NAME

perl – Practical Extraction and Report Language

SYNOPSIS

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

```
perl        Perl overview (this section)
perltoc     Perl documentation table of contents
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
perlref     Perl references
perldsc     Perl data structures intro
perllol     Perl data structures: lists of lists
perlobj     Perl objects
perltie     Perl objects hidden behind simple variables
perlbot     Perl OO tricks and examples
perldebug   Perl debugging
perldiag    Perl diagnostic messages
perlform    Perl formats
perlipc     Perl interprocess communication
perlsec     Perl security
perltrap    Perl traps for the unwary
perlstyle   Perl style guide
perlxs      Perl XS application programming interface
perlxstut   Perl XS tutorial
perlguts    Perl internal functions for those doing extensions
perlcall    Perl calling conventions from C
perlembed   Perl how to embed perl in your C or C++ app
perlpod     Perl plain old documentation
perlbook    Perl book information
```

(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 -le 'use Config; print "@Config{man1dir,man3dir}"'
```

If the directories were */usr/local/man/man1* and */usr/local/man/man3*, you would only need to add */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 an interpreted 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). It 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 hash tables used by 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 associative arrays (where dbm is available). 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".

• 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.

• Embeddible 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.

- 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 just for 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.

Ok, that's *definitely* enough hype.

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";
```

PERL5DB The command used to get the debugger code. If unset, uses

```
BEGIN { require 'perl5db.pl' }
```

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.

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'};
```

AUTHOR

Larry Wall <<*lwall@netlabs.com*>, with the help of oodles of other folks.

FILES

```
"/tmp/perl-e$$"        temporary file for -e commands
"@INC"                 locations of perl 5 libraries
```

SEE ALSO

```
a2p    awk to perl translator

s2p    sed to perl translator
```

DIAGNOSTICS

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

See the *perldiag* manpage 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 the *perlsec* manpage.

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 coredump 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 identifier 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.

See the perl bugs database at *http://perl.com/perl/bugs/*. You may mail your bug reports (be sure to include full configuration information as output by the myconfig program in the perl source tree) to *perl-bug@perl.com*.

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

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.

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.

Since 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 the *perlref* manpage.

Names that start with a digit may only contain 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 the `wantarray` entry in the *perlfunc* manpage.

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 the pretty much same thing for nearly all purposes, but 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 the 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 the *perlre* manpage 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 `$#days`, as in **csh**. (Actually, it's not the length of the array, it's the subscript of the last element, since there is (ordinarily) a 0th element.) Assigning to `$#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 make 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 just assume that

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

Some programmer choose to use an explcit 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 only useful 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 the `qq` entry in the *perlop* manpage for a list.

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, identifiers 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 identifier 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, since single quote is a valid (though deprecated) character in an identifier (see the `Packages` entry in the *perlmod* manpage).

Two special literals are __LINE__ and __FILE__, which represent the current line number and filename at that point in your program. They may only be used as separate tokens; they will not be interpolated into strings. In addition, the token __END__ 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 the DATA filehandle. (The DATA filehandle may read data only from the main script, but not from any required file or evaluated string.) The two control characters ^D and ^Z are synonyms for __END__ (or __DATA__ in a module; see the *SelfLoader* manpage for details on __DATA__).

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— see the Merry Christmas example below.) The terminating string must appear by itself (unquoted and with no surrounding whitespace) on the terminating line.

```
    print <<EOF;    # same as above
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 the *perlref* manpage.

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 PARENS

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

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

Lists may be assigned to 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, since 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 quotes its left-hand operand, which 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 the `sort` entry in the *perlfunc* manpage 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.

One place where you still use typeglobs (or references thereto) is for passing or storing filehandles. If you want to save away a filehandle, do it this way:

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

or perhaps as a real reference, like this:

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

This is also the 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');
```

See the *perlref* manpage, the *perlsub* manpage, and the section on *Symbols Tables* in the *perlmod* manpage for more discussion on typeglobs. See the `open` entry in the *perlfunc* manpage for other ways of generating filehandles.

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 to be able to access those private variables.

Declaring a subroutine allows a subroutine name to be used as if it were a list operator from that point forward in the program. You can declare a subroutine without defining it by saying 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 the *perlmod* manpage 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, since 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 the `do` entry in the *perlfunc* manpage. Note also that the loop control statements described later will *NOT* work in this construct, since 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. Since 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 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.

In either the `if` or the `while` statement, you may replace “(EXPR)” with a BLOCK, and the conditional is true if the value of the last statement in that block is true. While this “feature” continues to work in version 5, it has been deprecated, so please change any occurrences of “if BLOCK” to “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++;
}
```

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. The variable is implicitly local to the loop and regains its former value upon exiting the loop. If the variable was previously declared with `my`, it uses that variable instead of the global one, but it’s still localized to the loop. This can cause problems if you have subroutine or format declarations within that block’s scope.

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 $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 ($i = 0; $i < @ary1; $i++) {
    for ($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 confortable with the idiom might do it:

```
OUTER: foreach $wid (@ary1) {
INNER:   foreach $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, you won't accidentally excecute it because you've explicitly asked to iterate 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. 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) ? "w" : "a" }
    elsif  ($flag & O_RDWR)   {
        if ($flag & O_CREAT)  { "w+" }
        else                  { ($flag & O_APPEND) ? "r+" : "a+" }
    }
};
```

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 cases like this, it's usually a far, far better idea to use the structured control flow mechanisms of `next`, `last`, or `redo` insetad 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. If while expecting the beginning of a new statement, 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 the *perlpod* manpage.

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 only look at paragraphs beginning with a pod diretive (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.

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.)

```
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)
left        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 quotelike 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 the *perlfunc* manpage.

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 parens:

```
# 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 the section on *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 the section on *Quote and Quotelike Operators* toward the end of this section, as well as the section on *I/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 the *perlref* manpage.

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 the *perlobj* manpage.

Autoincrement and Autodecrement

"++" 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 autoincrement 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 only been used in 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 autodecrement 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.

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 the section on *List Operators*.)

Unary "\" creates a reference to whatever follows it. See the *perlref* manpage. 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 an 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 is less efficient than an explicit search, since the pattern must be compiled every time the expression is evaluated— unless you've used `/o`.)

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 parens, 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.

Binary “>>” returns the value of its left argument shifted right by the number of bits specified by the right argument. Arguments should be integers.

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 the *perlfunc* manpage.

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 the section on *List Operators*.

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.

Bitwise And

Binary "&" returns its operators ANDed together bit by bit.

Bitwise Or and Exclusive Or

Binary "|" returns its operators ORed together bit by bit.

Binary "^" returns its operators XORed together bit by bit.

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 autoincrement algorithm if the operaands 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 the section on *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 Quotelike 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
\n          newline
\r          return
\f          form feed
\v          vertical tab, whatever that is
\b          backspace
\a          alarm (bell)
\e          escape
\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
```

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, backquotes do *NOT* interpolate within double quotes, nor do single quotes impede evaluation of variables when used within double quotes.

Regexp Quotelike Operators

Here are the quotelike 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 only want to see 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 the *perlre* manpage.

Options are:

```
g   Match globally, i.e. find all occurrences.
i   Do case-insensitive pattern matching.
m   Treat string as multiple lines.
o   Only compile pattern 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 using the *pos()* function—see the *perlfunc* manpage.) 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;  # $* deprecated in Perl 5
while ($paragraph = <>) {
    while ($paragraph =~ /[a-z]['")]*[.!?]+['")]*\s/g) {
        $sentences++;
    }
}
print "$sentences\n";
```

q/STRING/

'STRING'

A single-quoted, literal string. Backslashes are ignored, 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.');
```

qq/STRING/

"STRING"

A double-quoted, interpolated string.

```
$_ .= qq
 (*** The previous line contains the naughty word "$1".\n)
          if /(tcl|rexx|python)/;      # :-)
```

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 the section on *I/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 (0).

