English speaker. Each word is reduced to a four character string, the first character being an upper case letter and the remaining three being digits.

If there is no soundex code representation for a string then the value of `$soundex_nocode` is returned. This is initially set to `undef`, but many people seem to prefer an *unlikely* value like `Z000` (how unlikely this is depends on the data set being dealt with.) Any value can be assigned to `$soundex_nocode.`

In scalar context `soundex` returns the soundex code of its first argument, and in array context a list is returned in which each element is the soundex code for the corresponding argument passed to `soundex` e.g.

```
@codes = soundex qw(Mike Stok);
```

leaves `@codes` containing `(‘M200‘, ‘S320’)`.

EXAMPLES

Knuth‘s examples of various names and the soundex codes they map to are listed below:

```
Euler, Ellery −> E460
Gauss, Ghosh −> G200
Hilbert, Heilbronn −> H416
Knuth, Kant −> K530
Lloyd, Ladd −> L300
Lukasiewicz, Lissajous −> L222
```

so:

```
$code = soundex ’Knuth’;              # $code contains ’K530’
@list = soundex qw(Lloyd Gauss);      # @list contains ’L300’, ’G200’
```

LIMITATIONS

As the soundex algorithm was originally used a **long** time ago in the US it considers only the English alphabet and pronunciation.

As it is mapping a large space (arbitrary length strings) onto a small space (single letter plus 3 digits) no inference can be made about the similarity of two strings which end up with the same soundex code. For example, both `Hilbert` and `Heilbronn` end up with a soundex code of `H416`.

AUTHOR

This code was implemented by Mike Stok (`stok@cybercom.net`) from the description given by Knuth. Ian Phillips (`ian@pipex.net`) and Rich Pinder (`rpinder@hsc.usc.edu`) supplied ideas and spotted mistakes.

NAME

Text::Tabs — expand and unexpand tabs per the unix expand(1) and unexpand(1)

SYNOPSIS

use Text::Tabs;

`$tabstop` = 4; @lines_without_tabs = expand(@lines_with_tabs); @lines_with_tabs = unexpand(@lines_without_tabs);

DESCRIPTION

Text::Tabs does about what the unix utilities expand(1) and unexpand(1) do. Given a line with tabs in it, expand will replace the tabs with the appropriate number of spaces. Given a line with or without tabs in it, unexpand will add tabs when it can save bytes by doing so. Invisible compression with plain ascii!

BUGS

expand doesn‘t handle newlines very quickly — do not feed it an entire document in one string. Instead feed it an array of lines.

AUTHOR

David Muir Sharnoff <muir@idiom.com

NAME

Text::Wrap − line wrapping to form simple paragraphs

SYNOPSIS

```
use Text::Wrap

print wrap($initial_tab, $subsequent_tab, @text);

use Text::Wrap qw(wrap $columns);

$columns = 132;
```

DESCRIPTION

`Text::Wrap::wrap()` is a very simple paragraph formatter. It formats a single paragraph at a time by breaking lines at word boundries. Indentation is controlled for the first line (`$initial_tab`) and all subsquent lines (`$subsequent_tab`) independently. `$Text::Wrap::columns` should be set to the full width of your output device.

EXAMPLE

```
print wrap("\t","","This is a bit of text that forms
        a normal book−style paragraph");
```

BUGS

It‘s not clear what the correct behavior should be when `Wrap()` is presented with a word that is longer than a line. The previous behavior was to die. Now the word is split at line−length.

AUTHOR

David Muir Sharnoff <muir@idiom.com with help from Tim Pierce and others.

NAME

Tie::Hash, Tie::StdHash − base class definitions for tied hashes

SYNOPSIS

```
    package NewHash;
    require Tie::Hash;

    @ISA = (Tie::Hash);

    sub DELETE { ... }          # Provides needed method
    sub CLEAR { ... }           # Overrides inherited method

    package NewStdHash;
    require Tie::Hash;

    @ISA = (Tie::StdHash);

    # All methods provided by default, define only those needing overrides
    sub DELETE { ... }

    package main;

    tie %new_hash, 'NewHash';
    tie %new_std_hash, 'NewStdHash';
```

DESCRIPTION

This module provides some skeletal methods for hash−tying classes. See *perltie* for a list of the functions required in order to tie a hash to a package. The basic **Tie::Hash** package provides a `new` method, as well as methods `TIEHASH`, `EXISTS` and `CLEAR`. The **Tie::StdHash** package provides most methods required for hashes in *perltie*. It inherits from **Tie::Hash**, and causes tied hashes to behave exactly like standard hashes, allowing for selective overloading of methods. The `new` method is provided as grandfathering in the case a class forgets to include a `TIEHASH` method.

For developers wishing to write their own tied hashes, the required methods are briefly defined below. See the *perltie* section for more detailed descriptive, as well as example code:

TIEHASH classname, LIST

The method invoked by the command `tie %hash, classname`. Associates a new hash instance with the specified class. `LIST` would represent additional arguments (along the lines of *AnyDBM_File* and compatriots) needed to complete the association.

STORE this, key, value

Store datum *value* into *key* for the tied hash *this*.

FETCH this, key

Retrieve the datum in *key* for the tied hash *this*.

FIRSTKEY this

Return the (key, value) pair for the first key in the hash.

NEXTKEY this, lastkey

Return the next (key, value) pair for the hash.

EXISTS this, key

Verify that *key* exists with the tied hash *this*.

DELETE this, key

Delete the key *key* from the tied hash *this*.

CLEAR this

Clear all values from the tied hash *this*.

CAVEATS

The *perltie* documentation includes a method called `DESTROY` as a necessary method for tied hashes. Neither **Tie::Hash** nor **Tie::StdHash** define a default for this method. This is a standard for class packages, but may be omitted in favor of a simple default.

MORE INFORMATION

The packages relating to various DBM−related implemetations (***DB_File***, ***NDBM_File***, etc.) show examples of general tied hashes, as does the *Config* module. While these do not utilize **Tie::Hash**, they serve as good working examples.

NAME

Tie::RefHash − use references as hash keys

SYNOPSIS

```
    require 5.004;
    use Tie::RefHash;
    tie HASHVARIABLE, ’Tie::RefHash’, LIST;

    untie HASHVARIABLE;
```

DESCRIPTION

This module provides the ability to use references as hash keys if you first `tie` the hash variable to this module.

It is implemented using the standard perl TIEHASH interface. Please see the `tie` entry in perlfunc(1) and perltie(1) for more information.

EXAMPLE

```
    use Tie::RefHash;
    tie %h, ’Tie::RefHash’;
    $a = [];
    $b = {};
    $c = \*main;
    $d = \"gunk";
    $e = sub { ’foo’ };
    %h = ($a => 1, $b => 2, $c => 3, $d => 4, $e => 5);
    $a−>[0] = ’foo’;
    $b−>{foo} = ’bar’;
    for (keys %h) {
       print ref($_), "\n";
    }
```

AUTHOR

Gurusamy Sarathy gsar@umich.edu

VERSION

Version 1.2 15 Dec 1996

SEE ALSO

perl(1), perlfunc(1), perltie(1)

NAME

Tie::Scalar, Tie::StdScalar − base class definitions for tied scalars

SYNOPSIS

```
package NewScalar;
require Tie::Scalar;

@ISA = (Tie::Scalar);

sub FETCH { ... }           # Provide a needed method
sub TIESCALAR { ... }       # Overrides inherited method

package NewStdScalar;
require Tie::Scalar;

@ISA = (Tie::StdScalar);

# All methods provided by default, so define only what needs be overridden
sub FETCH { ... }

package main;

tie $new_scalar, 'NewScalar';
tie $new_std_scalar, 'NewStdScalar';
```

DESCRIPTION

This module provides some skeletal methods for scalar−tying classes. See *perltie* for a list of the functions required in tying a scalar to a package. The basic **Tie::Scalar** package provides a `new` method, as well as methods `TIESCALAR`, `FETCH` and `STORE`. The **Tie::StdScalar** package provides all the methods specified in *perltie*. It inherits from **Tie::Scalar** and causes scalars tied to it to behave exactly like the built−in scalars, allowing for selective overloading of methods. The `new` method is provided as a means of grandfathering, for classes that forget to provide their own `TIESCALAR` method.

For developers wishing to write their own tied−scalar classes, the methods are summarized below. The *perltie* section not only documents these, but has sample code as well:

TIESCALAR classname, LIST

The method invoked by the command `tie $scalar, classname`. Associates a new scalar instance with the specified class. `LIST` would represent additional arguments (along the lines of *AnyDBM_File* and compatriots) needed to complete the association.

FETCH this

Retrieve the value of the tied scalar referenced by *this*.

STORE this, value

Store data *value* in the tied scalar referenced by *this*.

DESTROY this

Free the storage associated with the tied scalar referenced by *this*. This is rarely needed, as Perl manages its memory quite well. But the option exists, should a class wish to perform specific actions upon the destruction of an instance.

MORE INFORMATION

The *perltie* section uses a good example of tying scalars by associating process IDs with priority.

NAME

Tie::SubstrHash − Fixed−table−size, fixed−key−length hashing

SYNOPSIS

```
    require Tie::SubstrHash;

    tie %myhash, 'Tie::SubstrHash', $key_len, $value_len, $table_size;
```

DESCRIPTION

The **Tie::SubstrHash** package provides a hash−table−like interface to an array of determinate size, with constant key size and record size.

Upon tying a new hash to this package, the developer must specify the size of the keys that will be used, the size of the value fields that the keys will index, and the size of the overall table (in terms of key−value pairs, not size in hard memory). *These values will not change for the duration of the tied hash*. The newly−allocated hash table may now have data stored and retrieved. Efforts to store more than `$table_size` elements will result in a fatal error, as will efforts to store a value not exactly `$value_len` characters in length, or reference through a key not exactly `$key_len` characters in length. While these constraints may seem excessive, the result is a hash table using much less internal memory than an equivalent freely−allocated hash table.

CAVEATS

Because the current implementation uses the table and key sizes for the hashing algorithm, there is no means by which to dynamically change the value of any of the initialization parameters.

NAME

Time::Local − efficiently compute time from local and GMT time

SYNOPSIS

```
    $time = timelocal($sec,$min,$hours,$mday,$mon,$year);
    $time = timegm($sec,$min,$hours,$mday,$mon,$year);
```

DESCRIPTION

These routines are quite efficient and yet are always guaranteed to agree with `localtime()` and `gmtime()`. We manage this by caching the start times of any months we‘ve seen before. If we know the start time of the month, we can always calculate any time within the month. The start times themselves are guessed by successive approximation starting at the current time, since most dates seen in practice are close to the current date. Unlike algorithms that do a binary search (calling gmtime once for each bit of the time value, resulting in 32 calls), this algorithm calls it at most 6 times, and usually only once or twice. If you hit the month cache, of course, it doesn‘t call it at all.

timelocal is implemented using the same cache. We just assume that we‘re translating a GMT time, and then fudge it when we‘re done for the timezone and daylight savings arguments. The timezone is determined by examining the result of localtime(0) when the package is initialized. The daylight savings offset is currently assumed to be one hour.

Both routines return −1 if the integer limit is hit. I.e. for dates after the 1st of January, 2038 on most machines.

NAME

Time::gmtime − by−name interface to Perl‘s built−in `gmtime()` function

SYNOPSIS

```
use Time::gmtime;
$gm = gmtime();
printf "The day in Greenwich is %s\n",
   (qw(Sun Mon Tue Wed Thu Fri Sat Sun))[ gm−>wday() ];

use Time::gmtime w(:FIELDS;
printf "The day in Greenwich is %s\n",
   (qw(Sun Mon Tue Wed Thu Fri Sat Sun))[ gm_wday() ];

$now = gmctime();

use Time::gmtime;
use File::stat;
$date_string = gmctime(stat($file)−>mtime);
```

DESCRIPTION

This module‘s default exports override the core `gmtime()` function, replacing it with a version that returns "Time::tm" objects. This object has methods that return the similarly named structure field name from the C‘s tm structure from ***time.h***; namely sec, min, hour, mday, mon, year, wday, yday, and isdst.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `tm_` in front their method names. Thus, `$tm_obj−>mday()` corresponds to `$tm_mday` if you import the fields.

The `gmctime()` funtion provides a way of getting at the scalar sense of the original `CORE::gmtime()` function.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built−ins are still available via the `CORE::` pseudo−package.

NOTE

While this class is currently implemented using the Class::Template module to build a struct−like class, you shouldn‘t rely upon this.

AUTHOR

Tom Christiansen

NAME

Time::localtime − by−name interface to Perl‘s built−in `localtime()` function

SYNOPSIS

```
use Time::localtime;
printf "Year is %d\n", localtime->year() + 1900;

$now = ctime();

use Time::localtime;
use File::stat;
$date_string = ctime(stat($file)->mtime);
```

DESCRIPTION

This module‘s default exports override the core `localtime()` function, replacing it with a version that returns "Time::tm" objects. This object has methods that return the similarly named structure field name from the C‘s tm structure from ***time.h***; namely sec, min, hour, mday, mon, year, wday, yday, and isdst.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `tm_` in front their method names. Thus, `$tm_obj->mday()` corresponds to `$tm_mday` if you import the fields.

The `ctime()` funtion provides a way of getting at the scalar sense of the original `CORE::localtime()` function.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built−ins are still available via the `CORE::` pseudo−package.

NOTE

While this class is currently implemented using the Class::Template module to build a struct−like class, you shouldn‘t rely upon this.

AUTHOR

Tom Christiansen

NAME

Time::tm − internal object used by Time::gmtime and Time::localtime

SYNOPSIS

Don‘t use this module directly.

DESCRIPTION

This module is used internally as a base class by Time::localtime And Time::gmtime functions. It creates a Time::tm struct object which is addressable just like‘s C‘s tm structure from ***time.h***; namely with sec, min, hour, mday, mon, year, wday, yday, and isdst.

This class is an internal interface only.

AUTHOR

Tom Christiansen

NAME

User::grent − by−name interface to Perl‘s built−in `getgr*()` functions

SYNOPSIS

```
use User::grent;
$gr = getgrgid(0) or die "No group zero";
if ( $gr−>name eq ’wheel’ && @{$gr−>members} > 1 ) {
    print "gid zero name wheel, with other members";
}

use User::grent qw(:FIELDS;
getgrgid(0) or die "No group zero";
if ( $gr_name eq ’wheel’ && @gr_members > 1 ) {
    print "gid zero name wheel, with other members";
}

$gr = getgr($whoever);
```

DESCRIPTION

This module‘s default exports override the core `getgrent()`, `getgruid()`, and `getgrnam()` functions, replacing them with versions that return "User::grent" objects. This object has methods that return the similarly named structure field name from the C‘s passwd structure from ***grp.h***; namely name, passwd, gid, and members (not mem). The first three return scalars, the last an array reference.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `gr_`. Thus, `$group_obj−>gid()` corresponds to `$gr_gid` if you import the fields. Array references are available as regular array variables, so `@{ $group_obj−>members() }` would be simply @gr_members.

The `getpw()` funtion is a simple front−end that forwards a numeric argument to `getpwuid()` and the rest to `getpwnam()`.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built−ins are still available via the `CORE::` pseudo−package.

NOTE

While this class is currently implemented using the Class::Template module to build a struct−like class, you shouldn‘t rely upon this.

AUTHOR

Tom Christiansen

NAME

User::pwent − by−name interface to Perl‘s built−in `getpw*()` functions

SYNOPSIS

```
use User::pwent;
$pw = getpwnam('daemon') or die "No daemon user";
if ( $pw->uid == 1 && $pw->dir =~ m#^/(bin|tmp)?$# ) {
    print "gid 1 on root dir";
}

use User::pwent qw(:FIELDS);
getpwnam('daemon') or die "No daemon user";
if ( $pw_uid == 1 && $pw_dir =~ m#^/(bin|tmp)?$# ) {
    print "gid 1 on root dir";
}

$pw = getpw($whoever);
```

DESCRIPTION

This module‘s default exports override the core `getpwent()`, `getpwuid()`, and `getpwnam()` functions, replacing them with versions that return "User::pwent" objects. This object has methods that return the similarly named structure field name from the C‘s passwd structure from ***pwd.h***; namely name, passwd, uid, gid, quota, comment, gecos, dir, and shell.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `pw_` in front their method names. Thus, `$passwd_obj->shell()` corresponds to `$pw_shell` if you import the fields.

The `getpw()` funtion is a simple front−end that forwards a numeric argument to `getpwuid()` and the rest to `getpwnam()`.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built−ins are still available via the `CORE::` pseudo−package.

NOTE

While this class is currently implemented using the Class::Template module to build a struct−like class, you shouldn‘t rely upon this.