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 only want the pattern compiled 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 the *perlre* manpage for further explanation on these.

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   Only compile pattern 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). If backquotes are used, the replacement string is a command to execute whose output will be used as the actual replacement text. 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 only use the \<*digit*> form in the left hand side. Anywhere else it's $<*digit*>.

Occasionally, you can't just use 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 backticks (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 the *perlvar* manpage 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 backticks is `qx//`. (Because backticks always undergo shell expansion as well, see the *perlsec* manpage for security concerns.)

Evaluating a filehandle in angle brackets yields the next line from that file (newline included, so it's never false until end of file, at which time an undefined value is returned). 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` 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 the `open()` entry in the *perlfunc* manpage 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 version of Perl, programmers would insert curly brackets to force interpretation as a filename glob:

<${foo}>. These days, it's consdired cleaner to call the internal function directly as glob($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 just 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 only evaluates its (embedded) argument 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 notatin.

```
@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.

NAME

perlre – Perl regular expressions

DESCRIPTION

This page describes the syntax of regular expressions in Perl. For a description of how to actually *use* regular expressions in matching operations, plus various examples of the same, see `m//` and `s///` in the *perlop* manpage.

The matching operations can have various modifiers, some of which relate to the interpretation of the regular expression inside. These are:

```
i   Do case-insensitive pattern matching.
m   Treat string as multiple lines.
s   Treat string as single line.
x   Extend your pattern's legibilty with 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 not backslashed or within a character class. You can use this to break up your regular expression into (slightly) more readable parts. The `#` character is also treated as a metacharacter introducing a comment, just as in ordinary Perl code. Taken together, these features go a long way towards making Perl 5 a readable language. See the C comment deletion code in the *perlop* manpage.

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 the section on *Version 8 Regular Expressions* for details.

In particular the following metacharacters have their standard *egrep*–ish meanings:

```
\   Quote the next metacharacter
^   Match the beginning of the line
.   Match any character (except newline)
$   Match the end of the line
|   Alternation
()  Grouping
[]  Character class
```

By default, the "^" character is guaranteed to match only at the beginning of the string, the "$" character only at the end (or before the newline at the end) and Perl does certain optimizations with the assumption that the string contains only one line. Embedded newlines will not be matched by "^" or "$". 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 deprecated in Perl 5.)

To facilitate multi-line substitutions, the "." character never matches a newline unless you use the `/s` modifier, which 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 subpattern is "greedy", that is, it will match as many times as possible without causing the rest 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
```

Since patterns are processed as double quoted strings, the following also work:

```
\t          tab
\n          newline
\r          return
\f          form feed
\v          vertical tab, whatever that is
\a          alarm (bell)
\e          escape (think troff)
\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
```

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+`. 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 only at beginning of string
\Z  Match only at end of string
\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 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.

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. (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 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 parens before the backreference. Otherwise (for backward compatibilty) \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;
}
```

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 metacharacter. This makes it simple to quote a string that you want to use for a pattern but that you are afraid might contain metacharacters. Simply quote 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 5 defines a consistent extension syntax for regular expressions. The syntax is a pair of parens with a question mark as the first thing within the parens (this was a syntax error in Perl 4). 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 backrefences 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 merely need to include `(?i)` at the front of the pattern. For example:

```
$pattern = "foobar";
if ( /$pattern/i )

# more flexible:

$pattern = "(?i)foobar";
if ( /$pattern/ )
```

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...

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 *metacharacter* 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 metacharacter syntax much like that used in C: "\n" matches a newline, "\t" a tab, "\r" a carriage return, "\f" a form feed, etc. More generally, *nnn*, where *nnn* is a string of octal digits, matches the character whose ASCII value is *nnn*. Similarly, \x*nn*, where *nn* are hexidecimal digits, matches the character whose ASCII value is *nn*. The expression \c*x* matches the ASCII character control-*x*. Finally, the "." metacharacter matches any character except "\n" (unless you use `/s`).

You can specify a series of alternatives for a pattern using "|" to separate them, so that `fee|fie|foe` will match any of "fee", "fie", or "foe" in the target string (as would `f(e|i|o)e`). 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 subpatterns for later reference by enclosing them in parentheses, and you may refer back to the *n*th subpattern later in the pattern using the metacharacter *n*. Subpatterns are numbered based on the left to right order of their opening parenthesis. Note that a backreference matches whatever actually matched the subpattern in the string being examined, not the rules for that subpattern. Therefore, `(0|0x)\d*\s\1\d*` will match "0x1234 0x4321",but not "0x1234 01234", since subpattern 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 right-hand 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 [switches] filename args

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 only works 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.)

As of Perl 5, the #! line is always examined for switches as the line is being parsed. Thus, if you're on a machine that only allows 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 perl $0 -S ${1+"$@"}'
    if 0;
```

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.

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 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 since 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, since these are considered as occurring outside the execution of your program.

–d runs the script under the Perl debugger. See the *perldebug* manpage.

–D*number*

–D*list*

sets debugging flags. To watch how it executes your script, use **–D14**. (This only works 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*regexp*

specifies a regular expression to split on if **–a** is also in effect. If regexp has `//` around it, the slashes will be ignored.

–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 the `eof` entry in the *perlfunc* manpage).

–I*directory*

may be used in conjunction with **–P** to tell the C preprocessor where to look for include files. By default /usr/include and /usr/lib/perl are searched.

–l*octnum*

enables automatic line-ending processing. It has two effects: first, it automatically chomps the line terminator when used with **–n** or **–p**, and second, it assigns "`$\`" to have the value of *octnum* so that any print statements will have that line terminator added back on. If *octnum* is omitted, sets "`$\`" 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.

−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. (Since 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 $*"
        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. In order 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 0;
```

−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 the *perlsec* manpage.

−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.

−w prints warnings about identifiers 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 readonly 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. See the *perldiag* manpage and the *perltrap* manpage.

−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 (but only one group of switches, as with normal #! processing). If a directory name is specified, Perl will switch to that directory before running the script. The **−x** switch only controls the 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).