AUTHOR

Tom Christiansen

NAME

autouse − postpone load of modules until a function is used

SYNOPSIS

```
use autouse 'Carp' => qw(carp croak);
carp "this carp was predeclared and autoused ";
```

DESCRIPTION

If the module `Module` is already loaded, then the declaration

```
use autouse 'Module' => qw(func1 func2($;$) Module::func3);
```

is equivalent to

```
use Module qw(func1 func2);
```

if `Module` defines `func2()` with prototype `($;$)`, and `func1()` and `func3()` have no prototypes. (At least if `Module` uses `Exporter`'s `import`, otherwise it is a fatal error.)

If the module `Module` is not loaded yet, then the above declaration declares functions `func1()` and `func2()` in the current package, and declares a function `Module::func3()`. When these functions are called, they load the package `Module` if needed, and substitute themselves with the correct definitions.

WARNING

Using `autouse` will move important steps of your program's execution from compile time to runtime. This can

- Break the execution of your program if the module you `autouse`d has some initialization which it expects to be done early.
- hide bugs in your code since important checks (like correctness of prototypes) is moved from compile time to runtime. In particular, if the prototype you specified on `autouse` line is wrong, you will not find it out until the corresponding function is executed. This will be very unfortunate for functions which are not always called (note that for such functions `autouse`ing gives biggest win, for a workaround see below).

To alleviate the second problem (partially) it is advised to write your scripts like this:

```
use Module;
use autouse Module => qw(carp($) croak(&$));
carp "this carp was predeclared and autoused ";
```

The first line ensures that the errors in your argument specification are found early. When you ship your application you should comment out the first line, since it makes the second one useless.

BUGS

If `Module::func3()` is autoused, and the module is loaded between the `autouse` directive and a call to `Module::func3()`, warnings about redefinition would appear if warnings are enabled.

If `Module::func3()` is autoused, warnings are disabled when loading the module via autoused functions.

AUTHOR

Ilya Zakharevich (ilya@math.ohio−state.edu)

SEE ALSO

perl(1).

NAME

blib − Use MakeMaker‘s uninstalled version of a package

SYNOPSIS

```
 perl −Mblib script [args...]

 perl −Mblib=dir script [args...]
```

DESCRIPTION

Looks for MakeMaker−like *‘blib’* directory structure starting in *dir* (or current directory) and working back up to five levels of ’..’.

Intended for use on command line with **−M** option as a way of testing arbitary scripts against an uninstalled version of a package.

However it is possible to :

```
 use blib;
 or
 use blib ’..’;
```

etc. if you really must.

BUGS

Pollutes global name space for development only task.

AUTHOR

Nick Ing−Simmons nik@tiuk.ti.com

NAME

diagnostics − Perl compiler pragma to force verbose warning diagnostics

splain − standalone program to do the same thing

SYNOPSIS

As a pragma:

```
use diagnostics;
use diagnostics −verbose;

enable  diagnostics;
disable diagnostics;
```

Aa a program:

```
perl program 2>diag.out
splain [−v] [−p] diag.out
```

DESCRIPTION

The `diagnostics` Pragma

This module extends the terse diagnostics normally emitted by both the perl compiler and the perl interpeter, augmenting them with the more explicative and endearing descriptions found in *perldiag*. Like the other pragmata, it affects the compilation phase of your program rather than merely the execution phase.

To use in your program as a pragma, merely invoke

```
use diagnostics;
```

at the start (or near the start) of your program. (Note that this *does* enable perl‘s **−w** flag.) Your whole compilation will then be subject(ed :−) to the enhanced diagnostics. These still go out **STDERR**.

Due to the interaction between runtime and compiletime issues, and because it‘s probably not a very good idea anyway, you may not use `no diagnostics` to turn them off at compiletime. However, you may control there behaviour at runtime using the `disable()` and `enable()` methods to turn them off and on respectively.

The **−verbose** flag first prints out the *perldiag* introduction before any other diagnostics. The `$diagnostics::PRETTY` variable can generate nicer escape sequences for pagers.

The *splain* Program

While apparently a whole nuther program, *splain* is actually nothing more than a link to the (executable) ***diagnostics.pm*** module, as well as a link to the ***diagnostics.pod*** documentation. The **−v** flag is like the `use diagnostics −verbose` directive. The **−p** flag is like the `$diagnostics::PRETTY` variable. Since you‘re post−processing with *splain*, there‘s no sense in being able to `enable()` or `disable()` processing.

Output from *splain* is directed to **STDOUT**, unlike the pragma.

EXAMPLES

The following file is certain to trigger a few errors at both runtime and compiletime:

```
use diagnostics;
print NOWHERE "nothing\n";
print STDERR "\n\tThis message should be unadorned.\n";
warn "\tThis is a user warning";
print "\nDIAGNOSTIC TESTER: Please enter a <CR> here: ";
my $a, $b = scalar <STDIN>;
print "\n";
print $x/$y;
```

If you prefer to run your program first and look at its problem afterwards, do this:

```
perl −w test.pl 2>test.out
./splain < test.out
```

Note that this is not in general possible in shells of more dubious heritage, as the theoretical

```
(perl −w test.pl >/dev/tty) >& test.out
./splain < test.out
```

Because you just moved the existing **stdout** to somewhere else.

If you don‘t want to modify your source code, but still have on−the−fly warnings, do this:

```
exec 3>&1; perl −w test.pl 2>&1 1>&3 3>&− | splain 1>&2 3>&−
```

Nifty, eh?

If you want to control warnings on the fly, do something like this. Make sure you do the `use` first, or you won‘t be able to get at the `enable()` or `disable()` methods.

```
use diagnostics; # checks entire compilation phase
    print "\ntime for 1st bogus diags: SQUAWKINGS\n";
    print BOGUS1 ’nada’;
    print "done with 1st bogus\n";

disable diagnostics; # only turns off runtime warnings
    print "\ntime for 2nd bogus: (squelched)\n";
    print BOGUS2 ’nada’;
    print "done with 2nd bogus\n";

enable diagnostics; # turns back on runtime warnings
    print "\ntime for 3rd bogus: SQUAWKINGS\n";
    print BOGUS3 ’nada’;
    print "done with 3rd bogus\n";

disable diagnostics;
    print "\ntime for 4th bogus: (squelched)\n";
    print BOGUS4 ’nada’;
    print "done with 4th bogus\n";
```

INTERNALS

Diagnostic messages derive from the ***perldiag.pod*** file when available at runtime. Otherwise, they may be embedded in the file itself when the splain package is built. See the ***Makefile*** for details.

If an extant `$SIG{__WARN__}` handler is discovered, it will continue to be honored, but only after the `diagnostics::splainthis()` function (the module‘s `$SIG{__WARN__}` interceptor) has had its way with your warnings.

There is a `$diagnostics::DEBUG` variable you may set if you‘re desperately curious what sorts of things are being intercepted.

```
BEGIN { $diagnostics::DEBUG = 1 }
```

BUGS

Not being able to say "no diagnostics" is annoying, but may not be insurmountable.

The `−pretty` directive is called too late to affect matters. You have to to this instead, and *before* you load the module.

```
BEGIN { $diagnostics::PRETTY = 1 }
```

I could start up faster by delaying compilation until it should be needed, but this gets a "panic: top_level" when using the pragma form in Perl 5.001e.

While it‘s true that this documentation is somewhat subserious, if you use a program named *splain*, you should expect a bit of whimsy.

AUTHOR

Tom Christiansen <***tchrist@mox.perl.com***, 25 June 1995.

NAME

integer − Perl pragma to compute arithmetic in integer instead of double

SYNOPSIS

```
use integer;
$x = 10/3;
# $x is now 3, not 3.33333333333333333
```

DESCRIPTION

This tells the compiler that it‘s okay to use integer operations from here to the end of the enclosing BLOCK. On many machines, this doesn‘t matter a great deal for most computations, but on those without floating point hardware, it can make a big difference.

See *Pragmatic Modules*.

NAME

less − perl pragma to request less of something from the compiler

SYNOPSIS

```
    use less;  # unimplemented
```

DESCRIPTION

Currently unimplemented, this may someday be a compiler directive to make certain trade−offs, such as perhaps

```
    use less ’memory’;
    use less ’CPU’;
    use less ’fat’;
```

NAME

lib − manipulate @INC at compile time

SYNOPSIS

```
    use lib LIST;

    no lib LIST;
```

DESCRIPTION

This is a small simple module which simplifies the manipulation of @INC at compile time.

It is typically used to add extra directories to perl‘s search path so that later `use` or `require` statements will find modules which are not located on perl‘s default search path.

ADDING DIRECTORIES TO @INC

The parameters to `use lib` are added to the start of the perl search path. Saying

```
    use lib LIST;
```

is *almost* the same as saying

```
    BEGIN { unshift(@INC, LIST) }
```

For each directory in LIST (called `$dir` here) the lib module also checks to see if a directory called `$dir/$archname/auto` exists. If so the `$dir/$archname` directory is assumed to be a corresponding architecture specific directory and is added to @INC in front of `$dir.`

If LIST includes both `$dir` and `$dir/$archname` then `$dir/$archname` will be added to @INC twice (if `$dir/$archname/auto` exists).

DELETING DIRECTORIES FROM @INC

You should normally only add directories to @INC. If you need to delete directories from @INC take care to only delete those which you added yourself or which you are certain are not needed by other modules in your script. Other modules may have added directories which they need for correct operation.

By default the `no lib` statement deletes the *first* instance of each named directory from @INC. To delete multiple instances of the same name from @INC you can specify the name multiple times.

To delete *all* instances of *all* the specified names from @INC you can specify ‘:ALL’ as the first parameter of `no lib`. For example:

```
    no lib qw(:ALL .);
```

For each directory in LIST (called `$dir` here) the lib module also checks to see if a directory called `$dir/$archname/auto` exists. If so the `$dir/$archname` directory is assumed to be a corresponding architecture specific directory and is also deleted from @INC.

If LIST includes both `$dir` and `$dir/$archname` then `$dir/$archname` will be deleted from @INC twice (if `$dir/$archname/auto` exists).

RESTORING ORIGINAL @INC

When the lib module is first loaded it records the current value of @INC in an array `@lib::ORIG_INC`. To restore @INC to that value you can say

```
    @INC = @lib::ORIG_INC;
```

SEE ALSO

FindBin − optional module which deals with paths relative to the source file.

AUTHOR

Tim Bunce, 2nd June 1995.

NAME

locale − Perl pragma to use and avoid POSIX locales for built−in operations

SYNOPSIS

```
    @x = sort @y;       # ASCII sorting order
    {
        use locale;
        @x = sort @y;   # Locale−defined sorting order
    }
    @x = sort @y;       # ASCII sorting order again
```

DESCRIPTION

This pragma tells the compiler to enable (or disable) the use of POSIX locales for built−in operations (LC_CTYPE for regular expressions, and LC_COLLATE for string comparison). Each "use locale" or "no locale" affects statements to the end of the enclosing BLOCK.

NAME

overload − Package for overloading perl operations

SYNOPSIS

```
    package SomeThing;

    use overload
        '+' => \&myadd,
        '−' => \&mysub;
        # etc
    ...

    package main;
    $a = new SomeThing 57;
    $b=5+$a;
    ...
    if (overload::Overloaded $b) {...}
    ...
    $strval = overload::StrVal $b;
```

CAVEAT SCRIPTOR

Overloading of operators is a subject not to be taken lightly. Neither its precise implementation, syntax, nor semantics are 100% endorsed by Larry Wall. So any of these may be changed at some point in the future.

DESCRIPTION

Declaration of overloaded functions

The compilation directive

```
    package Number;
    use overload
        "+" => \&add,
        "*=" => "muas";
```

declares function `Number::add()` for addition, and method `muas()` in the "class" `Number` (or one of its base classes) for the assignment form `*=` of multiplication.

Arguments of this directive come in (key, value) pairs. Legal values are values legal inside a `&{ ... }` call, so the name of a subroutine, a reference to a subroutine, or an anonymous subroutine will all work. Note that values specified as strings are interpreted as methods, not subroutines. Legal keys are listed below.

The subroutine `add` will be called to execute `$a+$b` if `$a` is a reference to an object blessed into the package `Number`, or if `$a` is not an object from a package with defined mathemagic addition, but `$b` is a reference to a `Number`. It can also be called in other situations, like `$a+=7,` or `$a++.` See *MAGIC AUTOGENERATION*. (Mathemagical methods refer to methods triggered by an overloaded mathematical operator.)

Since overloading respects inheritance via the @ISA hierarchy, the above declaration would also trigger overloading of `+` and `*=` in all the packages which inherit from `Number`.

Calling Conventions for Binary Operations

The functions specified in the `use overload ...` directive are called with three (in one particular case with four, see *Last Resort*) arguments. If the corresponding operation is binary, then the first two arguments are the two arguments of the operation. However, due to general object calling conventions, the first argument should always be an object in the package, so in the situation of `7+$a,` the order of the arguments is interchanged. It probably does not matter when implementing the addition method, but whether the arguments are reversed is vital to the subtraction method. The method can query this information by examining the third argument, which can take three different values:

FALSE	the order of arguments is as in the current operation.
TRUE	the arguments are reversed.
`undef`	the current operation is an assignment variant (as in `$a+=7),` but the usual function is called instead. This additional information can be used to generate some optimizations.

Calling Conventions for Unary Operations

Unary operation are considered binary operations with the second argument being `undef`. Thus the functions that overloads `{"++"}` is called with arguments `($a,undef,'')` when `$a++` is executed.

Overloadable Operations

The following symbols can be specified in `use overload`:

• *Arithmetic operations*

```
"+", "+=", "-", "-=", "*", "*=", "/", "/=", "%", "%=",
"**", "**=", "<<", "<<=", ">>", ">>=", "x", "x=", ".", ".=",
```

For these operations a substituted non−assignment variant can be called if the assignment variant is not available. Methods for operations "+", "−", "+=", and "−=" can be called to automatically generate increment and decrement methods. The operation "−" can be used to autogenerate missing methods for unary minus or `abs`.

• *Comparison operations*

```
"<",  "<=", ">",  ">=", "==", "!=", "<=>",
"lt", "le", "gt", "ge", "eq", "ne", "cmp",
```

If the corresponding "spaceship" variant is available, it can be used to substitute for the missing operation. During `sort`ing arrays, `cmp` is used to compare values subject to `use overload`.

• *Bit operations*

```
"&", "^", "|", "neg", "!", "~",
```

"`neg`" stands for unary minus. If the method for `neg` is not specified, it can be autogenerated using the method for subtraction. If the method for "`!`" is not specified, it can be autogenerated using the methods for "`bool`", or "`\"\"`", or "`0+`".

• *Increment and decrement*

```
"++", "--",
```

If undefined, addition and subtraction methods can be used instead. These operations are called both in prefix and postfix form.

• *Transcendental functions*

```
"atan2", "cos", "sin", "exp", "abs", "log", "sqrt",
```

If `abs` is unavailable, it can be autogenerated using methods for "<" or "<=>" combined with either unary minus or subtraction.

• *Boolean, string and numeric conversion*

```
"bool", "\"\"", "0+",
```

If one or two of these operations are unavailable, the remaining ones can be used instead. `bool` is used in the flow control operators (like `while`) and for the ternary "`?:`" operation. These functions can return any arbitrary Perl value. If the corresponding operation for this value is overloaded too, that operation will be called again with this value.

• *Special*

```
"nomethod", "fallback", "=",
```

see *SPECIAL SYMBOLS FOR `use overload`*.

See *"Fallback"* for an explanation of when a missing method can be autogenerated.

Inheritance and overloading

Inheritance interacts with overloading in two ways.

Strings as values of `use overload` directive

If `value` in

```
use overload key => value;
```

is a string, it is interpreted as a method name.

Overloading of an operation is inherited by derived classes

Any class derived from an overloaded class is also overloaded. The set of overloaded methods is the union of overloaded methods of all the ancestors. If some method is overloaded in several ancestor, then which description will be used is decided by the usual inheritance rules:

If `A` inherits from `B` and `C` (in this order), `B` overloads `+` with `\&D::plus_sub,` and `C` overloads `+` by `"plus_meth"`, then the subroutine `D::plus_sub` will be called to implement operation `+` for an object in package `A`.

Note that since the value of the `fallback` key is not a subroutine, its inheritance is not governed by the above rules. In the current implementation, the value of `fallback` in the first overloaded ancestor is used, but this is accidental and subject to change.

SPECIAL SYMBOLS FOR `use overload`

Three keys are recognized by Perl that are not covered by the above description.

Last Resort

`"nomethod"` should be followed by a reference to a function of four parameters. If defined, it is called when the overloading mechanism cannot find a method for some operation. The first three arguments of this function coincide with the arguments for the corresponding method if it were found, the fourth argument is the symbol corresponding to the missing method. If several methods are tried, the last one is used. Say, `1-$a` can be equivalent to

```
    &nomethodMethod($a,1,1,"-")
```

if the pair `"nomethod" => "nomethodMethod"` was specified in the `use overload` directive.

If some operation cannot be resolved, and there is no function assigned to `"nomethod"`, then an exception will be raised via `die()`— unless `"fallback"` was specified as a key in `use overload` directive.