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 the *perlop* manpage.) 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 only ever be 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 parens.) If you use the parens, 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+3;        # Prints 6.
print(1+2) + 3;     # Prints 3.
print (1+2)+3;      # Also prints 3!
print +(1+2)+3;     # Prints 6.
print ((1+2)+3);    # Prints 6.
```

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, 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

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()* in order 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

Returns the absolute value of its argument.

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 the section on *Sockets: Client/Server Communication* in the *perlipc* manpage.

alarm SECONDS

Arranges to have a SIGALRM delivered to this process after the specified number of seconds have elapsed. (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 the `select()` entry elsewhere in this documentbelow. It is not advised to intermix *alarm()* and *sleep()* calls.

atan2 Y,X

Returns the arctangent of Y/X in the range $-\pi$ to π.

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 the section on *Sockets: Client/Server Communication* in the *perlipc* manpage.

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 referenced object (passed as 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, since 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 the *perlobj* manpage 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 TRUE if there is a caller, that is, if we’re in a subroutine or *eval()* or *require()*, and FALSE 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, $wantargs) = caller($i);
```

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. Returns the number of files successfully changed.

```
$cnt = chmod 0755, 'foo', 'bar';
chmod 0755, @executables;
```

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 number of characters removed. 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
: Returns the character represented by that NUMBER in the character set. For example, `chr(65)` is "A" in ASCII.

chroot FILENAME
: 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 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, since *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 the section on *Sockets: Client/Server Communication* in the *perlipc* manpage.

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 `$_`.

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.

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 whoever asks you for it is unwise.

dbmclose ASSOC_ARRAY

[This function has been superseded by the *untie()* function.]

Breaks the binding between a DBM file and an associative array.

dbmopen ASSOC,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 an associative array. ASSOC is the name of the associative array. (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 only supports 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 associative array variables, not set them. If you want to test whether you can write, either use file tests or try setting a dummy array 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 the *AnyDBM_File* manpage for a more general description of the pros and cons of the various dbm apparoches, as well as the *DB_File* manpage for a particularly rich implementation.

defined EXPR

Returns a boolean value saying whether EXPR has a real value or not. Many operations return the undefined value under exceptional conditions, such as end of file, uninitialized variable, system error and such. This function allows you to distinguish between an undefined null scalar and a defined null scalar with operations that might return a real null string, such as referencing elements of an array. You may also check to see if arrays or subroutines exist. Use of defined on predefined variables is not guaranteed to produce intuitive results.

When used on a hash array element, it tells you whether the value is defined, not whether the key exists in the hash. Use *exists()* for that.

Examples:

```
print if defined $switch{'D'};
print "$val\n" while defined($val = pop(@ary));
die "Can't readlink $sym: $!"
    unless defined($value = readlink $sym);
eval '@foo = ()' if defined(@foo);
die "No XYZ package defined" unless defined %_XYZ;
sub foo { defined &$bar ? &$bar(@_) : die "No bar"; }
```

See also *undef()*.

delete EXPR

Deletes the specified value from its hash array. Returns the deleted value, or the undefined value if nothing was deleted. Deleting from `$ENV{}` modifies the environment. Deleting from an array 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 an associative array:

```
foreach $key (keys %ARRAY) {
    delete $ARRAY{$key};
}
```

(But it would be faster to use the *undef()* command.) Note that the EXPR can be arbitrarily complicated as long as the final operation is a hash key lookup:

```
delete $ref->[$x][$y]{$key};
```

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)` (backtick 'command' status). If `($? >> 8)` is 0, exits with 255. Inside an *eval()*, the error message is stuffed into `$@`, and the *eval()* is terminated with the undefined value; this makes *die()* the way to raise an exception.

Equivalent examples:

```
die "Can't cd to spool: $!\n" unless chdir '/usr/spool/news';
chdir '/usr/spool/news' or die "Can't cd to spool: $!\n"
```

If the 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()*.

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 the *perlsub* manpage.

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 the section on *Predefined Names* in the *perlvar* manpage). It's the same, however, in that it does reparse the file every time you call it, so you probably don't want to do this inside a loop.

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 the *perlrun* manpage.

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 ASSOC_ARRAY

Returns a 2-element array consisting of the key and value for the next value of an associative array, so that you can iterate over it. Entries are returned in an apparently random order. When the array is entirely read, a null array is returned (which when assigned produces a FALSE (0) value). The next call to *each()* after that will start iterating again. The iterator can be reset only by reading all the elements from the array. You should not add elements to an array while you're iterating over it. There is a single iterator for each associative array, shared by all *each()*, *keys()* and *values()* function calls in the program. 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 pseudofile 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. Testing `eof`

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, subroutine or 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.

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.

Note that, since *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 $@
```

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 at all. (Case 4 is preferred for purely visual reasons.) 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*. Use the *system()* function 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*(0) 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 only be TRUE 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.

exp EXPR

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) on FILEHANDLE. See the *flock(2)* manpage for definition of OPERATION. Returns TRUE for success, FALSE on failure. Will produce a fatal error if used on a machine that doesn't implement either *flock*(2) or *fcntl*(2). The *fcntl*(2) system call will be automatically used if *flock*(2) is missing from your system. This makes *flock()* the portable file locking strategy, although it will only lock entire files, not records. 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.

Here's a mailbox appender for BSD systems.

```
$LOCK_SH = 1;
$LOCK_EX = 2;
$LOCK_NB = 4;
$LOCK_UN = 8;

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 the *DB_File* manpage 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()* FileHandle method 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 the *perlipc* manpage for more examples of forking and reaping moribund children.

format
Declare a picture format with use by the *write()* function. For example:

```
format Something =
    Test: @<<<<<<<< @||||| @>>>>>
          $str,     $%,    '$' . int($num)
.

$str = "widget";
$num = $cost/$quantiy;
$~ = 'Something';
write;
```

See the *perlform* manpage for many details and examples.

formline PICTURE, LIST
This is an internal function used by `formats`, though you may call it too. It formats (see the *perlform* manpage) 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, since an "`@`" character may be taken to mean the beginning of an array name. *formline()* always returns TRUE. See the *perlform* manpage 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 to whether `$BSD_STYLE` should be set is left as an exercise to the reader.

See also the `Term::ReadKey` module from your nearest CPAN site; details on CPAN can be found on the `CPAN` entry in the *perlmod* manpage getlogin Returns the current login from */etc/utmp*, if any. If null, use *getpwuid()*.

```
$login = getlogin || (getpwuid($<))[0] || "Kilroy";
```

Do not consider *getlogin()* for authorentication: 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, 0 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.

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 the *getpriority(2)* manpage.) 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

Returns the value of EXPR with filename expansions such as a shell would do. This is the internal function implementing the <*.*> operator, except it's easier to use.

gmtime EXPR

Converts a time as returned by the time function to a 9-element array with the time localized for the standard Greenwich timezone. 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. If EXPR is omitted, does `gmtime(time())`.

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. 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

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, since `$_` 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.

hex EXPR

Interprets EXPR as a hex string and returns the corresponding decimal value. (To convert strings that might start with 0 or 0x see *oct()*.) If EXPR is omitted, uses `$_`.

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 the `use` entry elsewhere in this documentthe *perlmod* manpage, and the *Exporter* manpage.