Fallback

The key `"fallback"` governs what to do if a method for a particular operation is not found. Three different cases are possible depending on the value of `"fallback"`:

- `undef` Perl tries to use a substituted method (see *MAGIC AUTOGENERATION*). If this fails, it then tries to calls `"nomethod"` value; if missing, an exception will be raised.
- TRUE The same as for the `undef` value, but no exception is raised. Instead, it silently reverts to what it would have done were there no `use overload` present.
- defined, but FALSE No autogeneration is tried. Perl tries to call `"nomethod"` value, and if this is missing, raises an exception.

Note. `"fallback"` inheritance via @ISA is not carved in stone yet, see *"Inheritance and overloading"*.

Copy Constructor

The value for `"="` is a reference to a function with three arguments, i.e., it looks like the other values in `use overload`. However, it does not overload the Perl assignment operator. This would go against Camel hair.

This operation is called in the situations when a mutator is applied to a reference that shares its object with some other reference, such as

```
    $a=$b;
    $a++;
```

To make this change `$a` and not change `$b,` a copy of `$$a` is made, and `$a` is assigned a reference to this new object. This operation is done during execution of the `$a++,` and not during the assignment, (so before the increment `$$a` coincides with `$$b).` This is only done if `++` is expressed via a method for `'++'` or `'+='`. Note that if this operation is expressed via `'+'` a nonmutator, i.e., as in

```
    $a=$b;
    $a=$a+1;
```

then `$a` does not reference a new copy of `$$a,` since `$$a` does not appear as lvalue when the above code is executed.

If the copy constructor is required during the execution of some mutator, but a method for `'='` was not specified, it can be autogenerated as a string copy if the object is a plain scalar.

Example

The actually executed code for

```
    $a=$b;
    Something else which does not modify $a or $b....
    ++$a;
```

may be

```
    $a=$b;
    Something else which does not modify $a or $b....
    $a = $a->clone(undef,"");
    $a->incr(undef,"");
```

if `$b` was mathemagical, and `'++'` was overloaded with `\&incr,` `'='` was overloaded with `\&clone.`

MAGIC AUTOGENERATION

If a method for an operation is not found, and the value for `"fallback"` is TRUE or undefined, Perl tries to autogenerate a substitute method for the missing operation based on the defined operations. Autogenerated method substitutions are possible for the following operations:

Assignment forms of arithmetic operations
: `$a+=$b` can use the method for `"+"` if the method for `"+="` is not defined.

Conversion operations
: String, numeric, and boolean conversion are calculated in terms of one another if not all of them are defined.

Increment and decrement
: The `++$a` operation can be expressed in terms of `$a+=1` or `$a+1,` and `$a--` in terms of `$a-=1` and `$a-1.`

`abs($a)`
: can be expressed in terms of `$a<0` and `-$a` (or `0-$a).`

Unary minus
: can be expressed in terms of subtraction.

Negation
: `!` and `not` can be expressed in terms of boolean conversion, or string or numerical conversion.

Concatenation can be expressed in terms of string conversion.

Comparison operations

can be expressed in terms of its "spaceship" counterpart: either `<=>` or `cmp`:

```
<, >, <=, >=, ==, !=        in terms of <=>
lt, gt, le, ge, eq, ne      in terms of cmp
```

Copy operator can be expressed in terms of an assignment to the dereferenced value, if this value is a scalar and not a reference.

WARNING

The restriction for the comparison operation is that even if, for example, ‘`cmp`’ should return a blessed reference, the autogenerated ‘`lt`’ function will produce only a standard logical value based on the numerical value of the result of ‘`cmp`’. In particular, a working numeric conversion is needed in this case (possibly expressed in terms of other conversions).

Similarly, `.=` and `x=` operators lose their mathemagical properties if the string conversion substitution is applied.

When you `chop()` a mathemagical object it is promoted to a string and its mathemagical properties are lost. The same can happen with other operations as well.

Run−time Overloading

Since all `use` directives are executed at compile−time, the only way to change overloading during run−time is to

```
eval 'use overload "+" => \&addmethod';
```

You can also use

```
eval 'no overload "+", "--", "<="';
```

though the use of these constructs during run−time is questionable.

Public functions

Package `overload.pm` provides the following public functions:

overload::StrVal(arg)

Gives string value of `arg` as in absence of stringify overloading.

overload::Overloaded(arg)

Returns true if `arg` is subject to overloading of some operations.

overload::Method(obj,op)

Returns `undef` or a reference to the method that implements `op`.

IMPLEMENTATION

What follows is subject to change RSN.

The table of methods for all operations is cached in magic for the symbol table hash for the package. The cache is invalidated during processing of `use overload`, `no overload`, new function definitions, and changes in @ISA. However, this invalidation remains unprocessed until the next `bless`ing into the package. Hence if you want to change overloading structure dynamically, you‘ll need an additional (fake) `bless`ing to update the table.

(Every SVish thing has a magic queue, and magic is an entry in that queue. This is how a single variable may participate in multiple forms of magic simultaneously. For instance, environment variables regularly have two forms at once: their %ENV magic and their taint magic. However, the magic which implements overloading is applied to the stashes, which are rarely used directly, thus should not slow down Perl.)

If an object belongs to a package using overload, it carries a special flag. Thus the only speed penalty during arithmetic operations without overloading is the checking of this flag.

In fact, if `use overload` is not present, there is almost no overhead for overloadable operations, so most programs should not suffer measurable performance penalties. A considerable effort was made to minimize the overhead when overload is used in some package, but the arguments in question do not belong to packages using overload. When in doubt, test your speed with `use overload` and without it. So far there have been no reports of substantial speed degradation if Perl is compiled with optimization turned on.

There is no size penalty for data if overload is not used. The only size penalty if overload is used in some package is that *all* the packages acquire a magic during the next `bless`ing into the package. This magic is three−words−long for packages without overloading, and carries the cache tabel if the package is overloaded.

Copying (`$a=$b`) is shallow; however, a one−level−deep copying is carried out before any operation that can imply an assignment to the object `$a` (or `$b`) refers to, like `$a++.` You can override this behavior by defining your own copy constructor (see *"Copy Constructor"*).

It is expected that arguments to methods that are not explicitly supposed to be changed are constant (but this is not enforced).

AUTHOR

Ilya Zakharevich <***ilya@math.mps.ohio−state.edu***>.

DIAGNOSTICS

When Perl is run with the **−Do** switch or its equivalent, overloading induces diagnostic messages.

Using the `m` command of Perl debugger (see *perldebug*) one can deduce which operations are overloaded (and which ancestor triggers this overloading). Say, if `eq` is overloaded, then the method `(eq` is shown by debugger. The method `()` corresponds to the `fallback` key (in fact a presence of this method shows that this package has overloading enabled, and it is what is used by the `Overloaded` function).

BUGS

Because it is used for overloading, the per−package hash %OVERLOAD now has a special meaning in Perl. The symbol table is filled with names looking like line−noise.

For the purpose of inheritance every overloaded package behaves as if `fallback` is present (possibly undefined). This may create interesting effects if some package is not overloaded, but inherits from two overloaded packages.

This document is confusing.

NAME

sigtrap − Perl pragma to enable simple signal handling

SYNOPSIS

```
    use sigtrap;
    use sigtrap qw(stack−trace old−interface−signals);  # equivalent
    use sigtrap qw(BUS SEGV PIPE ABRT);
    use sigtrap qw(die INT QUIT);
    use sigtrap qw(die normal−signals);
    use sigtrap qw(die untrapped normal−signals);
    use sigtrap qw(die untrapped normal−signals
                    stack−trace any error−signals);
    use sigtrap ’handler’ => \&my_handler, ’normal−signals’;
    use sigtrap qw(handler my_handler normal−signals
                    stack−trace error−signals);
```

DESCRIPTION

The **sigtrap** pragma is a simple interface to installing signal handlers. You can have it install one of two handlers supplied by **sigtrap** itself (one which provides a Perl stack trace and one which simply `die()`s), or alternately you can supply your own handler for it to install. It can be told only to install a handler for signals which are either untrapped or ignored. It has a couple of lists of signals to trap, plus you can supply your own list of signals.

The arguments passed to the `use` statement which invokes **sigtrap** are processed in order. When a signal name or the name of one of **sigtrap**‘s signal lists is encountered a handler is immediately installed, when an option is encountered it affects subsequently installed handlers.

OPTIONS

SIGNAL HANDLERS

These options affect which handler will be used for subsequently installed signals.

stack−trace

The handler used for subsequently installed signals outputs a Perl stack trace to STDERR and then tries to dump core. This is the default signal handler.

die The handler used for subsequently installed signals calls `die` (actually `croak`) with a message indicating which signal was caught.

handler *your−handler*

your−handler will be used as the handler for subsequently installed signals. *your−handler* can be any value which is valid as an assignment to an element of `%SIG`.

SIGNAL LISTS

sigtrap has a few built−in lists of signals to trap. They are:

normal−signals

These are the signals which a program might normally expect to encounter and which by default cause it to terminate. They are HUP, INT, PIPE and TERM.

error−signals

These signals usually indicate a serious problem with the Perl interpreter or with your script. They are ABRT, BUS, EMT, FPE, ILL, QUIT, SEGV, SYS and TRAP.

old−interface−signals

These are the signals which were trapped by default by the old **sigtrap** interface, they are ABRT, BUS, EMT, FPE, ILL, PIPE, QUIT, SEGV, SYS, TERM, and TRAP. If no signals or signals lists are passed to **sigtrap**, this list is used.

For each of these three lists, the collection of signals set to be trapped is checked before trapping; if your architecture does not implement a particular signal, it will not be trapped but rather silently ignored.

OTHER

untrapped

This token tells **sigtrap** to install handlers only for subsequently listed signals which aren‘t already trapped or ignored.

any

This token tells **sigtrap** to install handlers for all subsequently listed signals. This is the default behavior.

signal

Any argument which looks like a signal name (that is, `/^[A−Z][A−Z0−9]*$/`) indicates that **sigtrap** should install a handler for that name.

number

Require that at least version *number* of **sigtrap** is being used.

EXAMPLES

Provide a stack trace for the old−interface−signals:

```
use sigtrap;
```

Ditto:

```
use sigtrap qw(stack−trace old−interface−signals);
```

Provide a stack trace on the 4 listed signals only:

```
use sigtrap qw(BUS SEGV PIPE ABRT);
```

Die on INT or QUIT:

```
use sigtrap qw(die INT QUIT);
```

Die on HUP, INT, PIPE or TERM:

```
use sigtrap qw(die normal−signals);
```

Die on HUP, INT, PIPE or TERM, except don‘t change the behavior for signals which are already trapped or ignored:

```
use sigtrap qw(die untrapped normal−signals);
```

Die on receipt one of an of the **normal−signals** which is currently **untrapped**, provide a stack trace on receipt of **any** of the **error−signals**:

```
use sigtrap qw(die untrapped normal−signals
                stack−trace any error−signals);
```

Install `my_handler()` as the handler for the **normal−signals**:

```
use sigtrap ’handler’, \&my_handler, ’normal−signals’;
```

Install `my_handler()` as the handler for the normal−signals, provide a Perl stack trace on receipt of one of the error−signals:

```
use sigtrap qw(handler my_handler normal−signals
                stack−trace error−signals);
```

NAME

strict − Perl pragma to restrict unsafe constructs

SYNOPSIS

```
    use strict;

    use strict "vars";
    use strict "refs";
    use strict "subs";

    use strict;
    no strict "vars";
```

DESCRIPTION

If no import list is supplied, all possible restrictions are assumed. (This is the safest mode to operate in, but is sometimes too strict for casual programming.) Currently, there are three possible things to be strict about: "subs", "vars", and "refs".

`strict refs`

This generates a runtime error if you use symbolic references (see *perlref*).

```
    use strict 'refs';
    $ref = \$foo;
    print $$ref;        # ok
    $ref = "foo";
    print $$ref;        # runtime error; normally ok
```

`strict vars`

This generates a compile−time error if you access a variable that wasn‘t localized via `my()` or wasn‘t fully qualified. Because this is to avoid variable suicide problems and subtle dynamic scoping issues, a merely `local()` variable isn‘t good enough. See *my* and *local*.

```
    use strict 'vars';
    $X::foo = 1;         # ok, fully qualified
    my $foo = 10;        # ok, my() var
    local $foo = 9;      # blows up
```

The `local()` generated a compile−time error because you just touched a global name without fully qualifying it.

`strict subs`

This disables the poetry optimization, generating a compile−time error if you try to use a bareword identifier that‘s not a subroutine, unless it appears in curly braces or on the left hand side of the "=>" symbol.

```
    use strict 'subs';
    $SIG{PIPE} = Plumber;       # blows up
    $SIG{PIPE} = "Plumber";     # just fine: bareword in curlies always ok
    $SIG{PIPE} = \&Plumber;     # preferred form
```

See *Pragmatic Modules*.

NAME

subs − Perl pragma to predeclare sub names

SYNOPSIS

```
use subs qw(frob);
frob 3..10;
```

DESCRIPTION

This will predeclare all the subroutine whose names are in the list, allowing you to use them without parentheses even before they‘re declared.

Unlike pragmas that affect the `$^H` hints variable, the `use vars` and `use subs` declarations are not BLOCK−scoped. They are thus effective for the entire file in which they appear. You may not rescind such declarations with `no vars` or `no subs`.

See *Pragmatic Modules* and *strict subs*.

NAME

vars − Perl pragma to predeclare global variable names

SYNOPSIS

```
    use vars qw($frob @mung %seen);
```

DESCRIPTION

This will predeclare all the variables whose names are in the list, allowing you to use them under "use strict", and disabling any typo warnings.

Unlike pragmas that affect the `$^H` hints variable, the `use vars` and `use subs` declarations are not BLOCK−scoped. They are thus effective for the entire file in which they appear. You may not rescind such declarations with `no vars` or `no subs`.

Packages such as the **AutoLoader** and **SelfLoader** that delay loading of subroutines within packages can create problems with package lexicals defined using `my()`. While the **vars** pragma cannot duplicate the effect of package lexicals (total transparency outside of the package), it can act as an acceptable substitute by pre−declaring global symbols, ensuring their availability to to the later−loaded routines.

See *Pragmatic Modules*.

NAME

Config − access Perl configuration information

SYNOPSIS

```
use Config;
if ($Config{'cc'} =~ /gcc/) {
    print "built by gcc\n";
}

use Config qw(myconfig config_sh config_vars);

print myconfig();

print config_sh();

config_vars(qw(osname archname));
```

DESCRIPTION

The Config module contains all the information that was available to the `Configure` program at Perl build time (over 900 values).

Shell variables from the ***config.sh*** file (written by Configure) are stored in the readonly−variable `%Config`, indexed by their names.

Values stored in config.sh as 'undef' are returned as undefined values. The perl `exists` function can be used to check if a named variable exists.

`myconfig()`

Returns a textual summary of the major perl configuration values. See also `−V` in *Switches*.

`config_sh()`

Returns the entire perl configuration information in the form of the original config.sh shell variable assignment script.

config_vars(@names)

Prints to STDOUT the values of the named configuration variable. Each is printed on a separate line in the form:

```
name='value';
```

Names which are unknown are output as `name='UNKNOWN';`. See also `−V:name` in *Switches*.

EXAMPLE

Here's a more sophisticated example of using %Config:

```
use Config;
use strict;

my %sig_num;
my @sig_name;
unless($Config{sig_name} && $Config{sig_num}) {
    die "No sigs?";
} else {
    my @names = split ' ', $Config{sig_name};
    @sig_num{@names} = split ' ', $Config{sig_num};
    foreach (@names) {
        $sig_name[$sig_num{$_}] ||= $_;
    }
}

print "signal #17 = $sig_name[17]\n";
```

```
        if ($sig_num{ALRM}) {
            print "SIGALRM is $sig_num{ALRM}\n";
        }
```

WARNING

Because this information is not stored within the perl executable itself it is possible (but unlikely) that the information does not relate to the actual perl binary which is being used to access it.

The Config module is installed into the architecture and version specific library directory (`$Config{installarchlib}`) and it checks the perl version number when loaded.

NOTE

This module contains a good example of how to use tie to implement a cache and an example of how to make a tied variable readonly to those outside of it.

NAME

DynaLoader − Dynamically load C libraries into Perl code

`dl_error()`, `dl_findfile()`, `dl_expandspec()`, `dl_load_file()`, `dl_find_symbol()`, `dl_find_symbol_anywhere()`, `dl_undef_symbols()`, `dl_install_xsub()`, `dl_load_flags()`, `bootstrap()` − routines used by DynaLoader modules

SYNOPSIS

```
    package YourPackage;
    require DynaLoader;
    @ISA = qw(... DynaLoader ...);
    bootstrap YourPackage;

    # optional method for 'global' loading
    sub dl_load_flags { 0x01 }
```

DESCRIPTION

This document defines a standard generic interface to the dynamic linking mechanisms available on many platforms. Its primary purpose is to implement automatic dynamic loading of Perl modules.

This document serves as both a specification for anyone wishing to implement the DynaLoader for a new platform and as a guide for anyone wishing to use the DynaLoader directly in an application.