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

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 the `split` entry in the *perlfunc* manpage.

keys ASSOC_ARRAY
Returns a normal array consisting of all the keys of the named associative array. (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 associative array has not been modified). 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;
}
```

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 the the section on *Signals* in the *perlipc* manpage man page 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
Returns an lowercased version of EXPR. This is the internal function implementing the \L escape in double-quoted strings. Should respect any POSIX *setlocale()* settings.

lcfirst EXPR

Returns the value of EXPR with the first character lowercased. This is the internal function implementing the \l escape in double-quoted strings. Should respect any POSIX *setlocale()* settings.

length EXPR

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 the section on *Sockets: Client/Server Communication* in the *perlipc* manpage.

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 parens. See L<perlsub/"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 L<perlsub/"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 timezone. 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. If EXPR is omitted, does *localtime* (time).

In a scalar context, prints out the *ctime* (3) value:

```
$now_string = localtime;  # e.g. "Thu Oct 13 04:54:34 1994"
```

See also the `timelocal` entry in the *perlmod* manpage and the *strftime* (3) function available via the POSIX modulie.

log EXPR

Returns logarithm (base *e*) of EXPR. If EXPR is omitted, returns log of `$_`.

lstat FILEHANDLE

lstat EXPR

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.

m// The match operator. See the *perlop* manpage.

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 parens. See the section on *Private Variables via my()* in the *perlsub* manpage 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

Interprets EXPR as an octal string and returns the corresponding decimal 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 `$_`.

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. If the filename begins with "<" or nothing, the file is opened for input. If the filename begins with ">", the file is 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 usually preferred for read/write updates—the '+>' mode would clobber the file first. These 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 the section on *Using open() for IPC* in the *perlipc* manpage 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 See the *open2* manpage, the *open3* manpage, and the section on *Bidirectional Communication* in the *perlipc* manpage 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 the `binmode` entry elsewhere in this documentfor 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 acount 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 the section on *Safe Pipe Opens* in the *perlipc* manpage 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.

The filename that is passed to open will have leading and trailing whitespace deleted. In order 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 L<*open* (2)) on your system, then you should probably use the *POSIX::open()* function as found in the the *POSIX* manpage documents. For example:

```
use FileHandle;
use POSIX qw(:fcntl_h);
$fd = POSIX::open($path, O_RDWR|O_CREAT|O_EXCL, 0700);
die "POSIX::open $path: $!" unless defined $fd;
$fh = FileHandle->new_from_fd($fd, $amode) || die "fdopen: $!";
$fh->autoflush(1);
$fh->print("stuff $$\n");
seek($fh, 0, SEEK_SET);
print "File contains: ", <$fh>;
```

See the `seek()` entry elsewhere in this documentfor 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

Returns the numeric ascii value of the first character of EXPR. If EXPR is omitted, uses `$_`.

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.

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” and “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 only affects 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 merely influences which symbol table is used by the compiler for the rest of that block. You can refer to variables and filehandles in other packages by prefixing the identifier with the package name and a double colon: `$Package::Variable`. If the package name is null, the `main` package as assumed. That is, `$::sail` is equivalent to `$main::sail`.

See the section on *Packages* in the *perlmod* manpage for more information about packages, modules, and classes. See the *perlsub* manpage 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 the *open2* manpage, the *open3* manpage, and the section on *Bidirectional Communication* in the *perlipc* manpage for examples of such things.

pop ARRAY

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

Returns the offset of where the last `m//g` search left off for the variable in question. May be modified to change that offset.

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 parens 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 parens 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 LIST

printf LIST

Equivalent to a "print FILEHANDLE *sprintf*(LIST)". The first argument of the list will be interpreted as the printf format.

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 the *perlop* manpage.

quotemeta EXPR

Returns the value of EXPR with with all regular expression metacharacters backslashed. This is the internal function implementing the \Q escape in double-quoted strings.

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. This function produces repeatable sequences unless *srand()* is invoked. 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. As a workaround, you can usually multiply EXPR by the correct power of 2 to get the range you want. This will make your script unportable, however. It's better to recompile if you can.)

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, since 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

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 the section on *UDP: Message Passing* in the *perlipc* manpage 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

Returns a TRUE value if EXPR is a reference, FALSE otherwise. 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($r) eq "HASH") {
    print "r is a reference to an associative array.\n";
}
if (!ref ($r) {
    print "r is not a reference at all.\n";
}
```

See also the *perlref* manpage.

rename OLDNAME,NEWNAME

Changes the name of a file. Returns 1 for success, 0 otherwise. Will not work across filesystem 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 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 the the `use()` entry elsewhere in this documentthe *perlmod* manpage.

reset EXPR

reset Generally used in a `continue` block at the end of a loop to clear variables and reset ?? searches so that they work again. The expression is interpreted as a list of single characters (hyphens allowed for ranges). All variables and arrays beginning with one of those letters are reset to their pristine state. If the expression is omitted, one-match searches (?pattern?) are reset to match again. Only resets variables or searches in the current package. Always returns 1. Examples:

```
reset 'X';          # reset all X variables
reset 'a-z';        # reset lower case variables
reset;              # just reset ?? searches
```

Resetting "A–Z" is not recommended since you'll wipe out your ARGV and ENV arrays. Only resets package variables— lexical variables are unaffected, but they clean themselves up on scope exit anyway, so anymore you probably want to use them instead. See the `my` entry elsewhere in this document.

return LIST

Returns from a subroutine or eval with the value specified. (Note that in the absence of a return a subroutine or *eval()* 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, returns a string value consisting of the bytes of the first element of LIST in the opposite order.

```
print reverse <>;                       # line tac
```

```
undef $/;
print scalar reverse scalar <>;     # byte tac
```

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
: 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 the *perlop* manpage.

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 C<<FILE<>> makes Perl try again to read something. Hopefully.

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 FileHandle;
STDERR->autoflush(1);
```

select RBITS,WBITS,EBITS,TIMEOUT

This calls the *select*(2) system call with the bitmasks 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:

```
$nfound = select($rout=$rin, $wout=$win, $eout=$ein, undef);
```

Any of the bitmasks 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 250-microsecond sleep 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 the section on *UDP: Message Passing* in the *perlipc* manpage 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).

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 *push()* and *pop()* 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
: Returns the sine of EXPR (expressed in radians). If EXPR is omitted, returns sine of `$_`.

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, since *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 the `select()` entry elsewhere in this documentbelow.

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 the section on *Sockets: Client/Server Communication* in the *perlipc* manpage.

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. Nonexistent values of arrays are stripped out. 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.

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 integers
}
@sortedclass = sort byage @class;

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 and 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;
```

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 equivalencies 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 partially split a line

```
($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");
```

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 the `chop`, `chomp`, and `join` entries elsewhere in this document.)

sprintf FORMAT,LIST

Returns a string formatted by the usual printf conventions of the C language. See the *sprintf(3)* manpage or the *printf(3)* manpage 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.) Some C libraries' implementations of *sprintf()* can dump core when fed ludicrous arguments.

sqrt EXPR

Return the square root of EXPR. If EXPR is omitted, returns square root of `$_`.

srand EXPR

Sets the random number seed for the `rand` operator. If EXPR is omitted, does `srand(time)`. Many folks use an explicit `srand(time ^ $$)` instead. Of course, you'd need something much more random than that for cryptographic purposes, since it's easy to guess the current time. Checksumming the compressed output of rapidly changing operating system status programs is the usual method. Examples are posted regularly to the comp.security.unix newsgroup.