The DynaLoader is designed to be a very simple high−level interface that is sufficiently general to cover the requirements of SunOS, HP−UX, NeXT, Linux, VMS and other platforms.

It is also hoped that the interface will cover the needs of OS/2, NT etc and also allow pseudo−dynamic linking (using `ld -A` at runtime).

It must be stressed that the DynaLoader, by itself, is practically useless for accessing non−Perl libraries because it provides almost no Perl−to−C 'glue'. There is, for example, no mechanism for calling a C library function or supplying arguments. It is anticipated that any glue that may be developed in the future will be implemented in a separate dynamically loaded module.

DynaLoader Interface Summary

```
  @dl_library_path
  @dl_resolve_using
  @dl_require_symbols
  $dl_debug
  @dl_librefs
  @dl_modules
                                                  Implemented in:
  bootstrap($modulename)                               Perl
  @filepaths = dl_findfile(@names)                     Perl
  $flags = $modulename->dl_load_flags                  Perl
  $symref  = dl_find_symbol_anywhere($symbol)          Perl

  $libref  = dl_load_file($filename, $flags)           C
  $symref  = dl_find_symbol($libref, $symbol)          C
  @symbols = dl_undef_symbols()                        C
  dl_install_xsub($name, $symref [, $filename])        C
  $message = dl_error                                  C
```

@dl_library_path

The standard/default list of directories in which `dl_findfile()` will search for libraries etc. Directories are searched in order: `$dl_library_path[0]`, [1], ... etc

@dl_library_path is initialised to hold the list of 'normal' directories (***/usr/lib***, etc) determined by **Configure** (`$Config{'libpth'}`). This should ensure portability across a wide range of platforms.

@dl_library_path should also be initialised with any other directories that can be determined from the environment at runtime (such as LD_LIBRARY_PATH for SunOS).

After initialisation @dl_library_path can be manipulated by an application using push and unshift before calling `dl_findfile()`. Unshift can be used to add directories to the front of the search order either to save search time or to override libraries with the same name in the 'normal' directories.

The load function that `dl_load_file()` calls may require an absolute pathname. The `dl_findfile()` function and @dl_library_path can be used to search for and return the absolute pathname for the library/object that you wish to load.

@dl_resolve_using

A list of additional libraries or other shared objects which can be used to resolve any undefined symbols that might be generated by a later call to `load_file()`.

This is only required on some platforms which do not handle dependent libraries automatically. For example the Socket Perl extension library (***auto/Socket/Socket.so***) contains references to many socket functions which need to be resolved when it's loaded. Most platforms will automatically know where to find the 'dependent' library (e.g., ***/usr/lib/libsocket.so***). A few platforms need to to be told the location of the dependent library explicitly. Use @dl_resolve_using for this.

Example usage:

```
@dl_resolve_using = dl_findfile('-lsocket');
```

@dl_require_symbols

A list of one or more symbol names that are in the library/object file to be dynamically loaded. This is only required on some platforms.

@dl_librefs

An array of the handles returned by successful calls to `dl_load_file()`, made by bootstrap, in the order in which they were loaded. Can be used with `dl_find_symbol()` to look for a symbol in any of the loaded files.

@dl_modules

An array of module (package) names that have been bootstrap'ed.

`dl_error()`

Syntax:

```
$message = dl_error();
```

Error message text from the last failed DynaLoader function. Note that, similar to errno in unix, a successful function call does not reset this message.

Implementations should detect the error as soon as it occurs in any of the other functions and save the corresponding message for later retrieval. This will avoid problems on some platforms (such as SunOS) where the error message is very temporary (e.g., `dlerror()`).

`$dl_debug`

Internal debugging messages are enabled when `$dl_debug` is set true. Currently setting `$dl_debug` only affects the Perl side of the DynaLoader. These messages should help an application developer to resolve any DynaLoader usage problems.

`$dl_debug` is set to `$ENV{'PERL_DL_DEBUG'}` if defined.

For the DynaLoader developer/porter there is a similar debugging variable added to the C code (see dlutils.c) and enabled if Perl was built with the **−DDEBUGGING** flag. This can also be set via the PERL_DL_DEBUG environment variable. Set to 1 for minimal information or higher for more.

`dl_findfile()`

Syntax:

```
    @filepaths = dl_findfile(@names)
```

Determine the full paths (including file suffix) of one or more loadable files given their generic names and optionally one or more directories. Searches directories in @dl_library_path by default and returns an empty list if no files were found.

Names can be specified in a variety of platform independent forms. Any names in the form **−lname** are converted into ***libname.****, where **.*** is an appropriate suffix for the platform.

If a name does not already have a suitable prefix and/or suffix then the corresponding file will be searched for by trying combinations of prefix and suffix appropriate to the platform: "`$name.o`", "lib`$name.*`" and "`$name`".

If any directories are included in @names they are searched before @dl_library_path. Directories may be specified as **−Ldir**. Any other names are treated as filenames to be searched for.

Using arguments of the form `−Ldir` and `−lname` is recommended.

Example:

```
    @dl_resolve_using = dl_findfile(qw(−L/usr/5lib −lposix));
```

`dl_expandspec()`

Syntax:

```
    $filepath = dl_expandspec($spec)
```

Some unusual systems, such as VMS, require special filename handling in order to deal with symbolic names for files (i.e., VMS‘s Logical Names).

To support these systems a `dl_expandspec()` function can be implemented either in the ***dl_*.xs*** file or code can be added to the autoloadable `dl_expandspec()` function in ***DynaLoader.pm***. See ***DynaLoader.pm*** for more information.

`dl_load_file()`

Syntax:

```
    $libref = dl_load_file($filename, $flags)
```

Dynamically load `$filename,` which must be the path to a shared object or library. An opaque ‘library reference’ is returned as a handle for the loaded object. Returns undef on error.

The `$flags` argument to alters dl_load_file behaviour. Assigned bits:

```
 0x01  make symbols available for linking later dl_load_file’s.
       (only known to work on Solaris 2 using dlopen(RTLD_GLOBAL))
       (ignored under VMS; this is a normal part of image linking)
```

(On systems that provide a handle for the loaded object such as SunOS and HPUX, `$libref` will be that handle. On other systems `$libref` will typically be `$filename` or a pointer to a buffer containing `$filename.` The application should not examine or alter `$libref` in any way.)

This is the function that does the real work. It should use the current values of @dl_require_symbols and @dl_resolve_using if required.

```
    SunOS: dlopen($filename)
    HP−UX: shl_load($filename)
    Linux: dld_create_reference(@dl_require_symbols); dld_link($filename)
    NeXT:  rld_load($filename, @dl_resolve_using)
    VMS:   lib$find_image_symbol($filename,$dl_require_symbols[0])
```

(The `dlopen()` function is also used by Solaris and some versions of Linux, and is a common choice when providing a "wrapper" on other mechanisms as is done in the OS/2 port.)

`dl_loadflags()`

Syntax:

```
$flags = dl_loadflags $modulename;
```

Designed to be a method call, and to be overridden by a derived class (i.e. a class which has DynaLoader in its @ISA). The definition in DynaLoader itself returns 0, which produces standard behavior from `dl_load_file()`.

`dl_find_symbol()`

Syntax:

```
$symref = dl_find_symbol($libref, $symbol)
```

Return the address of the symbol `$symbol` or `undef` if not found. If the target system has separate functions to search for symbols of different types then `dl_find_symbol()` should search for function symbols first and then other types.

The exact manner in which the address is returned in `$symref` is not currently defined. The only initial requirement is that `$symref` can be passed to, and understood by, `dl_install_xsub()`.

```
SunOS: dlsym($libref, $symbol)
HP-UX: shl_findsym($libref, $symbol)
Linux: dld_get_func($symbol) and/or dld_get_symbol($symbol)
NeXT:  rld_lookup("_$symbol")
VMS:   lib$find_image_symbol($libref,$symbol)
```

`dl_find_symbol_anywhere()`

Syntax:

```
$symref = dl_find_symbol_anywhere($symbol)
```

Applies `dl_find_symbol()` to the members of @dl_librefs and returns the first match found.

`dl_undef_symbols()`

Example

```
@symbols = dl_undef_symbols()
```

Return a list of symbol names which remain undefined after `load_file()`. Returns `()` if not known. Don‘t worry if your platform does not provide a mechanism for this. Most do not need it and hence do not provide it, they just return an empty list.

`dl_install_xsub()`

Syntax:

```
dl_install_xsub($perl_name, $symref [, $filename])
```

Create a new Perl external subroutine named `$perl_name` using `$symref` as a pointer to the function which implements the routine. This is simply a direct call to `newXSUB()`. Returns a reference to the installed function.

The `$filename` parameter is used by Perl to identify the source file for the function if required by `die()`, `caller()` or the debugger. If `$filename` is not defined then "DynaLoader" will be used.

`bootstrap()`

Syntax:

bootstrap(`$module)`

This is the normal entry point for automatic dynamic loading in Perl.

It performs the following actions:

- locates an auto/`$module` directory by searching @INC
- uses `dl_findfile()` to determine the filename to load
- sets @dl_require_symbols to `("boot_$module")`
- executes an ***auto/$module/$module.bs*** file if it exists (typically used to add to @dl_resolve_using any files which are required to load the module on the current platform)
- calls `dl_load_flags()` to determine how to load the file.
- calls `dl_load_file()` to load the file
- calls `dl_undef_symbols()` and warns if any symbols are undefined
- calls `dl_find_symbol()` for "boot_`$module"`
- calls `dl_install_xsub()` to install it as "`${module}::bootstrap"`
- calls `&{"${module}::bootstrap"}` to bootstrap the module (actually it uses the function reference returned by dl_install_xsub for speed)

AUTHOR

Tim Bunce, 11 August 1994.

This interface is based on the work and comments of (in no particular order): Larry Wall, Robert Sanders, Dean Roehrich, Jeff Okamoto, Anno Siegel, Thomas Neumann, Paul Marquess, Charles Bailey, myself and others.

Larry Wall designed the elegant inherited bootstrap mechanism and implemented the first Perl 5 dynamic loader using it.

Solaris global loading added by Nick Ing−Simmons with design/coding assistance from Tim Bunce, January 1996.

NAME

UNIVERSAL − base class for ALL classes (blessed references)

SYNOPSIS

```
    use UNIVERSAL qw(isa);

    $yes = isa($ref, "HASH");
    $io = $fd−>isa("IO::Handle");
    $sub = $obj−>can('print');
```

DESCRIPTION

`UNIVERSAL` is the base class which all bless references will inherit from, see *perlobj*

`UNIVERSAL` provides the following methods

isa (TYPE)

`isa` returns *true* if `REF` is blessed into package `TYPE` or inherits from package `TYPE`.

`isa` can be called as either a static or object method call.

can (METHOD)

`can` checks if the 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.

`can` can be called as either a static or object method call.

VERSION ([REQUIRE])

`VERSION` will return the value of the variable `$VERSION` in the package the object is blessed into. If `REQUIRE` is given then it will do a comparison and die if the package version is not greater than or equal to `REQUIRE`.

`VERSION` can be called as either a static or object method call.

`UNIVERSAL` also optionally exports the following subroutines

isa (REF, TYPE)

`isa` returns *true* if the first argument is a reference and either of the following statements is true.

`REF` is a blessed reference and is blessed into package `TYPE` or inherits from package `TYPE`

`REF` is a reference to a `TYPE` of perl variable (er 'HASH')

NAME

a2p − Awk to Perl translator

SYNOPSIS

a2p [options] filename

DESCRIPTION

A2p takes an awk script specified on the command line (or from standard input) and produces a comparable *perl* script on the standard output.

Options

Options include:

−D<number>

sets debugging flags.

−F<character>

tells a2p that this awk script is always invoked with this **−F** switch.

−n<fieldlist>

specifies the names of the input fields if input does not have to be split into an array. If you were translating an awk script that processes the password file, you might say:

```
        a2p −7 −nlogin.password.uid.gid.gcos.shell.home
```

Any delimiter can be used to separate the field names.

−<number>

causes a2p to assume that input will always have that many fields.

"Considerations"

A2p cannot do as good a job translating as a human would, but it usually does pretty well. There are some areas where you may want to examine the perl script produced and tweak it some. Here are some of them, in no particular order.

There is an awk idiom of putting `int()` around a string expression to force numeric interpretation, even though the argument is always integer anyway. This is generally unneeded in perl, but a2p can‘t tell if the argument is always going to be integer, so it leaves it in. You may wish to remove it.

Perl differentiates numeric comparison from string comparison. Awk has one operator for both that decides at run time which comparison to do. A2p does not try to do a complete job of awk emulation at this point. Instead it guesses which one you want. It‘s almost always right, but it can be spoofed. All such guesses are marked with the comment "`#???`". You should go through and check them. You might want to run at least once with the **−w** switch to perl, which will warn you if you use == where you should have used eq.

Perl does not attempt to emulate the behavior of awk in which nonexistent array elements spring into existence simply by being referenced. If somehow you are relying on this mechanism to create null entries for a subsequent for...in, they won‘t be there in perl.

If a2p makes a split line that assigns to a list of variables that looks like (Fld1, Fld2, Fld3...) you may want to rerun a2p using the **−n** option mentioned above. This will let you name the fields throughout the script. If it splits to an array instead, the script is probably referring to the number of fields somewhere.

The exit statement in awk doesn‘t necessarily exit; it goes to the END block if there is one. Awk scripts that do contortions within the END block to bypass the block under such circumstances can be simplified by removing the conditional in the END block and just exiting directly from the perl script.

Perl has two kinds of array, numerically−indexed and associative. Perl associative arrays are called "hashes". Awk arrays are usually translated to hashes, but if you happen to know that the index is always going to be numeric you could change the {...} to [...]. Iteration over a hash is done using the `keys()` function, but iteration over an array is NOT. You might need to modify any loop that iterates over such an array.

Awk starts by assuming OFMT has the value %.6g. Perl starts by assuming its equivalent, `$#,` to have the value %.20g. You‘ll want to set `$#` explicitly if you use the default value of OFMT.

Near the top of the line loop will be the split operation that is implicit in the awk script. There are times when you can move this down past some conditionals that test the entire record so that the split is not done as often.

For aesthetic reasons you may wish to change the array base `$[` from 1 back to perl‘s default of 0, but remember to change all array subscripts AND all `substr()` and `index()` operations to match.

Cute comments that say "# Here is a workaround because awk is dumb" are passed through unmodified.

Awk scripts are often embedded in a shell script that pipes stuff into and out of awk. Often the shell script wrapper can be incorporated into the perl script, since perl can start up pipes into and out of itself, and can do other things that awk can‘t do by itself.

Scripts that refer to the special variables RSTART and RLENGTH can often be simplified by referring to the variables `$‘,` `$&` and `$‘,` as long as they are within the scope of the pattern match that sets them.

The produced perl script may have subroutines defined to deal with awk‘s semantics regarding getline and print. Since a2p usually picks correctness over efficiency. it is almost always possible to rewrite such code to be more efficient by discarding the semantic sugar.

For efficiency, you may wish to remove the keyword from any return statement that is the last statement executed in a subroutine. A2p catches the most common case, but doesn‘t analyze embedded blocks for subtler cases.

ARGV[0] translates to `$ARGV0,` but ARGV[n] translates to `$ARGV[$n].` A loop that tries to iterate over ARGV[0] won‘t find it.

ENVIRONMENT

A2p uses no environment variables.

AUTHOR

Larry Wall <***larry@wall.org***>

FILES

SEE ALSO

```
perl   The perl compiler/interpreter

s2p    sed to perl translator
```

DIAGNOSTICS

BUGS

It would be possible to emulate awk‘s behavior in selecting string versus numeric operations at run time by inspection of the operands, but it would be gross and inefficient. Besides, a2p almost always guesses right.

Storage for the awk syntax tree is currently static, and can run out.

NAME

c2ph, pstruct − Dump C structures as generated from `cc −g −S` stabs

SYNOPSIS

```
c2ph [−dpnP] [var=val] [files ...]
```

OPTIONS

```
Options:

−w  wide; short for: type_width=45 member_width=35 offset_width=8
−x  hex; short for:  offset_fmt=x offset_width=08 size_fmt=x size_width=04

−n  do not generate perl code  (default when invoked as pstruct)
−p  generate perl code         (default when invoked as c2ph)
−v  generate perl code, with C decls as comments

−i  do NOT recompute sizes for intrinsic datatypes
−a  dump information on intrinsics also

−t  trace execution
−d  spew reams of debugging output

−slist  give comma−separated list a structures to dump
```

DESCRIPTION

The following is the old c2ph.doc documentation by Tom Christiansen <tchrist@perl.com Date: 25 Jul 91 08:10:21 GMT

Once upon a time, I wrote a program called pstruct. It was a perl program that tried to parse out C structures and display their member offsets for you. This was especially useful for people looking at binary dumps or poking around the kernel.

Pstruct was not a pretty program. Neither was it particularly robust. The problem, you see, was that the C compiler was much better at parsing C than I could ever hope to be.

So I got smart: I decided to be lazy and let the C compiler parse the C, which would spit out debugger stabs for me to read. These were much easier to parse. It‘s still not a pretty program, but at least it‘s more robust.

Pstruct takes any .c or .h files, or preferably .s ones, since that‘s the format it is going to massage them into anyway, and spits out listings like this:

```
 struct tty {
   int                          tty.t_locker                         000      4
   int                          tty.t_mutex_index                    004      4
   struct tty *                 tty.t_tp_virt                        008      4
   struct clist                 tty.t_rawq                           00c     20
     int                        tty.t_rawq.c_cc                      00c      4
     int                        tty.t_rawq.c_cmax                    010      4
     int                        tty.t_rawq.c_cfx                     014      4
     int                        tty.t_rawq.c_clx                     018      4
     struct tty *               tty.t_rawq.c_tp_cpu                  01c      4
     struct tty *               tty.t_rawq.c_tp_iop                  020      4
     unsigned char *            tty.t_rawq.c_buf_cpu                 024      4
     unsigned char *            tty.t_rawq.c_buf_iop                 028      4
   struct clist                 tty.t_canq                           02c     20
     int                        tty.t_canq.c_cc                      02c      4
     int                        tty.t_canq.c_cmax                    030      4
     int                        tty.t_canq.c_cfx                     034      4
     int                        tty.t_canq.c_clx                     038      4
     struct tty *               tty.t_canq.c_tp_cpu                  03c      4
```

```
      struct tty *              tty.t_canq.c_tp_iop                040      4
      unsigned char *           tty.t_canq.c_buf_cpu               044      4
      unsigned char *           tty.t_canq.c_buf_iop               048      4
    struct clist                tty.t_outq                         04c     20
      int                       tty.t_outq.c_cc                    04c      4
      int                       tty.t_outq.c_cmax                  050      4
      int                       tty.t_outq.c_cfx                   054      4
      int                       tty.t_outq.c_clx                   058      4
      struct tty *              tty.t_outq.c_tp_cpu                05c      4
      struct tty *              tty.t_outq.c_tp_iop                060      4
      unsigned char *           tty.t_outq.c_buf_cpu               064      4
      unsigned char *           tty.t_outq.c_buf_iop               068      4
    (*int)()                    tty.t_oproc_cpu                    06c      4
    (*int)()                    tty.t_oproc_iop                    070      4
    (*int)()                    tty.t_stopproc_cpu                 074      4
    (*int)()                    tty.t_stopproc_iop                 078      4
    struct thread *             tty.t_rsel                         07c      4
```

etc.

Actually, this was generated by a particular set of options. You can control the formatting of each column, whether you prefer wide or fat, hex or decimal, leading zeroes or whatever.

All you need to be able to use this is a C compiler than generates BSD/GCC−style stabs. The **−g** option on native BSD compilers and GCC should get this for you.

To learn more, just type a bogus option, like **−\?**, and a long usage message will be provided. There are a fair number of possibilities.

If you‘re only a C programmer, than this is the end of the message for you. You can quit right now, and if you care to, save off the source and run it when you feel like it. Or not.

But if you‘re a perl programmer, then for you I have something much more wondrous than just a structure offset printer.

You see, if you call pstruct by its other incybernation, c2ph, you have a code generator that translates C code into perl code! Well, structure and union declarations at least, but that‘s quite a bit.

Prior to this point, anyone programming in perl who wanted to interact with C programs, like the kernel, was forced to guess the layouts of the C strutures, and then hardwire these into his program. Of course, when you took your wonderfully crafted program to a system where the sgtty structure was laid out differently, you program broke. Which is a shame.

We‘ve had Larry‘s h2ph translator, which helped, but that only works on cpp symbols, not real C, which was also very much needed. What I offer you is a symbolic way of getting at all the C structures. I‘ve couched them in terms of packages and functions. Consider the following program:

```
#!/usr/local/bin/perl

require 'syscall.ph';
require 'sys/time.ph';
require 'sys/resource.ph';

$ru = "\0" x &rusage'sizeof();

syscall(&SYS_getrusage, &RUSAGE_SELF, $ru)      && die "getrusage: $!";

@ru = unpack($t = &rusage'typedef(), $ru);

$utime =  $ru[ &rusage'ru_utime + &timeval'tv_sec  ]
       + ($ru[ &rusage'ru_utime + &timeval'tv_usec ]) / 1e6;
```

```
    $stime =  $ru[ &rusage'ru_stime + &timeval'tv_sec  ]
           + ($ru[ &rusage'ru_stime + &timeval'tv_usec ]) / 1e6;

    printf "you have used %8.3fs+%8.3fu seconds.\n", $utime, $stime;
```

As you see, the name of the package is the name of the structure. Regular fields are just their own names. Plus the following accessor functions are provided for your convenience:

```
    struct      This takes no arguments, and is merely the number of first-level
                elements in the structure.  You would use this for indexing
                into arrays of structures, perhaps like this

                    $usec = $u[ &user'u_utimer
                                + (&ITIMER_VIRTUAL * &itimerval'struct)
                                + &itimerval'it_value
                                + &timeval'tv_usec
                              ];

    sizeof      Returns the bytes in the structure, or the member if
                you pass it an argument, such as

                        &rusage'sizeof(&rusage'ru_utime)

    typedef     This is the perl format definition for passing to pack and
                unpack.  If you ask for the typedef of a nothing, you get
                the whole structure, otherwise you get that of the member
                you ask for.  Padding is taken care of, as is the magic to
                guarantee that a union is unpacked into all its aliases.
                Bitfields are not quite yet supported however.

    offsetof    This function is the byte offset into the array of that
                member.  You may wish to use this for indexing directly
                into the packed structure with vec() if you're too lazy
                to unpack it.

    typeof      Not to be confused with the typedef accessor function, this
                one returns the C type of that field.  This would allow
                you to print out a nice structured pretty print of some
                structure without knoning anything about it beforehand.
                No args to this one is a noop.  Someday I'll post such
                a thing to dump out your u structure for you.
```

The way I see this being used is like basically this:

```
        % h2ph <some_include_file.h  >  /usr/lib/perl/tmp.ph
        % c2ph  some_include_file.h  >> /usr/lib/perl/tmp.ph
        % install
```

It‘s a little tricker with c2ph because you have to get the includes right. I can‘t know this for your system, but it‘s not usually too terribly difficult.

The code isn‘t pretty as I mentioned — I never thought it would be a 1000− line program when I started, or I might not have begun. :−) But I would have been less cavalier in how the parts of the program communicated with each other, etc. It might also have helped if I didn‘t have to divine the makeup of the stabs on the fly, and then account for micro differences between my compiler and gcc.

Anyway, here it is. Should run on perl v4 or greater. Maybe less.

```
 --tom
```

NAME

h2ph − convert .h C header files to .ph Perl header files

SYNOPSIS

h2ph [headerfiles]

DESCRIPTION

h2ph converts any C header files specified to the corresponding Perl header file format. It is most easily run while in /usr/include:

```
        cd /usr/include; h2ph * sys/*
```

If run with no arguments, filters standard input to standard output.

ENVIRONMENT

No environment variables are used.

FILES

```
 /usr/include/*.h
 /usr/include/sys/*.h
```

etc.

AUTHOR

Larry Wall

SEE ALSO

perl(1)

DIAGNOSTICS

The usual warnings if it can‘t read or write the files involved.

BUGS

Doesn‘t construct the %sizeof array for you.

It doesn‘t handle all C constructs, but it does attempt to isolate definitions inside evals so that you can get at the definitions that it can translate.

It‘s only intended as a rough tool. You may need to dicker with the files produced.

NAME

h2xs − convert .h C header files to Perl extensions

SYNOPSIS

h2xs [**−AOPXcdf**] [**−v** version] [**−n** module_name] [**−p** prefix] [**−s** sub] [headerfile [extra_libraries]]

h2xs −h

DESCRIPTION

h2xs builds a Perl extension from any C header file. The extension will include functions which can be used to retrieve the value of any #define statement which was in the C header.

The *module_name* will be used for the name of the extension. If module_name is not supplied then the name of the header file will be used, with the first character capitalized.

If the extension might need extra libraries, they should be included here. The extension Makefile.PL will take care of checking whether the libraries actually exist and how they should be loaded. The extra libraries should be specified in the form −lm −lposix, etc, just as on the cc command line. By default, the Makefile.PL will search through the library path determined by Configure. That path can be augmented by including arguments of the form **−L/another/library/path** in the extra−libraries argument.

OPTIONS

−A Omit all autoload facilities. This is the same as **−c** but also removes the `require AutoLoader` statement from the .pm file.

−F Additional flags to specify to C preprocessor when scanning header for function declarations. Should not be used without **−x**.

−O Allows a pre−existing extension directory to be overwritten.

−P Omit the autogenerated stub POD section.

−X Omit the XS portion. Used to generate templates for a module which is not XS−based.

−c Omit `constant()` from the .xs file and corresponding specialised `AUTOLOAD` from the .pm file.

−d Turn on debugging messages.

−f Allows an extension to be created for a header even if that header is not found in /usr/include.

−h Print the usage, help and version for this h2xs and exit.

−n *module_name*

Specifies a name to be used for the extension, e.g., −n RPC::DCE

−p *prefix*

Specify a prefix which should be removed from the Perl function names, e.g., −p sec_rgy_ This sets up the XS **PREFIX** keyword and removes the prefix from functions that are autoloaded via the `constant()` mechansim.

−s *sub1,sub2*

Create a perl subroutine for the specified macros rather than autoload with the `constant()` subroutine. These macros are assumed to have a return type of **char ***, e.g.,
−s sec_rgy_wildcard_name,sec_rgy_wildcard_sid.

−v *version*

Specify a version number for this extension. This version number is added to the templates. The default is 0.01.

−x Automatically generate XSUBs basing on function declarations in the header file. The package `C::Scan` should be installed. If this option is specified, the name of the header file may look like `NAME1,NAME2`. In this case NAME1 is used instead of the specified string, but XSUBs are emitted only for the declarations included from file NAME2.

Note that some types of arguments/return−values for functions may result in XSUB−declarations/typemap−entries which need hand−editing. Such may be objects which cannot be converted from/to a pointer (like `long long`), pointers to functions, or arrays.

EXAMPLES

```
# Default behavior, extension is Rusers
h2xs rpcsvc/rusers

# Same, but extension is RUSERS
h2xs −n RUSERS rpcsvc/rusers

# Extension is rpcsvc::rusers. Still finds <rpcsvc/rusers.h>
h2xs rpcsvc::rusers

# Extension is ONC::RPC.  Still finds <rpcsvc/rusers.h>
h2xs −n ONC::RPC rpcsvc/rusers

# Without constant() or AUTOLOAD
h2xs −c rpcsvc/rusers

# Creates templates for an extension named RPC
h2xs −cfn RPC

# Extension is ONC::RPC.
h2xs −cfn ONC::RPC

# Makefile.PL will look for library −lrpc in
# additional directory /opt/net/lib
h2xs rpcsvc/rusers −L/opt/net/lib −lrpc

# Extension is DCE::rgynbase
# prefix "sec_rgy_" is dropped from perl function names
h2xs −n DCE::rgynbase −p sec_rgy_ dce/rgynbase

# Extension is DCE::rgynbase
# prefix "sec_rgy_" is dropped from perl function names
# subroutines are created for sec_rgy_wildcard_name and sec_rgy_wildcard_sid
h2xs −n DCE::rgynbase −p sec_rgy_ \
−s sec_rgy_wildcard_name,sec_rgy_wildcard_sid dce/rgynbase

# Make XS without defines in perl.h, but with function declarations
# visible from perl.h. Name of the extension is perl1.
# When scanning perl.h, define −DEXT=extern −DdEXT= −DINIT(x)=
# Extra backslashes below because the string is passed to shell.
# Note that a directory with perl header files would
#  be added automatically to include path.
h2xs −xAn perl1 −F "−DEXT=extern −DdEXT= −DINIT\(x\)=" perl.h

# Same with function declaration in proto.h as visible from perl.h.
h2xs −xAn perl2 perl.h,proto.h
```

ENVIRONMENT

No environment variables are used.

AUTHOR

Larry Wall and others

SEE ALSO

perl, *perlxstut*, *ExtUtils::MakeMaker*, and *AutoLoader*.

DIAGNOSTICS

The usual warnings if it cannot read or write the files involved.

NAME

perlbug − how to submit bug reports on Perl

SYNOPSIS

perlbug [**−v**] [**−a** *address*] [**−s** *subject*] [**−b** *body* | **−f** *file*] [**−r** *returnaddress*] [**−e** *editor*]
[**−c** *adminaddress* | **−C**] [**−S**] [**−t**] [**−d**] [**−h**]

DESCRIPTION

A program to help generate bug reports about perl or the modules that come with it, and mail them.

If you have found a bug with a non−standard port (one that was not part of the *standard distribution*), a binary distribution, or a non−standard module (such as Tk, CGI, etc), then please see the documentation that came with that distribution to determine the correct place to report bugs.

`perlbug` is designed to be used interactively. Normally no arguments will be needed. Simply run it, and follow the prompts.

If you are unable to run **perlbug** (most likely because you don‘t have a working setup to send mail that perlbug recognizes), you may have to compose your own report, and email it to **perlbug@perl.com**. You might find the **−d** option useful to get summary information in that case.

In any case, when reporting a bug, please make sure you have run through this checklist:

What version of perl you are running?

Type `perl −v` at the command line to find out.

Are you running the latest released version of perl?

Look at http://www.perl.com/ to find out. If it is not the latest released version, get that one and see whether your bug has been fixed. Note that bug reports about old versions of perl, especially those prior to the 5.0 release, are likely to fall upon deaf ears. You are on your own if you continue to use perl1 .. perl4.

Are you sure what you have is a bug?

A significant number of the bug reports we get turn out to be documented features in perl. Make sure the behavior you are witnessing doesn‘t fall under that category, by glancing through the documentation that comes with perl (we‘ll admit this is no mean task, given the sheer volume of it all, but at least have a look at the sections that *seem* relevant).

Be aware of the familiar traps that perl programmers of various hues fall into. See *perltrap*.

Try to study the problem under the perl debugger, if necessary. See *perldebug*.

Do you have a proper test case?

The easier it is to reproduce your bug, the more likely it will be fixed, because if no one can duplicate the problem, no one can fix it. A good test case has most of these attributes: fewest possible number of lines; few dependencies on external commands, modules, or libraries; runs on most platforms unimpeded; and is self−documenting.

A good test case is almost always a good candidate to be on the perl test suite. If you have the time, consider making your test case so that it will readily fit into the standard test suite.

Can you describe the bug in plain English?

The easier it is to understand a reproducible bug, the more likely it will be fixed. Anything you can provide by way of insight into the problem helps a great deal. In other words, try to analyse the problem to the extent you feel qualified and report your discoveries.

Can you fix the bug yourself?

A bug report which *includes a patch to fix it* will almost definitely be fixed. Use the `diff` program to generate your patches (`diff` is being maintained by the GNU folks as part of the **diffutils** package, so you should be able to get it from any of the GNU software repositories). If you do submit a patch, the cool−dude counter at perlbug@perl.com will register you as a savior of the world. Your patch may be returned with requests for changes, or requests for more detailed explanations about your fix.

Here are some clues for creating quality patches: Use the **−c** or **−u** switches to the diff program (to create a so−called context or unified diff). Make sure the patch is not reversed (the first argument to diff is typically the original file, the second argument your changed file). Make sure you test your patch by applying it with the `patch` program before you send it on its way. Try to follow the same style as the code you are trying to patch. Make sure your patch really does work (`make test`, if the thing you‘re patching supports it).

Can you use `perlbug` to submit the report?

perlbug will, amongst other things, ensure your report includes crucial information about your version of perl. If `perlbug` is unable to mail your report after you have typed it in, you may have to compose the message yourself, add the output produced by `perlbug −d` and email it to **perlbug@perl.com**. If, for some reason, you cannot run `perlbug` at all on your system, be sure to include the entire output produced by running `perl −V` (note the uppercase V).

Having done your bit, please be prepared to wait, to be told the bug is in your code, or even to get no reply at all. The perl maintainers are busy folks, so if your problem is a small one or if it is difficult to understand, they may not respond with a personal reply. If it is important to you that your bug be fixed, do monitor the `Changes` file in any development releases since the time you submitted the bug, and encourage the maintainers with kind words (but never any flames!). Feel free to resend your bug report if the next released version of perl comes out and your bug is still present.