stat FILEHANDLE

stat EXPR

Returns a 13-element array giving the status info for a file, either the file opened via FILEHANDLE, or named by EXPR. 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);
```

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 only works on machines 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 runtimes 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 delim
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 the *perlsub* manpage and the *perlref* manpage 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. 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 only supports passing of up to 14 arguments to your system call, which in practice should usually suffice.

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 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.

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 the `exec` entry elsewhere in this document. This is *NOT* what you want to use to capture the output from a command, for that you should merely use backticks, as described in the section on *'STRING'* in the *perlop* manpage.

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. An OFFSET may be specified to place the read data at some other place than the beginning 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 an associative array 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 the *DB_File* manpage or the *Config* module for interesting *tie()*

implementations.

time Returns the number of non-leap seconds since 00:00:00 UTC, January 1, 1970. 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 the *perlop* manpage.

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 Returns an uppercased version of EXPR. This is the internal function implementing the \U escape in double-quoted strings. Should respect any POSIX *setlocale()* settings.

ucfirst EXPR
Returns the value of EXPR with the first character uppercased. This is the internal function implementing the \u escape in double-quoted strings. Should respect any POSIX *setlocale()* settings.

umask EXPR

umask Sets the umask for the process and returns the old one. If EXPR is omitted, merely returns current umask.

undef EXPR

undef Undefines the value of EXPR, which must be an lvalue. Use only on a scalar value, an entire array, 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. Examples:

```
undef $foo;
undef $bar{'blurfl'};
undef @ary;
undef %assoc;
undef &mysub;
return (wantarray ? () : undef) if $they_blew_it;
```

unlink LIST
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.

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 merely returns 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
: 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; }
```

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 the *Exporter* manpage.

If you don't want your namespace altered, explicitly supply an empty list:

```
use Module ();
```

That is exactly equivalent to

```
BEGIN { require Module; }
```

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 pseudomodules 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.

```
no integer;
no strict 'refs';
```

See the *perlmod* manpage 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 ASSOC_ARRAY
: Returns a normal array consisting of all the values of the named associative array. (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 array. See also *keys()* and *each()*.

vec EXPR,OFFSET,BITS
: Treats a string as a vector of unsigned integers, and returns the value of the bitfield specified. May also be assigned to. BITS must be a power of two from 1 to 32.

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 "wait_h";
...
waitpid(-1,&WNOHANG);
```

then you can do a non-blocking wait for any process. Non-blocking wait is only 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 on an exception.

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 the *perlform* manpage.

Note that write is *NOT* the opposite of read. Unfortunately.

y/// The translation operator. See the section on *tr///* in the *perlop* manpage.

NAME

perlvar – Perl predefined variables

DESCRIPTION

Predefined Names

The following names have special meaning to Perl. Most of the punctuational 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 (<>) {...}     # only equivalent in 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 subpattern 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

$* Set to 1 to do multiline matching within a string, 0 to tell Perl that it can assume that strings contain a single line, for the purpose of optimizing pattern matches. Pattern matches on strings containing multiple newlines can produce confusing results when "`$*`" is 0. Default is 0. (Mnemonic: * matches multiple things.) Note that this variable only influences the interpretation of "`^`" and "`$`". A literal newline can be searched for even when `$* == 0`.

Use of "`$*`" is deprecated in Perl 5.

input_line_number HANDLE EXPR

$INPUT_LINE_NUMBER

$NR

$. The current input line number of the last filehandle that was read. An explicit close on the filehandle resets the line number. Since "<>" never does an explicit close, line numbers increase across ARGV files (but see examples under *eof()*). Localizing `$.` 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 blank lines as delimiters if set to the null string. 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 blank lines. Setting it to `""` will treat two or more consecutive blank lines as a single blank 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;
```

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. 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. In order 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

$\ The output record separator for the print operator. Ordinarily the print operator simply prints out the comma separated fields you specify, with no trailing newline or record separator assumed. In order to get behavior more like **awk**, set this variable as you would set **awk**'s ORS variable to specify what is printed at the end of the print. (Mnemonic: you set "`$\`" 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 in Perl 5.

$OFMT

$# The output format for printed numbers. This variable is a half-hearted attempt to emulate **awk**'s OFMT variable. There are times, however, when **awk** and Perl have differing notions of what is in fact numeric. Also, the initial value is %.20g rather than %.6g, so you need to set "`$#`" explicitly to get **awk**'s value. (Mnemonic: # is the number sign.)

Use of "`$#`" is deprecated in Perl 5.

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 formfeed. 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 the *perlform* manpage and the `formline()` entry in the *perlfunc* manpage.

$CHILD_ERROR

$? The status returned by the last pipe close, backtick (``` `` ```) command, or *system()* operator. Note that this is the status word returned by the *wait()* system call, so the exit value of the subprocess is actually (`$? >> 8`). Thus on many systems, `$? & 255` gives which signal, if any, the process died from, and whether there was a core dump. (Mnemonic: similar to **sh** and **ksh**.)

$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 "`$!`" in order to set *errno* if, for instance, you want "`$!`" to return the string for error *n*, or you want to set the exit value for the *die()* operator. (Mnemonic: What just went bang?)

$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 only be swapped on machines supporting *setreuid()*.

$REAL_GROUP_ID

$GID

$(The real gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by *getgid()*, and the subsequent ones by *getgroups()*, one of which may be the same as the first number. (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 only be set on machines that support the corresponding *set[re][ug]id()* routine. "`$(`" and "`$)`" can only be swapped on 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.

$INPLACE_EDIT

$^I The current value of the inplace-edit extension. Use `undef` to disable inplace editing. (Mnemonic: value of **–i** switch.)

$PERLDB

$^P The internal flag that the debugger clears so that it doesn't debug itself. You could conceivable 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 `$#ARGV` is the generally number of arguments minus one, since `$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 in order to also get the machine-dependent library properly loaded:

```
use lib '/mypath/libdir/';
use SomeMod;

=item %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 only contains values for 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 the *perlsubs* manpage.

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__.

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 parens.
NAME LIST;      # Parens optional if predeclared/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.

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 the *perlref* manpage) 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 the sections below on L<"Private Variables via my()"> and the section on */"Temporary Values via local()*. To create protected environments for a set of functions in a separate package (and probably a separate file), see the section on *Packages* in the *perlmod* manpage.

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, since the assignment copies the values. Otherwise a function is free to do in-place modifications of `@_` and change its callers 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/ }
    return @parms;
}
```

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 the section on */"Pass by Reference* for alternatives.

A subroutine may be called using the "&" prefix. The "&" is optional in Perl 5, and so are the parens if the subroutine has been predeclared. (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 the *perlref* manpage 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"
```

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, subroutine, `eval`, or `do/require/use`'d file. If more than one value is listed, the list must be placed in parens. 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 the *perlref* manpage.)

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 righthand side, while

```
    my $foo = <STDIN>;
```

supplies a scalar context. But the following only declares 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.

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 outer most scope of a file to totally hide any such identifiers from the outside world. This is similar to a 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, since 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, merely placing 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 the *perlrun* manpage 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. Execeptions 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 *the 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 parens. 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, and so 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 righthand 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 ‘‘type glob’’, since 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 type glob 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, since 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 the section on *Typeglobs* in the *perldata* manpage.

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; see the *perlref* manpage.

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 just using the normal list return mechanism. What happens if you want to pass or return a hash? Well, if you're only using 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 just sets `@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, $dref);
    } else {
        return ($cref, $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, since 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>;
}
```

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 the `open()` entry in the *perlfunc* manpage for a somewhat cleaner way using the FileHandle functions supplied with the POSIX package.

Prototypes

As of the 5.002 release of perl, if you declare

```
sub mypush (\@@)
```

then *mypush()* takes arguments exactly like *push()* does. (This only works for function calls that are visible at compile time, not indirect function calls through a `&$func` reference nor for method calls as described in the *perlobj* manpage.)

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 must be passed something starting with that character. 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, and * 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.

Note that the last three are syntactically distinguished by the lexer. *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 argumentless, 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 (&) { @_ }

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? (Nevermind.))))

And here's a reimplementation of grep:

```
sub mygrep (&@) {
    my $code = shift;
    my @result;
    foreach $_ (@_) {
        push(@result, $_) if &$ref;
    }
    @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 elments 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 only be used in moderation to make the world a better place.

Overriding Builtin Functions

Many builtin functions may be overridden, though this should only be tried 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 only be done by importing the name from a module—ordinary predeclaration isn't good enough. However, the `subs` pragma (compiler directive) lets you, in effect, predeclare 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, since 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 preclare 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 the *Autoloader* manpage, the standard SelfLoader modules in the *SelfLoader* manpage, and the document on adding C functions to perl code in the *perlxs* manpage.

SEE ALSO

See the *perlref* manpage for more on references. See the *perlxs* manpage if you'd like to learn about calling C subroutines from perl. See the *perlmod* manpage 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 others 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 only affects 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 merely influences which symbol table is used by the compiler for the rest of that block. You can refer to variables and filehandles in other packages by prefixing the identifier with the package name and a double colon: `$Package::Variable`. If the package name is null, the `main` package as assumed. That is, `$::sail` is equivalent to `$main::sail`.

(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 the *perldebug* manpage.

See the *perlsub* manpage for other scoping issues related to *my()* and *local()*, or the *perlref* manpage regarding closures.

Symbol Tables

The symbol table for a package happens to be stored in the associative array of that name appended with two colons. The main symbol table's name is thus `%main::`, or `%::` for short. Likewise the nested package mentioned earlier is named `%OUTER::INNER::`.

The value in each entry of the associative array 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 symbol `dick`. If you only want to alias 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 wil overwrite the hash slot in the symbol table specified by the *some_hash typeglob. This is a somewhat tricky way of passing around refernces 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.

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).

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 the *perlobj* manpage.

Perl Modules

A module is a just 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 Fred, create a file called Fred.pm and put this at the start of it:

```
package      Fred;
require      Exporter;
@ISA       = qw(Exporter);
@EXPORT    = qw(func1 func2);
@EXPORT_OK = qw($sally @listabob %harry func3);
```

Then go on to declare and use your variables in functions without any qualifications. See the *Exporter* manpage 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 just say `use POSIX` to get it all.

For more information on writing extension modules, see the *perlxs* manpage and the *perlguts* manpage.

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 (AKA 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 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 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, so is far from bullet proof.

Pragmatic Modules

They work somewhat like pragmas in that they tend to affect the compilation of your program, and thus will usually only work well when used within a `use`, or `no`. 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.

The following programs are defined (and have their own documentation).

diagnostics	Pragma to produce enhanced diagnostics
integer	Pragma to compute arithmetic in integer instead of double
less	Pragma to request less of something from the compiler
overload	Pragma for overloading operators
sigtrap	Pragma to enable stack backtrace on unexpected signals
strict	Pragma to restrict unsafe constructs
subs	Pragma to predeclare sub 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
Carp	warn of errors (from perspective of caller)
Config	access Perl configuration option
Cwd	get pathname of current working directory
DB_File	Perl access to Berkeley DB
Devel::SelfStubber	generate stubs for a SelfLoading module
DynaLoader	Dynamically load C libraries into Perl code
English	use nice English (or awk) names for ugly punctuation variables
Env	perl module that imports environment variables
Exporter	provide inport/export controls for Perl modules
ExtUtils::Liblist	determine libraries to use and how to use them
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::Miniperl	!!!GOOD QUESTION!!!
Fcntl	load the C Fcntl.h defines
File::Basename	parse file specifications
File::CheckTree	run many filetest checks on a tree
File::Find	traverse a file tree
FileHandle	supply object methods for filehandles
File::Path	create or remove a series of directories
Getopt::Long	extended getopt processing
Getopt::Std	Process single-character switches with switch clustering
I18N::Collate	compare 8-bit scalar data according to the current locale
IPC::Open2	a process for both reading and writing
IPC::Open3	open a process for reading, writing, and error handling
Net::Ping	check a host for upness
POSIX	Perl interface to IEEE Std 1003.1
SelfLoader	load functions only on demand
Socket	load the C socket.h defines and structure manipulators
Test::Harness	run perl standard test scripts with statistics
Text::Abbrev	rceate an abbreviation table from a list

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 size, 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 unbunded 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

Some of the reguster CPAN sites as of this writing include the following. You should try to choose one close to you:

- ftp://ftp.sterling.com/programming/languages/perl/
- ftp://ftp.sedl.org/pub/mirrors/CPAN/

- ftp://ftp.uoknor.edu/mirrors/CPAN/
- ftp://ftp.delphi.com/pub/mirrors/packages/perl/CPAN/
- ftp://uiarchive.cso.uiuc.edu/pub/lang/perl/CPAN/
- ftp://ftp.cis.ufl.edu/pub/perl/CPAN/
- ftp://ftp.switch.ch/mirror/CPAN/
- ftp://ftp.sunet.se/pub/lang/perl/CPAN/
- ftp://ftp.ci.uminho.pt/pub/lang/perl/
- ftp://ftp.cs.ruu.nl/pub/PERL/CPAN/
- ftp://ftp.demon.co.uk/pub/mirrors/perl/CPAN/
- ftp://ftp.rz.ruhr-uni-bochum.de/pub/programming/languages/perl/CPAN/
- ftp://ftp.leo.org/pub/comp/programming/languages/perl/CPAN/
- ftp://ftp.pasteur.fr/pub/computing/unix/perl/CPAN/
- ftp://ftp.ibp.fr/pub/perl/CPAN/
- ftp://ftp.funet.fi/pub/languages/perl/CPAN/
- ftp://ftp.tekotago.ac.nz/pub/perl/CPAN/
- ftp://ftp.mame.mu.oz.au/pub/perl/CPAN/
- ftp://coombs.anu.edu.au/pub/perl/
- ftp://dongpo.math.ncu.edu.tw/perl/CPAN/
- ftp://ftp.lab.kdd.co.jp/lang/perl/CPAN/
- ftp://ftp.is.co.za/programming/perl/CPAN/

For an up-to-date listing of CPAN sites, see http://www.perl.com/perl/ 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 5 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 hard-wired class names as far as possible.

Avoid `$r`−>*Class::func()* where using `@ISA`=qw(... Class ...) and `$r`−>*func()* would work (see perlbot man page 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 sub-class' 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 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.

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 informally group or categorise a module. A module should have a very good reason 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 licence your work is a personal decision. The general mechanism is to assert your Copyright and then make a declaration of how others may copy/use/modify your work.

Perl, for example, is supplied with two types of licence: The GNU GPL and The Artistic 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 simply state something 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 `$VERSION`. This should be a valid floating point number with at least two digits after the decimal (ie hundredths, e.g, `$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 it's 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

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 5 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. Since 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. If that thing happens to be an object, the object is destructed. See the *perlobj* manpage 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 merely contains 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, since 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   = \*STDOUT;
```

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!
```

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 the *perlobj* manpage.

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 only applies to lexical variables, by the way. Dynamic variables continue to work as they have always worked. Closure is not something that most Perl programmers need trouble themselves about to begin with.

5. References are often returned by special subroutines called constructors. Perl objects are just references to a special 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. Since we haven't talked about dereferencing yet, we can't show you any examples yet.

7. References to filehandles can be created by taking a reference to a typeglob. This is currently the best way to pass filehandles into or out of subroutines, or to store them in larger data structures.