OPTIONS

−a	Address to send the report to. Defaults to ‘perlbug@perl.com’.
−b	Body of the report. If not included on the command line, or in a file with **−f**, you will get a chance to edit the message.
−C	Don‘t send copy to administrator.
−c	Address to send copy of report to. Defaults to the address of the local perl administrator (recorded when perl was built).
−d	Data mode (the default if you redirect or pipe output). This prints out your configuration data, without mailing anything. You can use this with **−v** to get more complete data.
−e	Editor to use.
−f	File containing the body of the report. Use this to quickly send a prepared message.
−h	Prints a brief summary of the options.
−r	Your return address. The program will ask you to confirm its default if you don‘t use this option.
−S	Send without asking for confirmation.
−s	Subject to include with the message. You will be prompted if you don‘t supply one on the command line.
−t	Test mode. The target address defaults to ‘perlbug−test@perl.com’.
−v	Include verbose configuration data in the report.

AUTHORS

Kenneth Albanowski (<kjahds@kjahds.com>), subsequently *doc*tored by Gurusamy Sarathy (<gsar@umich.edu>), Tom Christiansen (<tchrist@perl.com>), and Nathan Torkington (<gnat@frii.com>).

SEE ALSO

perl(1), perldebug(1), perltrap(1), diff(1), patch(1)

BUGS

None known (guess what must have been used to report them?)

NAME

perldoc − Look up Perl documentation in pod format.

SYNOPSIS

perldoc [**−h**] [**−v**] [**−t**] [**−u**] [**−m**] [**−l**] PageName|ModuleName|ProgramName

perldoc −f BuiltinFunction

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.

OPTIONS

−h help

Prints out a brief help message.

−v verbose

Describes search for the item in detail.

−t text output

Display docs using plain text converter, instead of nroff. This may be faster, but it won‘t look as nice.

−u unformatted

Find docs only; skip reformatting by pod2*

−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 file name only

Display the file name of the module found.

−f perlfunc

The **−f** option followed by the name of a perl built in function will extract the documentation of this function from *perlfunc*.

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`. You make also give a partial or wrong−case name, such as "basename" for "File::Basename", but this will be slower, if there is more then one page with the same partial name, you will only get the first one.

ENVIRONMENT

Any switches in the `PERLDOC` environment variable will be used before the command line arguments. `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.)

AUTHOR

Kenneth Albanowski <kjahds@kjahds.com

Minor updates by Andy Dougherty <doughera@lafcol.lafayette.edu

NAME

pod2man − translate embedded Perl pod directives into man pages

SYNOPSIS

pod2man [**—section**=*manext*] [**—release**=*relpatch*] [**—center**=*string*] [**—date**=*string*] [**—fixed**=*font*] [**—official**] [**—lax**] *inputfile*

DESCRIPTION

pod2man converts its input file containing embedded pod directives (see *perlpod*) into nroff source suitable for viewing with nroff(1) or troff(1) using the man(7) macro set.

Besides the obvious pod conversions, **pod2man** also takes care of `func()`, func(n), and simple variable references like `$foo` or @bar so you don‘t have to use code escapes for them; complex expressions like `$fred{‘stuff‘}` will still need to be escaped, though. Other nagging little roffish things that it catches include translating the minus in something like foo−bar, making a long dash—like this—into a real em dash, fixing up "paired quotes", putting a little space after the parens in something like `func()`, making C++ and PI look right, making double underbars have a little tiny space between them, making ALLCAPS a teeny bit smaller in troff(1), and escaping backslashes so you don‘t have to.

OPTIONS

center
: Set the centered header to a specific string. The default is "User Contributed Perl Documentation", unless the `—official` flag is given, in which case the default is "Perl Programmers Reference Guide".

date
: Set the left−hand footer string to this value. By default, the modification date of the input file will be used.

fixed
: The fixed font to use for code refs. Defaults to CW.

official
: Set the default header to indicate that this page is of the standard release in case `—center` is not given.

release
: Set the centered footer. By default, this is the current perl release.

section
: Set the section for the `.TH` macro. The standard conventions on sections are to use 1 for user commands, 2 for system calls, 3 for functions, 4 for devices, 5 for file formats, 6 for games, 7 for miscellaneous information, and 8 for administrator commands. This works best if you put your Perl man pages in a separate tree, like ***/usr/local/perl/man/***. By default, section 1 will be used unless the file ends in ***.pm*** in which case section 3 will be selected.

lax
: Don‘t complain when required sections aren‘t present.

Anatomy of a Proper Man Page

For those not sure of the proper layout of a man page, here‘s an example of the skeleton of a proper man page. Head of the major headers should be setout as a `=head1` directive, and are historically written in the rather startling ALL UPPER CASE format, although this is not mandatory. Minor headers may be included using `=head2`, and are typically in mixed case.

NAME
: Mandatory section; should be a comma−separated list of programs or functions documented by this podpage, such as:

```
    foo, bar − programs to do something
```

SYNOPSIS
: A short usage summary for programs and functions, which may someday be deemed mandatory.

DESCRIPTION
: Long drawn out discussion of the program. It‘s a good idea to break this up into subsections using the `=head2` directives, like

```
    =head2 A Sample Subection

    =head2 Yet Another Sample Subection
```

OPTIONS
: Some people make this separate from the description.

RETURN VALUE
: What the program or function returns if successful.

ERRORS
: Exceptions, return codes, exit stati, and errno settings.

EXAMPLES
: Give some example uses of the program.

ENVIRONMENT
: Envariables this program might care about.

FILES
: All files used by the program. You should probably use the F<> for these.

SEE ALSO
: Other man pages to check out, like man(1), man(7), makewhatis(8), or catman(8).

NOTES
: Miscellaneous commentary.

CAVEATS
: Things to take special care with; sometimes called WARNINGS.

DIAGNOSTICS
: All possible messages the program can print out—and what they mean.

BUGS
: Things that are broken or just don‘t work quite right.

RESTRICTIONS
: Bugs you don‘t plan to fix :−)

AUTHOR
: Who wrote it (or AUTHORS if multiple).

HISTORY
: Programs derived from other sources sometimes have this, or you might keep a modification log here.

EXAMPLES

```
    pod2man program > program.1
    pod2man some_module.pm > /usr/perl/man/man3/some_module.3
    pod2man −−section=7 note.pod > note.7
```

DIAGNOSTICS

The following diagnostics are generated by **pod2man**. Items marked "(W)" are non−fatal, whereas the "(F)" errors will cause **pod2man** to immediately exit with a non−zero status.

bad option in paragraph %d of %s: “%s“ should be [%s]<%s

(W) If you start include an option, you should set it off as bold, italic, or code.

can‘t open %s: %s

(F) The input file wasn‘t available for the given reason.

Improper man page − no dash in NAME header in paragraph %d of %s

(W) The NAME header did not have an isolated dash in it. This is considered important.

Invalid man page − no NAME line in %s

(F) You did not include a NAME header, which is essential.

roff font should be 1 or 2 chars, not ‘%s’ (F)

(F) The font specified with the `−fixed` option was not a one− or two−digit roff font.

%s is missing required section: %s

(W) Required sections include NAME, DESCRIPTION, and if you‘re using a section starting with a 3, also a SYNOPSIS. Actually, not having a NAME is a fatal.

Unknown escape: %s in %s

(W) An unknown HTML entity (probably for an 8−bit character) was given via a `E<>` directive. Besides amp, lt, gt, and quot, recognized entities are Aacute, aacute, Acirc, acirc, AElig, aelig, Agrave, agrave, Aring, aring, Atilde, atilde, Auml, auml, Ccedil, ccedil, Eacute, eacute, Ecirc, ecirc, Egrave, egrave, ETH, eth, Euml, euml, Iacute, iacute, Icirc, icirc, Igrave, igrave, Iuml, iuml, Ntilde, ntilde, Oacute, oacute, Ocirc, ocirc, Ograve, ograve, Oslash, oslash, Otilde, otilde, Ouml, ouml, szlig, THORN, thorn, Uacute, uacute, Ucirc, ucirc, Ugrave, ugrave, Uuml, uuml, Yacute, yacute, and yuml.

Unmatched =back

(W) You have a `=back` without a corresponding `=over`.

Unrecognized pod directive: %s

(W) You specified a pod directive that isn‘t in the known list of `=head1`, `=head2`, `=item`, `=over`, `=back`, or `=cut`.

NOTES

If you would like to print out a lot of man page continuously, you probably want to set the C and D registers to set contiguous page numbering and even/odd paging, at least on some versions of man(7). Settting the F register will get you some additional experimental indexing:

```
troff −man −rC1 −rD1 −rF1 perl.1 perldata.1 perlsyn.1 ...
```

The indexing merely outputs messages via `.tm` for each major page, section, subsection, item, and any `X<>` directives.

RESTRICTIONS

None at this time.

BUGS

The =over and =back directives don‘t really work right. They take absolute positions instead of offsets, don‘t nest well, and making people count is suboptimal in any event.

AUTHORS

Original prototype by Larry Wall, but so massively hacked over by Tom Christiansen such that Larry probably doesn‘t recognize it anymore.

NAME

Pumpkin − Notes on handling the Perl Patch Pumpkin

SYNOPSIS

There is no simple synopsis, yet.

DESCRIPTION

This document attempts to begin to describe some of the considerations involved in patching and maintaining perl.

This document is still under construction, and still subject to significant changes. Still, I hope parts of it will be useful, so I‘m releasing it even though it‘s not done.

For the most part, it‘s a collection of anecdotal information that already assumes some familiarity with the Perl sources. I really need an introductory section that describes the organization of the sources and all the various auxiliary files that are part of the distribution.

Where Do I Get Perl Sources and Related Material?

The Comprehensive Perl Archive Network (or CPAN) is the place to go. There are many mirrors, but the easiest thing to use is probably http://www.perl.com/CPAN/README.html , which automatically points you to a mirror site "close" to you.

Perl5−porters mailing list

The mailing list perl5−porters@perl.org is the main group working with the development of perl. If you‘re interested in all the latest developments, you should definitely subscribe. The list is high volume, but generally has a fairly low noise level.

Subscribe by sending the message (in the body of your letter)

```
        subscribe perl5−porters
```

to perl5−porters−request@perl.org .

How are Perl Releases Numbered?

Perl version numbers are floating point numbers, such as 5.004. (Observations about the imprecision of floating point numbers for representing reality probably have more relevance than you might imagine :−) The major version number is 5 and the ‘004’ is the patchlevel. (Questions such as whether or not ‘004’ is really a minor version number can safely be ignored.:)

The version number is available as the magic variable `$],` and can be used in comparisons, e.g.

```
        print "You’ve got an old perl\n" if $] < 5.002;
```

You can also require particular version (or later) with

```
        use 5.002;
```

At some point in the future, we may need to decide what to call the next big revision. In the .package file used by metaconfig to generate Configure, there are two variables that might be relevant: `$baserev=5.0` and `$package=perl5.` At various times, I have suggested we might change them to `$baserev=5.1` and `$package=perl5.1` if want to signify a fairly major update. Or, we might want to jump to perl6. Let‘s worry about that problem when we get there.

Subversions

In addition, there may be "developer" sub−versions available. These are not official releases. They may contain unstable experimental features, and are subject to rapid change. Such developer sub−versions are numbered with sub−version numbers. For example, version 5.004_04 is the 4‘th developer version built on top of 5.004. It might include the _01, _02, and _03 changes, but it also might not. Sub−versions are allowed to be subversive.

These sub−versions can also be used as floating point numbers, so you can do things such as

```
        print "You’ve got an unstable perl\n" if $] == 5.00303;
```

You can also require particular version (or later) with

```
        use 5.003_03;    # the "_" is optional
```

Sub−versions produced by the members of perl5−porters are usually available on CPAN in the ***src/5.0/unsupported*** directory.

Maintenance and Development Subversions

As an experiment, starting with version 5.004, subversions _01 through _49 will be reserved for bug−fix maintenance releases, and subversions _50 through _99 will be available for unstable development versions.

The separate bug−fix track is being established to allow us an easy way to distribute important bug fixes without waiting for the developers to untangle all the other problems in the current developer‘s release.

Watch for announcements of maintenance subversions in comp.lang.perl.announce.

Why such a complicated scheme?

Two reasons, really. At least.

First, we need some way to identify and release collections of patches that are known to have new features that need testing and exploration. The subversion scheme does that nicely while fitting into the `use 5.004;` mold.

Second, since most of the folks who help maintain perl do so on a free−time voluntary basis, perl development does not proceed at a precise pace, though it always seems to be moving ahead quickly. We needed some way to pass around the "patch pumpkin" to allow different people chances to work on different aspects of the distribution without getting in each other‘s way. It wouldn‘t be constructive to have multiple people working on incompatible implementations of the same idea. Instead what was needed was some kind of "baton" or "token" to pass around so everyone knew whose turn was next.

Why is it called the patch pumpkin?

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.

Philosophical Issues in Patching Perl

There are no absolute rules, but there are some general guidelines I have tried to follow as I apply patches to the perl sources. (This section is still under construction.)

Solve problems as generally as possible

Never implement a specific restricted solution to a problem when you can solve the same problem in a more general, flexible way.

For example, for dynamic loading to work on some SVR4 systems, we had to build a shared libperl.so library. In order to build "FAT" binaries on NeXT 4.0 systems, we had to build a special libperl library. Rather than continuing to build a contorted nest of special cases, I generalized the process of building libperl so that NeXT and SVR4 users could still get their work done, but others could build a shared libperl if they wanted to as well.

Seek consensus on major changes

If you are making big changes, don‘t do it in secret. Discuss the ideas in advance on perl5−porters.

Keep the documentation up−to−date

If your changes may affect how users use perl, then check to be sure that the documentation is in sync with your changes. Be sure to check all the files ***pod/*.pod*** and also the ***INSTALL*** document.

Consider writing the appropriate documentation first and then implementing your change to correspond to the documentation.

Avoid machine−specific #ifdef‘s

To the extent reasonable, try to avoid machine−specific #ifdef‘s in the sources. Instead, use feature−specific #ifdef‘s. The reason is that the machine−specific #ifdef‘s may not be valid across major releases of the operating system. Further, the feature−specific tests may help out folks on another platform who have the same problem.

Allow for lots of testing

We should never release a main version without testing it as a subversion first.

Automate generation of derivative files

The ***embed.h***, ***keywords.h***, ***opcode.h***, and ***perltoc.pod*** files are all automatically generated by perl scripts. In general, don‘t patch these directly; patch the data files instead.

Configure and ***config_h.SH*** are also automatically generated by **metaconfig**. In general, you should patch the metaconfig units instead of patching these files directly. However, minor changes to ***Configure*** may be made in between major sync−ups with the metaconfig units, which tends to be complicated operations.

How to Make a Distribution

There really ought to be a ‘make dist’ target, but there isn‘t. The ‘dist’ suite of tools also contains a number of tools that I haven‘t learned how to use yet. Some of them may make this all a bit easier.

Here are the steps I go through to prepare a patch & distribution.

Lots of it could doubtless be automated but isn‘t.

Announce your intentions

First, you should volunteer out loud to take the patch pumpkin. It‘s generally counter−productive to have multiple people working in secret on the same thing.

At the same time, announce what you plan to do with the patch pumpkin, to allow folks a chance to object or suggest alternatives, or do it for you. Naturally, the patch pumpkin holder ought to incorporate various bug fixes and documentation improvements that are posted while he or she has the pumpkin, but there might also be larger issues at stake.

One of the precepts of the subversion idea is that we shouldn‘t give the patch pumpkin to anyone unless we have some idea what he or she is going to do with it.

refresh pod/perltoc.pod

Presumably, you have done a full `make` in your working source directory. Before you `make spotless` (if you do), and if you have changed any documentation in any module or pod file, change to the ***pod*** directory and run `make toc`.

update patchlevel.h

Don‘t be shy about using the subversion number, even for a relatively modest patch. We‘ve never even come close to using all 99 subversions, and it‘s better to have a distinctive number for your patch. If you need feedback on your patch, go ahead and issue it and promise to incorporate that feedback quickly (e.g. within 1 week) and send out a second patch.

run metaconfig

If you need to make changes to Configure or config_h.SH, it may be best to change the appropriate metaconfig units instead, and regenerate Configure.

```
        metaconfig −m
```

will regenerate Configure and config_h.SH. More information on obtaining and running metaconfig is in the ***U/README*** file that comes with Perl‘s metaconfig units. Perl‘s metaconfig units should be available the same place you found this file. On CPAN, look under my directory ***id/ANDYD/*** for a file such as ***5.003_07−02.U.tar.gz***. That file should be unpacked in your main perl source directory. It contains the files needed to run **metaconfig** to reproduce Perl‘s Configure script. (Those units are for 5.003_07. There have been changes since then; please contact me if you want more recent versions, and I will try to point you in the right direction.)

Alternatively, do consider if the ****ish.h*** files might be a better place for your changes.

MANIFEST

Make sure the MANIFEST is up−to−date. You can use dist‘s **manicheck** program for this. You can also use

```
    perl −MExtUtils::Manifest −e fullcheck
```

to do half the job. This will make sure everything listed in MANIFEST is included in the distribution. dist‘s **manicheck** command will also list extra files in the directory that are not listed in MANIFEST.