```
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>;
}
```

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 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 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 you use FileHandle
```

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 parens 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 the *perlfunc* manpage.

The *bless()* operator may be used to associate a reference with a package functioning as an object class. See the *perlobj* manpage.

A typeglob may be dereferenced the same way a reference can, since 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 yeilds @{[$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"} = 5;    # 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 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 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 just write

```
$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, since 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 attemping. 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 the *perldsc* manpage and the *perllol* manpage for how to use references to create complex data structures, and the *perlobj* manpage 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 just say `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 documents on each of the following:

- arrays of arrays
- hashes of arrays
- arrays of hashes
- hashes of hashes
- more elaborate constructs
- recursive and self-referential data structures
- objects

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 only hold 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.) That means that when you have something that looks to you like an access to two-or-more-dimensional array and/or hash, that what's really going on is that in all these cases, 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 only contains 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, just 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 to always 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", "alroy", "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 instead write:

```
print $listref->[2][2]
```

DEBUGGING

The standard Perl debugger in 5.001 doesn't do a very nice job of printing out complex data structures. However, the perl5db that Ilya Zakharevich <*ilya@math.ohio-state.edu*> wrote, which is accessible at

```
ftp://ftp.perl.com/pub/perl/ext/perl5db-kit-0.9.tar.gz
```

has 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  'alroy'
      3  'judy'
```

There's also a lower-case **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{$b}} } keys %HoL ) {
    print "$family: @{ $HoL{$family} }\n"

# print the whole thing sorted by number of members and name
foreach $family ( sort { @{$HoL{$b}} <=> @{$HoL{$a}} } keys %HoL ) {
    print "$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 list, like
# "lead","fred","daughter","pebbles"
while ( %fields = getnextpairset() )
    push @a, { %fields };

# 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{$b}} } keys %HoH ) {
    print "$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{$b}} } keys %HoH ) {
    print "$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 = {
    STRING  => $string,
    LIST    => [ @old_values ],
    LOOKUP  => { %some_table },
    FUNC    => \&some_function,
    FANON   => sub { $_[0] ** $_[1] },
    FH      => \*STDOUT,
};

print $rec->{STRING};
```

```
print $rec->{LIST}[0];
$last = pop @ { $rec->{LIST} };

print $rec->{LOOKUP}{"key"};
($first_k, $first_v) = each %{ $rec->{LOOKUP} };

$answer = &{ $rec->{FUNC} }($arg);
$answer = &{ $rec->{FANON} }($arg1, $arg2);

# careful of extra block braces on fh ref
print { $rec->{FH} } "a string\n";

use FileHandle;
$rec->{FH}->autoflush(1);
$rec->{FH}->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 datastructures 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";
}
```

SEE ALSO

the *perlref* manpage, the *perllol* manpage, the *perldata* manpage, the *perlobj* manpage

AUTHOR

Tom Christiansen <*tchrist@perl.com*>

Last update: Tue Dec 12 09:20:26 MST 1995

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 parens. 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 array. But you need not 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 leary 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 not only have to declare `$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 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 just dealing with 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 just wanted 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* just say:

```
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 only want one of the elements, it's trivial:

```
print $LoL[0][0];
```

If you want to print the whole thing, though, you can't just say

```
print @LoL;          # WRONG
```

because you'll just get 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";
    }
}
```

Hm... 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 slide (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? Hm... 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 the *perlref* manpage for that.

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 static 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, since 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 L<perlmod/"Modules: Creation, Use and Abuse">), 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 only be accessed 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 only belong to 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 simply uses the reference in order 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 just 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. If that doesn't work, Perl finally gives up and complains.

Perl classes only do 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 static and virtual, in honor of the two C++ method types they most closely resemble.

A static 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 static methods. Many static methods simply ignore their first argument, since they already know what package they're in, and don't care what package they were invoked via. (These aren't necessarily the same, since static methods follow the inheritance tree just like ordinary virtual methods.) Another typical use for static methods is to look up an object by name:

```
sub find {
    my ($class, $name) = @_;
    $objtable{$name};
}
```

A virtual 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 static or virtual methods. We'll use the two methods defined above, the static method to lookup an object reference and the virtual 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 parens 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 merely wish 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 pseudoclass, which says to start looking in your base class's `@ISA` list without having to explicitly name it:

```
$self->SUPER::display('Height', 'Weight');
```

Please note that the `SUPER::` construct is *only* meaningful 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);
```

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 reblessed a reference from one of your base classes, your DESTROY may need to call DESTROY for any base classes that need it. But this only applies to reblessed 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 just need 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 multithreadable language. For example, this program demonstrates Perl's two-phased garbage collection:

```
#!/usr/bin/perl
package Subtle;

sub new {
    my $test;
    $test = \$test;
    warn "CREATING " . \$test;
    return bless \$test;
}
```

```
sub DESTROY {
    my $self = shift;
    warn "DESTROYING $self";
}

package main;

warn "starting program";
{
    my $a = Subtle->new;
    my $b = Subtle->new;
    $$a = 0;  # break selfref
    warn "leaving block";
}

warn "just exited block";
warn "time to die...";
exit;
```

When run as */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

You should also check out the *perlbot* manpage for other object tricks, traps, and tips, as well as the *perlmod* manpage 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

untie VARIABLE
```

DESCRIPTION

Prior to release 5.0 of Perl, a programmer could use *dbmopen()* to magically connect an on-disk database in the standard Unix *dbm*(3x) format 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()*, or *TIEHASH()*. (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.)

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::TieScalar got non-numeric pid $pid" if $^W;
        return undef;
    }

    unless (kill 0, $pid) { # EPERM or ERSCH, no doubt
        carp "Nice::TieScalar 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. Since in this case we're just using 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 ncessary, since 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, since 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 C<{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 simply keep them at 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 sclar 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()*), associative arrays have the most complete and useful *tie()* implementation. A class implementing a tied associative array 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 merely inherit from the standard TieHash module for most of your methods, redefining only the interesting ones. See the *TieHash* manpage 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 dotfiles. You index into the hash with the name of the file (minus the dot) and you get back that dotfile'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 dotfiles live

CLOBBER
: whether we should try to change or remove those dot files

LIST
: the hash of dotfile 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 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, since 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, since 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";
```

Where 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};
    unlink($file) || carp "@{[&whowasi]}: can't unlink $file: $!";
}
```

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 dotfiles! 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 dotfiles 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}};
    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 our using a real hash so we'll just do the simple thing, but we'll have to indirect through the LIST field.

```
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 isn't implemented yet. Sorry; maybe someday.

SEE ALSO

See the *DB_File* manpage or the *Config* manpage 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()*.