The MANIFEST is normally sorted, with one exception. Perl includes both a ***Configure*** script and a ***configure*** script. The ***configure*** script is a front−end to the main ***Configure***, but is there to aid folks who use autoconf−generated ***configure*** files for other software. The problem is that ***Configure*** and ***configure*** are the same on case−insensitive file systems, so I deliberately put ***configure*** first in the MANIFEST so that the extraction of ***Configure*** will overwrite ***configure*** and leave you with the correct script. (The ***configure*** script must also have write permission for this to work, so it‘s the only file in the distribution I normally have with write permission.)

If you are using metaconfig to regenerate Configure, then you should note that metaconfig actually uses MANIFEST.new, so you want to be sure MANIFEST.new is up−to−date too. I haven‘t found the MANIFEST/MANIFEST.new distinction particularly useful, but that‘s probably because I still haven‘t learned how to use the full suite of tools in the dist distribution.

Check permissions

All the tests in the t/ directory ought to be executable. The main makefile used to do a ‘chmod t/*/*.t‘, but that resulted in a self−modifying distribution—something some users would strongly prefer to avoid. Probably, the ***t/TEST*** script should check for this and do the chmod if needed, but it doesn‘t currently.

In all, the following files should probably be executable:

```
    Configure
    configpm
    configure
    embed.pl
    installperl
    installman
    keywords.pl
    lib/splain
    myconfig
    opcode.pl
    perly.fixer
    t/TEST
    t/*/*.t
    *.SH
    vms/ext/Stdio/test.pl
    vms/ext/filespec.t
    vms/fndvers.com
    x2p/*.SH
```

Other things ought to be readable, at least :−).

Probably, the permissions for the files could be encoded in MANIFEST somehow, but I‘m reluctant to change MANIFEST itself because that could break old scripts that use MANIFEST.

I seem to recall that some SVR3 systems kept some sort of file that listed permissions for system files; something like that might be appropriate.

Run Configure

This will build a config.sh and config.h. You can skip this if you haven‘t changed Configure or config_h.SH at all.

Update config_H

The config_H file is provided to help those folks who can‘t run Configure. It is important to keep it up−to−date. If you have changed config_h.SH, those changes must be reflected in config_H as well. (The name config_H was chosen to distinguish the file from config.h even on case−insensitive file systems.) Simply edit the existing config_H file; keep the first few explanatory lines and then copy your new config.h below.

It may also be necessary to update vms/config.vms and plan9/config.plan9, though you should be quite careful in doing so if you are not familiar with those systems. You might want to issue your patch with a promise to quickly issue a follow−up that handles those directories.

make run_byacc

If you have byacc−1.8.2 (available from CPAN), and if there have been changes to ***perly.y***, you can regenerate the ***perly.c*** file. The run_byacc makefile target does this by running byacc and then applying some patches so that byacc dynamically allocates space, rather than having fixed limits. This patch is handled by the ***perly.fixer*** script. Depending on the nature of the changes to ***perly.y***, you may or may not have to hand−edit the patch to apply correctly. If you do, you should include the edited patch in the new distribution. If you have byacc−1.9, the patch won‘t apply cleanly. Changes to the printf output statements mean the patch won‘t apply cleanly. Long ago I started to fix ***perly.fixer*** to detect this, but I never completed the task.

Some additional notes from Larry on this:

Don‘t forget to regenerate perly.c.diff.

```
    byacc −d perly.y
    mv y.tab.c perly.c
    patch perly.c <perly.c.diff
    # manually apply any failed hunks
    diff −c2 perly.c.orig perly.c >perly.c.diff
```

One chunk of lines that often fails begins with

```
    #line 29 "perly.y"
```

and ends one line before

```
    #define YYERRCODE 256
```

This only happens when you add or remove a token type. I suppose this could be automated, but it doesn‘t happen very often nowadays.

Larry

make regen_headers

The ***embed.h***, ***keywords.h***, and ***opcode.h*** files are all automatically generated by perl scripts. Since the user isn‘t guaranteed to have a working perl, we can‘t require the user to generate them. Hence you have to, if you‘re making a distribution.

I used to include rules like the following in the makefile:

```
# The following three header files are generated automatically
# The correct versions should be already supplied with the perl kit,
# in case you don't have perl or 'sh' available.
# The - is to ignore error return codes in case you have the source
# installed read-only or you don't have perl yet.
keywords.h: keywords.pl
        @echo "Don't worry if this fails."
        - perl keywords.pl
```

However, I got **lots** of mail consisting of people worrying because the command failed. I eventually decided that I would save myself time and effort by manually running `make regen_headers` myself rather than answering all the questions and complaints about the failing command.

global.sym and interp.sym

Make sure these files are up−to−date. Read the comments in these files and in perl_exp.SH to see what to do.

Binary compatibility

If you do change ***global.sym*** or ***interp.sym***, think carefully about what you are doing. To the extent reasonable, we‘d like to maintain souce and binary compatibility with older releases of perl. That way, extensions built under one version of perl will continue to work with new versions of perl.

Of course, some incompatible changes may well be necessary. I‘m just suggesting that we not make any such changes without thinking carefully about them first. If possible, we should provide backwards−compatibility stubs. There‘s a lot of XS code out there. Let‘s not force people to keep changing it.

Changes

Be sure to update the ***Changes*** file. Try to include both an overall summary as well as detailed descriptions of the changes. Your audience will include bother developers and users, so describe user−visible changes (if any) in terms they will understand, not in code like "initialize foo variable in bar function".

There are differing opinions on whether the detailed descriptions ought to go in the Changes file or whether they ought to be available separately in the patch file (or both). There is no disagreement that detailed descriptions ought to be easily available somewhere.

OS/2−specific updates

In the os2 directory is ***diff.configure***, a set of OS/2−specific diffs against **Configure**. If you make changes to Configure, you may want to consider regenerating this diff file to save trouble for the OS/2 maintainer.

You can also consider the OS/2 diffs as reminders of portability things that need to be fixed in Configure.

VMS−specific updates

If you have changed ***perly.y***, then you may want to update ***vms/perly_{h,c}.vms*** by running `perl vms/vms_yfix.pl`.

The Perl version number appears in several places under ***vms***. It is courteous to update these versions. For example, if you are making 5.004_42, replace "5.00441" with "5.00442".

Making the new distribution

Suppose, for example, that you want to make version 5.004_08. Then you can do something like the following

```
mkdir ../perl5.004_08
awk '{print $1}' MANIFEST | cpio -pdm ../perl5.004_08
cd ../
tar cf perl5.004_08.tar perl5.004_08
gzip --best perl5.004_08.tar
```

Making a new patch

I find the ***makepatch*** utility quite handy for making patches. You can obtain it from any CPAN archive under http://www.perl.com/CPAN/authors/Johan_Vromans/ . The only difference between my version and the standard one is that I have mine do a

```
# Print a reassuring "End of Patch" note so people won't
# wonder if their mailer truncated patches.
print "\n\nEnd of Patch.\n";
```

at the end. That's because I used to get questions from people asking if their mail was truncated.

Here's how I generate a new patch. I'll use the hypothetical 5.004_07 to 5.004_08 patch as an example.

```
# unpack perl5.004_07/
gzip -d -c perl5.004_07.tar.gz | tar -xof -
# unpack perl5.004_08/
gzip -d -c perl5.004_08.tar.gz | tar -xof -
makepatch perl5.004_07 perl5.004_08 > perl5.004_08.pat
```

Makepatch will automatically generate appropriate **rm** commands to remove deleted files. Unfortunately, it will not correctly set permissions for newly created files, so you may have to do so manually. For example, patch 5.003_04 created a new test ***t/op/gv.t*** which needs to be executable, so at the top of the patch, I inserted the following lines:

```
# Make a new test
touch t/op/gv.t
chmod +x t/opt/gv.t
```

Now, of course, my patch is now wrong because makepatch didn't know I was going to do that command, and it patched against /dev/null.

So, what I do is sort out all such shell commands that need to be in the patch (including possible mv−ing of files, if needed) and put that in the shell commands at the top of the patch. Next, I delete all the patch parts of perl5.004_08.pat, leaving just the shell commands. Then, I do the following:

```
cd perl5.004_07
sh ../perl5.004_08.pat
cd ..
makepatch perl5.004_07 perl5.004_08 >> perl5.004_08.pat
```

(Note the append to preserve my shell commands.) Now, my patch will line up with what the end users are going to do.

Testing your patch

It seems obvious, but be sure to test your patch. That is, verify that it produces exactly the same thing as your full distribution.

```
rm -rf perl5.004_07
gzip -d -c perl5.004_07.tar.gz | tar -xf -
cd perl5.004_07
sh ../perl5.004_08.pat
patch -p1 -N < ../perl5.004_08.pat
cd ..
gdiff -r perl5.004_07 perl5.004_08
```

where **gdiff** is GNU diff. Other diff‘s may also do recursive checking.

More testing

Again, it‘s obvious, but you should test your new version as widely as you can. You can be sure you‘ll hear about it quickly if your version doesn‘t work on both ANSI and pre−ANSI compilers, and on common systems such as SunOS 4.1.[34], Solaris, and Linux.

If your changes include conditional code, try to test the different branches as thoroughly as you can. For example, if your system supports dynamic loading, you can also test static loading with

```
sh Configure −Uusedl
```

You can also hand−tweak your config.h to try out different #ifdef branches.

Common Gotcha‘s

#elif The ‘#elif’ preprocessor directive is not understood on all systems. Specifically, I know that Pyramids don‘t understand it. Thus instead of the simple

```
#if defined(I_FOO)
#  include <foo.h>
#elif defined(I_BAR)
#  include <bar.h>
#else
#  include <fubar.h>
#endif
```

You have to do the more Byzantine

```
#if defined(I_FOO)
#  include <foo.h>
#else
#  if defined(I_BAR)
#    include <bar.h>
#  else
#    include <fubar.h>
#  endif
#endif
```

Incidentally, whitespace between the leading ‘#’ and the preprocessor command is not guaranteed, but is very portable and you may use it freely. I think it makes things a bit more readable, especially once things get rather deeply nested. I also think that things should almost never get too deeply nested, so it ought to be a moot point :−)

Probably Prefer POSIX

It‘s often the case that you‘ll need to choose whether to do something the BSD−ish way or the POSIX−ish way. It‘s usually not a big problem when the two systems use different names for similar functions, such as `memcmp()` and `bcmp()`. The perl.h header file handles these by appropriate #defines, selecting the POSIX `mem*()` functions if available, but falling back on the `b*()` functions, if need be.

More serious is the case where some brilliant person decided to use the same function name but give it a different meaning or calling sequence :−). `getpgrp()` and `setpgrp()` come to mind. These are a real problem on systems that aim for conformance to one standard (e.g. POSIX), but still try to support the other way of doing things (e.g. BSD). My general advice (still not really implemented in the source) is to do something like the following. Suppose there are two alternative versions, `fooPOSIX()` and `fooBSD()`.

```
#ifdef HAS_FOOPOSIX
    /* use fooPOSIX(); */
#else
```

```
#  ifdef HAS_FOOBSD
    /* try to emulate fooPOSIX() with fooBSD();
       perhaps with the following:  */
#    define fooPOSIX fooBSD
#  else
#  /* Uh, oh.  We have to supply our own. */
#    define fooPOSIX Perl_fooPOSIX
#  endif
#endif
```

Think positively

If you need to add an #ifdef test, it is usually easier to follow if you think positively, e.g.

```
#ifdef HAS_NEATO_FEATURE
    /* use neato feature */
#else
    /* use some fallback mechanism */
#endif
```

rather than the more impenetrable

```
#ifndef MISSING_NEATO_FEATURE
    /* Not missing it, so we must have it, so use it */
#else
    /* Are missing it, so fall back on something else. */
#endif
```

Of course for this toy example, there‘s not much difference. But when the #ifdef‘s start spanning a couple of screen fulls, and the #else‘s are marked something like

```
#else /* !MISSING_NEATO_FEATURE */
```

I find it easy to get lost.

Providing Missing Functions — Problem

Not all systems have all the neat functions you might want or need, so you might decide to be helpful and provide an emulation. This is sound in theory and very kind of you, but please be careful about what you name the function. Let me use the `pause()` function as an illustration.

Perl5.003 has the following in ***perl.h***

```
#ifndef HAS_PAUSE
#define pause() sleep((32767<<16)+32767)
#endif
```

Configure sets HAS_PAUSE if the system has the `pause()` function, so this #define only kicks in if the `pause()` function is missing. Nice idea, right?

Unfortunately, some systems apparently have a prototype for `pause()` in ***unistd.h***, but don‘t actually have the function in the library. (Or maybe they do have it in a library we‘re not using.)

Thus, the compiler sees something like

```
extern int pause(void);
/* . . . */
#define pause() sleep((32767<<16)+32767)
```

and dies with an error message. (Some compilers don‘t mind this; others apparently do.)

To work around this, 5.003_03 and later have the following in perl.h:

```
/* Some unistd.h’s give a prototype for pause() even though
   HAS_PAUSE ends up undefined.  This causes the #define
```

```
       below to be rejected by the compiler.  Sigh.
    */
    #ifdef HAS_PAUSE
    #  define Pause pause
    #else
    #  define Pause() sleep((32767<<16)+32767)
    #endif
```

This works.

The curious reader may wonder why I didn‘t do the following in ***util.c*** instead:

```
    #ifndef HAS_PAUSE
    void pause()
    {
    sleep((32767<<16)+32767);
    }
    #endif
```

That is, since the function is missing, just provide it. Then things would probably be been alright, it would seem.

Well, almost. It could be made to work. The problem arises from the conflicting needs of dynamic loading and namespace protection.

For dynamic loading to work on AIX (and VMS) we need to provide a list of symbols to be exported. This is done by the script ***perl_exp.SH***, which reads ***global.sym*** and ***interp.sym***. Thus, the `pause` symbol would have to be added to ***global.sym*** So far, so good.

On the other hand, one of the goals of Perl5 is to make it easy to either extend or embed perl and link it with other libraries. This means we have to be careful to keep the visible namespace "clean". That is, we don‘t want perl‘s global variables to conflict with those in the other application library. Although this work is still in progress, the way it is currently done is via the ***embed.h*** file. This file is built from the ***global.sym*** and ***interp.sym*** files, since those files already list the globally visible symbols. If we had added `pause` to global.sym, then ***embed.h*** would contain the line

```
    #define pause       Perl_pause
```

and calls to `pause` in the perl sources would now point to `Perl_pause`. Now, when **ld** is run to build the ***perl*** executable, it will go looking for `perl_pause`, which probably won‘t exist in any of the standard libraries. Thus the build of perl will fail.

Those systems where `HAS_PAUSE` is not defined would be ok, however, since they would get a `Perl_pause` function in util.c. The rest of the world would be in trouble.

And yes, this scenario has happened. On SCO, the function `chsize` is available. (I think it‘s in ***−lx***, the Xenix compatibility library.) Since the perl4 days (and possibly before), Perl has included a `chsize` function that gets called something akin to

```
    #ifndef HAS_CHSIZE
    I32 chsize(fd, length)
    /*  . . . */
    #endif
```

When 5.003 added

```
    #define chsize      Perl_chsize
```

to ***embed.h***, the compile started failing on SCO systems.

The "fix" is to give the function a different name. The one implemented in 5.003_05 isn‘t optimal, but here‘s what was done:

```
#ifdef HAS_CHSIZE
# ifdef my_chsize  /* Probably #defined to Perl_my_chsize in embed.h */
#   undef my_chsize
# endif
# define my_chsize chsize
#endif
```

My explanatory comment in patch 5.003_05 said:

```
Undef and then re-define my_chsize from Perl_my_chsize to
just plain chsize if this system HAS_CHSIZE.  This probably only
applies to SCO.  This shows the perils of having internal
functions with the same name as external library functions :-).
```

Now, we can safely put `my_chsize` in ***global.sym***, export it, and hide it with ***embed.h***.

To be consistent with what I did for `pause`, I probably should have called the new function `Chsize`, rather than `my_chsize`. However, the perl sources are quite inconsistent on this (Consider New, Mymalloc, and Myremalloc, to name just a few.)

There is a problem with this fix, however, in that `Perl_chsize` was available as a ***libperl.a*** library function in 5.003, but it isn‘t available any more (as of 5.003_07). This means that we‘ve broken binary compatibility. This is not good.

Providing missing functions — some ideas

We currently don‘t have a standard way of handling such missing function names. Right now, I‘m effectively thinking aloud about a solution. Some day, I‘ll try to formally propose a solution.

Part of the problem is that we want to have some functions listed as exported but not have their names mangled by embed.h or possibly conflict with names in standard system headers. We actually already have such a list at the end of ***perl_exp.SH*** (though that list is out−of−date):

```
# extra globals not included above.
cat <<END >> perl.exp
perl_init_ext
perl_init_fold
perl_init_i18nl14n
perl_alloc
perl_construct
perl_destruct
perl_free
perl_parse
perl_run
perl_get_sv
perl_get_av
perl_get_hv
perl_get_cv
perl_call_argv
perl_call_pv
perl_call_method
perl_call_sv
perl_requirepv
safecalloc
safemalloc
saferealloc
safefree
```

This still needs much thought, but I‘m inclined to think that one possible solution is to prefix all such functions with `perl_` in the source and list them along with the other `perl_*` functions in

perl_exp.SH.

Thus, for `chsize`, we‘d do something like the following:

```
/* in perl.h */
#ifdef HAS_CHSIZE
#  define perl_chsize chsize
#endif
```

then in some file (e.g. ***util.c*** or ***doio.c***) do

```
#ifndef HAS_CHSIZE
I32 perl_chsize(fd, length)
/* implement the function here . . . */
#endif
```

Alternatively, we could just always use `chsize` everywhere and move `chsize` from ***global.sym*** to the end of ***perl_exp.SH***. That would probably be fine as long as our `chsize` function agreed with all the `chsize` function prototypes in the various systems we‘ll be using. As long as the prototypes in actual use don‘t vary that much, this is probably a good alternative. (As a counter−example, note how Configure and perl have to go through hoops to find and use get Malloc_t and Free_t for `malloc` and `free`.)

At the moment, this latter option is what I tend to prefer.

All the world‘s a VAX

Sorry, showing my age:−). Still, all the world is not BSD 4.[34], SVR4, or POSIX. Be aware that SVR3−derived systems are still quite common (do you have any idea how many systems run SCO?) If you don‘t have a bunch of v7 manuals handy, the metaconfig units (by default installed in ***/usr/local/lib/dist/U***) are a good resource to look at for portability.

Miscellaneous Topics

Autoconf

Why does perl use a metaconfig−generated Configure script instead of an autoconf−generated configure script?

Metaconfig and autoconf are two tools with very similar purposes. Metaconfig is actually the older of the two, and was originally written by Larry Wall, while autoconf is probably now used in a wider variety of packages. The autoconf info file discusses the history of autoconf and how it came to be. The curious reader is referred there for further information.

Overall, both tools are quite good, I think, and the choice of which one to use could be argued either way. In March, 1994, when I was just starting to work on Configure support for Perl5, I considered both autoconf and metaconfig, and eventually decided to use metaconfig for the following reasons:

Compatibility with Perl4

Perl4 used metaconfig, so many of the #ifdef‘s were already set up for metaconfig. Of course metaconfig had evolved some since Perl4‘s days, but not so much that it posed any serious problems.

Metaconfig worked for me

My system at the time was Interactive 2.2, a SVR3.2/386 derivative that also had some POSIX support. Metaconfig−generated Configure scripts worked fine for me on that system. On the other hand, autoconf−generated scripts usually didn‘t. (They did come quite close, though, in some cases.) At the time, I actually fetched a large number of GNU packages and checked. Not a single one configured and compiled correctly out−of−the−box with the system‘s cc compiler.

Configure can be interactive

With both autoconf and metaconfig, if the script works, everything is fine. However, one of my main problems with autoconf−generated scripts was that if it guessed wrong about something, it could be **very** hard to go back and fix it. For example, autoconf always insisted on passing the −Xp flag to cc (to turn on POSIX behavior), even when that wasn‘t what I wanted or needed for that package. There was no way short of editing the configure script to turn this off. You couldn‘t just edit the resulting Makefile at the end because the −Xp flag influenced a number of other configure tests.

Metaconfig‘s Configure scripts, on the other hand, can be interactive. Thus if Configure is guessing things incorrectly, you can go back and fix them. This isn‘t as important now as it was when we were actively developing Configure support for new features such as dynamic loading, but it‘s still useful occasionally.

GPL

At the time, autoconf−generated scripts were covered under the GNU Public License, and hence weren‘t suitable for inclusion with Perl, which has a different licensing policy. (Autoconf‘s licensing has since changed.)

Modularity

Metaconfig builds up Configure from a collection of discrete pieces called "units". You can override the standard behavior by supplying your own unit. With autoconf, you have to patch the standard files instead. I find the metaconfig "unit" method easier to work with. Others may find metaconfig‘s units clumsy to work with.

@INC search order

By default, the list of perl library directories in @INC is the following:

```
$archlib
$privlib
$sitearch
$sitelib
```

Specifically, on my Solaris/x86 system, I run **sh Configure −Dprefix=/opt/perl** and I have the following directories:

```
/opt/perl/lib/i86pc−solaris/5.00307
/opt/perl/lib
/opt/perl/lib/site_perl/i86pc−solaris
/opt/perl/lib/site_perl
```

That is, perl‘s directories come first, followed by the site−specific directories.

The site libraries come second to support the usage of extensions across perl versions. Read the relevant section in ***INSTALL*** for more information. If we ever make `$sitearch` version−specific, this topic could be revisited.

Why isn‘t there a directory to override Perl‘s library?

Mainly because no one‘s gotten around to making one. Note that "making one" involves changing perl.c, Configure, config_h.SH (and associated files, see above), and *documenting* it all in the INSTALL file.

Apparently, most folks who want to override one of the standard library files simply do it by overwriting the standard library files.

APPLLIB

In the perl.c sources, you‘ll find an undocumented APPLLIB_EXP variable, sort of like PRIVLIB_EXP and ARCHLIB_EXP (which are documented in config_h.SH). Here‘s what APPLLIB_EXP is for, from a mail message from Larry:

```
The main intent of APPLLIB_EXP is for folks who want to send out a
version of Perl embedded in their product.  They would set the symbol
to be the name of the library containing the files needed to run or to
support their particular application.  This works at the "override"
level to make sure they get their own versions of any library code that
they absolutely must have configuration control over.
```

```
    As such, I don't see any conflict with a sysadmin using it for a
    override-ish sort of thing, when installing a generic Perl.  It should
    probably have been named something to do with overriding though.  Since
    it's undocumented we could still change it...  :-)
```

Given that it's already there, you can use it to override distribution modules. If you do

```
    sh Configure -Dccflags='-DAPPLLIB_EXP=/my/override'
```

then perl.c will put /my/override ahead of ARCHLIB and PRIVLIB.

Upload Your Work to CPAN

You can upload your work to CPAN if you have a CPAN id. Check out http://www.perl.com/CPAN/modules/04pause.html for information on _PAUSE_, the Perl Author's Upload Server.

I typically upload both the patch file, e.g. ***perl5.004_08.pat.gz*** and the full tar file, e.g. ***perl5.004_08.tar.gz***.

If you want your patch to appear in the ***src/5.0/unsupported*** directory on CPAN, send e−mail to the CPAN master librarian. (Check out http://www.perl.com/CPAN/CPAN.html).

Help Save the World

You should definitely announce your patch on the perl5−porters list. You should also consider announcing your patch on comp.lang.perl.announce, though you should make it quite clear that a subversion is not a production release, and be prepared to deal with people who will not read your disclaimer.

Todo

Here, in no particular order, are some Configure and build−related items that merit consideration. This list isn't exhaustive, it's just what I came up with off the top of my head.

Good ideas waiting for round tuits

installprefix

I think we ought to support

```
    Configure -Dinstallprefix=/blah/blah
```

Currently, we support **−Dprefix=/blah/blah**, but the changing the install location has to be handled by something like the ***config.over*** trick described in ***INSTALL***. AFS users also are treated specially. We should probably duplicate the metaconfig prefix stuff for an install prefix.

Configure −Dsrcdir=/blah/blah

We should be able to emulate **configure —srcdir**. Tom Tromey tromey@creche.cygnus.com has submitted some patches to the dist−users mailing list along these lines. Eventually, they ought to get folded back into the main distribution.

Hint file fixes

Various hint files work around Configure problems. We ought to fix Configure so that most of them aren't needed.

Hint file information

Some of the hint file information (particularly dynamic loading stuff) ought to be fed back into the main metaconfig distribution.

Probably good ideas waiting for round tuits

GNU configure —options

I've received sensible suggestions for —exec_prefix and other GNU configure —options. It's not always obvious exactly what is intended, but this merits investigation.

make clean

Currently, **make clean** isn‘t all that useful, though **make realclean** and **make distclean** are. This needs a bit of thought and documentation before it gets cleaned up.

Try gcc if cc fails

Currently, we just give up.

bypassing safe*alloc wrappers

On some systems, it may be safe to call the system malloc directly without going through the util.c safe* layers. (Such systems would accept free(0), for example.) This might be a time−saver for systems that already have a good malloc. (Recent Linux libc‘s apparently have a nice malloc that is well−tuned for the system.)

Vague possibilities

Win95, WinNT, and Win32 support

We need to get something into the distribution for 32−bit Windows. I‘m tired of all the private e−mail questions I get, and I‘m saddened that so many folks keep trying to reinvent the same wheel.

MacPerl

Get some of the Macintosh stuff folded back into the main distribution.

gconvert replacement

Maybe include a replacement function that doesn‘t lose data in rare cases of coercion between string and numerical values.

long long

Can we support `long long` on systems where `long long` is larger than what we‘ve been using for `IV`? What if you can‘t `sprintf` a `long long`?

Improve makedepend

The current makedepend process is clunky and annoyingly slow, but it works for most folks. Alas, it assumes that there is a filename `$firstmakefile` that the **make** command will try to use before it uses ***Makefile***. Such may not be the case for all **make** commands, particularly those on non−Unix systems.

Probably some variant of the BSD ***.depend*** file will be useful. We ought to check how other packages do this, if they do it at all. We could probably pre−generate the dependencies (with the exception of malloc.o, which could probably be determined at ***Makefile.SH*** extraction time.

GNU Makefile standard targets

GNU software generally has standardized Makefile targets. Unless we have good reason to do otherwise, I see no reason not to support them.

File locking

Somehow, straighten out, document, and implement `lockf()`, `flock()`, and/or `fcntl()` file locking. It‘s a mess.

AUTHOR

Andy Dougherty <doughera@lafcol.lafayette.edu.

Additions by Chip Salzenberg <chip@atlantic.net.

All opinions expressed herein are those of the author(s).

LAST MODIFIED

`$Id:` pumpkin.pod,v 1.9 1997/02/24 20:37:43 doughera Released `$`

NAME

perlplan9 − Plan 9−specific documentation for Perl

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.

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.

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.

What‘s not in Plan 9 Perl

As mentioned previously, dynamic loading isn‘t currently available nor is MakeMaker. Both are high−priority items.

Perl5 Functions not currently supported

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.

Signals

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
```

BUGS

"As many as there are grains of sand on all the beaches of the world . . ." − Carl Sagan

Revision date

This document was revised 09−October−1996 for Perl 5.003_7.

AUTHOR

Luther Huffman, lutherh@stratcom.com

NAME

perlvms − VMS−specific documentation for Perl

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@genetics.upenn.edu.

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..

Organization of Perl Images

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.

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 *perlapi* 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.

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).

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, you shouldn't nest the source directory too deeply in your directory structure, lest you eccedd 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 ***[.lib.site_perl.auto****Arch*.*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`$PVers`Extname****]*** subdirectory
of one of the directories in `@INC` (where *PVers*
is the version of Perl you're using, as supplied in `$]`,
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.

File specifications

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 fle 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.

Wildcard expansion

File specifications containing wildcards are allowed both on the command line and within Perl globs (e.g. <C<*.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.

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.

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 passed directly to lib`$spawn` as the command to execute. In this case, Perl will wait for the subprocess to complete before continuing.

PERL5LIB and PERLLIB

The PERL5LIB and PERLLIB logical names work as documented *perl*, except that the element separator is '|' instead of ':'. The directory specifications may use either VMS or Unix syntax.

Command line

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:

```
<F<file> reads stdin from F<file>,
>F<file> writes stdout to F<file>,
>>F<file> appends stdout to F<file>,
2>F<file> writes stderr to F<file>, and
2>>F<file> appends stderr to F<file>.
```

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.

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` 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.

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, fork*, getc, getlogin,
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, fcntl, flock,
getpgrp, getppid, 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, truncate
```

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

The `exec` operator behaves in one of two different ways. If called after a call to `fork`, it will invoke the CRTL `execv()` routine, passing its arguments to the subprocess created by `fork` for execution. In this case, it is subject to all limitations that affect `execv()`. (In particular, this usually means that the command executed in the subprocess must be an image compiled from C source code, and that your options for passing file descriptors and signal handlers to the subprocess are limited.)

If the call to `exec` does not follow a call to `fork`, it will cause Perl to exit, and to invoke the command given as an argument to `exec` via `lib$do_command.` If the argument begins with a '`$`' (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.

You can use `exec` in both ways within the same script, as long as you call `fork` and `exec` in pairs. Perl keeps track of how many times `fork` and `exec` have been called, and will call the CRTL `execv()` routine if there have previously been more calls to `fork` than to `exec`.

fork The `fork` operator works in the same way as the CRTL `vfork()` routine, which is quite different under VMS than under Unix. Specifically, while `fork` returns 0 after it is called and the subprocess PID after `exec` is called, in both cases the thread of execution is within the parent process, so there is no opportunity to perform operations in the subprocess before calling `exec`.

In general, the use of `fork` and `exec` to create subprocess is not recommended under VMS; wherever possible, use the `system` operator or piped filehandles instead.

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` 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 `$DELPRC` system service is is called directly. This insures that the target process is actually deleted, if at all possible. (The CRTL's `kill()` function is presently implemented via `$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$spawn()`, any valid DCL command string may be specified. If LIST consists of the empty string, `system` spawns an interactive DCL subprocess, in the same fashion as typiing **SPAWN** at the DCL prompt. Perl waits for the subprocess to complete before continuing execution in the current process. As described in *perlfunc*, the return value of `system` is a fake "status" which follows POSIX semantics; see the description of `$?` in this document for more detail. The actual VMS exit status of the subprocess is available in `$^S` (as long as you haven‘t used another Perl function that resets `$?` and `$^S` in the meantime).

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 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 suprocesses 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*, `waitpid` will wait for that subprocess, and return its final status value. If PID is a subprocess created in some other way (e.g. SPAWNed before Perl was invoked), or is not a subprocess of the current process, `waitpid` will check once per second whether the process has completed, and when it has, will return 0. (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.)

The FLAGS argument is ignored in all cases.

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 infrmation takes precedence.

%ENV

Reading the elements of the %ENV array returns the translation of the logical name specified by the key, according to the normal search order of access modes and logical name tables. If you append a semicolon to the logical name, followed by an integer, that integer is used as the translation index for the logical name, so that you can 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`.

The %ENV keys `home`, `path`,`term`, and `user` return the CRTL "environment variables" of the same names, if these logical names are not defined. The key `default` returns the current default device and directory specification, regardless of whether there is a logical name DEFAULT defined..

Setting an element of %ENV defines a supervisor−mode logical name in the process logical name table. `Undef`ing or `delete`ing an element of %ENV deletes the equivalent user− mode or supervisor−mode logical name from the process logical name table. If you use `undef`, the %ENV element remains empty. If you use `delete`, another attempt is made at logical name translation after the deletion, so an inner−mode logical name or a name in another logical name table will replace the logical name just deleted. It is not possible at present to define a search list logical name via %ENV.

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. The `each` operator is special: it returns each element *already* in %ENV, but doesn‘t go out and look for more. Therefore, if you‘ve previously used `keys` or `values`, you‘ll see all the logical names visible to your process, and if not, you‘ll see only the names you‘ve looked up so far. (This is a consequence of the way `each` is implemented now, and it may change in the future, so it wouldn‘t be a good idea to rely on it too much.)

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.

`$!` The string value of `$!` is that returned by the CRTL's `strerror()` function, so it will include the VMS message for VMS−specific errors. The numeric value of `$!` is the value of `errno`, except if errno is EVMSERR, in which case `$!` contains the value of vaxc`$errno.` Setting `$!` always sets errno to the value specified. If this value is EVMSERR, it also sets vaxc`$errno` to 4 (NONAME−F−NOMSG), so that the string value of `$!` won't reflect the VMS error message from before `$!` was set.

`$^E` This variable provides direct access to VMS status values in vaxc`$errno,` which are often more specific than the generic Unix−style error messages in `$!.` Its numeric value is the value of vaxc`$errno,` and its string value is the corresponding VMS message string, as retrieved by `sys$getmsg()`. Setting `$^E` sets vaxc`$errno` to the value specified.

`$?` The "status value" returned in `$?` 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 `$?` are always 0 under VMS, since the termination status of a process may or may not have been generated by an exception. The next 8 bits are derived from severity portion of the subprocess' exit status: if the severity was success or informational, these bits are all 0; otherwise, they contain the severity value shifted left one bit. As a result, `$?` will always be zero if the subprocess' exit status indicated successful completion, and non−zero if a warning or error occurred. The actual VMS exit status may be found in `$^S` (q.v.).

`$^S` Under VMS, this is the 32−bit VMS status value returned by the last subprocess to complete. Unlink `$?,` no manipulation is done to make this look like a POSIX wait(5) value, so it may be treated as a normal VMS status value.

`$|` Setting `$|` 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.

Revision date

This document was last updated on 28−Feb−1996, for Perl 5, patchlevel 2.

AUTHOR

Charles Bailey bailey@genetics.upenn.edu