AUTHOR

Tom Christiansen

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 the section on *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 the section on *OBJECT RELATIONSHIPS*, the section on *USING RELATIONSHIP WITH SDBM*, and the section on *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 the section on *CLASS CONTEXT AND THE OBJECT*.
7. IO syntax is certainly less noisy, but it is also prone to ambiguities which 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 the section on *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 TieHash;
@ISA = qw( TieHash );
```

```
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 to just 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 TieHash;
@ISA = qw(TieHash);

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

perldebug – Perl debugging

DESCRIPTION

First of all, have you tried using the **–w** switch?

Debugging

If you invoke Perl with a **–d** switch, your script will be run under the debugger. However, the Perl debugger is not a separate program as it is in a C environment. Instead, the **–d** flag tells the compiler to insert source information into the pseudocode it's about to hand to the interpreter. (That means your code must 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 before the first executable statement (but see below) and ask you for one of the following commands:

h
Prints out a help message.

T
Stack trace. If you do bizarre things to your `@_` arguments in a subroutine, the stack backtrace will not always show the original values.

s
Single step. Executes until it reaches the beginning of another statement.

n
Next. Executes over subroutine calls, until it reaches the beginning of the next statement.

f
Finish. Executes statements until it has finished the current subroutine.

c
Continue. Executes until the next breakpoint is reached.

c line
Continue to the specified line. Inserts a one-time-only breakpoint at the specified line.

<CR>
Repeat last n or s.

l min+incr
List incr+1 lines starting at min. If min is omitted, starts where last listing left off. If incr is omitted, previous value of incr is used.

l min-max
List lines in the indicated range.

l line
List just the indicated line.

l
List next window.

-
List previous window.

w line
List window (a few lines worth of code) around line.

l subname
List subroutine. If it's a long subroutine it just lists the beginning. Use "l" to list more.

/pattern/
Regular expression search forward in the source code for pattern; the final / is optional.

?pattern?
Regular expression search backward in the source code for pattern; the final ? is optional.

L
List lines that have breakpoints or actions.

S
Lists the names of all subroutines.

t
Toggle trace mode on or off.

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 is evaluated each time the statement is reached and a breakpoint is taken only if the condition is true. Breakpoints may only be set on lines that begin an executable statement. Conditions don't use `if`:

```
b 237 $x > 30
b 33 /pattern/i
```

b subname [condition]
: Set breakpoint at first executable line of subroutine.

d line
: Delete breakpoint. If line is omitted, deletes the breakpoint on the line that is about to be executed.

D
: Delete all breakpoints.

a line command
: Set an action for line. A multiline command may be entered by backslashing the newlines. This command is Perl code, not another debugger command.

A
: Delete all line actions.

< command
: Set an action to happen before every debugger prompt. A multiline command may be entered by backslashing the newlines.

> command
: Set an action to happen after the prompt when you've just given a command to return to executing the script. A multiline command may be entered by backslashing the newlines.

V package [symbols]
: 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
```

X [symbols]
: Same as as ''V'' command, but within the current package.

! number
: Redo a debugging command. If number is omitted, redoes the previous command.

! −number
: Redo the command that was that many commands ago.

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.)

command
: Execute command as a Perl statement. A missing semicolon will be supplied.

p expr
: Same as `print DB::OUT expr`. The DB::OUT filehandle is opened to /dev/tty, regardless of where STDOUT may be redirected to.

Any command you type in that isn't recognized by the debugger will be directly executed (`eval`'d) as Perl code. Leading white space will cause the debugger to think it's `NOT` a debugger command.

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 `requires` will. From your own 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;
```

Customization

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 environment variable PERL5DB to say something like this:

```
BEGIN { require "myperl5db.pl" }
```

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 in particular most people seem to expect to be there):

```
$DB::alias{'len'} = 's/^len(.*)/p length($1)/';
$DB::alias{'stop'} = 's/^stop (at|in)/b/';
$DB::alias{'.'} = 's/^\./p '
                . '"\$DB::sub(\$DB::filename:\$DB::line):\t"'
                . ',\$DB::dbline[\$DB::line]/' ;
```

Other resources

You did try the **–w** switch, didn't you?

BUGS

If your program *exit()*s or *die()*s, so does the debugger.

There's no builtin way to restart the debugger without exiting and coming back into it. You could use an alias like this:

```
$DB::alias{'rerun'} = 'exec "perl -d $DB::filename"';
```

But you'd lose any pending breakpoint information, and that might not be the right path, etc.

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 `$^Q` to a reference to a routine that will be called on each warning instead of printing it. See the *perlvar* manpage. Trappable errors may be trapped using the eval operator. See the `eval` entry in the *perlfunc* manpage.

Some of these messages are generic. Spots that vary are denoted with a `%s`, just as in a printf format. Note that some message 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.

``no'' not allowed in expression
: (F) The ``no'' keyword is recognized and executed at compile time, and returns no useful value. See the *perlmod* manpage.

``use'' not allowed in expression
: (F) The ``use'' keyword is recognized and executed at compile time, and returns no useful value. See the *perlmod* manpage.

% may only be used in unpack
: (F) You can't pack a string by supplying a checksum, since the checksumming process loses information, and you can't go the other way. See the `unpack` entry in the *perlfunc* manpage.

%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 parens. See the section on *Terms and List Operators (Leftward)* in the *perlop* manpage.

%s argument is not a HASH element
: (F) The argument to *delete()* or *exists()* must be a hash element, such as

```
$foo{$bar}
$ref->[12]->{"susie"}
```

%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 the `require` entry in the *perlfunc* manpage.

%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 the *perlre* manpage.

%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.

?+* follows nothing in regexp
  (F) You started a regular expression with a quantifier. Backslash it if you meant it literally. See the *perlre* manpage.

@ outside of string
  (F) You had a pack template that specified an absolution position outside the string being unpacked. See the `pack` entry in the *perlfunc* manpage.

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 the `accept` entry in the *perlfunc* manpage.

Allocation too large: %lx
  (F) You can't allocate more than 64K on an MSDOS machine.

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, paren 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.

Argument "%s" isn't numeric

(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 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.

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 sized are, respectively, sizeof(struct msqid_ds *), sizeof(struct semid_ds *) and sizeof(struct shmid_ds *).

Bad associative array

(P) One of the internal hash routines was passed a null HV pointer.

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.

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 places ()'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.

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 the `bind` entry in the *perlfunc* manpage.

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 "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. You can usually double the curlies to get the same effect though, since the inner curlies will be considered a block that loops once. See the `last` entry in the *perlfunc* manpage.

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. You can usually double the curlies to get the same effect though, since the inner curlies will be considered a block that loops once. See the `last` entry in the *perlfunc* manpage.

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. You can usually double the curlies to get the same effect though, since the inner curlies will be considered a block that loops once. See the `last` entry in the *perlfunc* manpage.

Can't bless non-reference value

(F) Only hard references may be blessed. This is how Perl "enforces" encapsulation of objects. See the *perlobj* manpage.

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 the *perlobj* manpage.

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 the *perlobj* manpage.

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 (type GLOB), 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 (type GLOB), 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 (type GLOB), 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 inplace 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 the *perlre* manpage.

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 the `goto` entry in the *perlfunc* manpage.

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. Since 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.

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, since 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 only be calling it out of an AUTOLOAD routine anyway. See the `goto` entry in the *perlfunc* manpage.

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 previous 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 the `require` entry in the *perlfunc* manpage.

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 it's base classes. See the *perlobj* manpage.

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 autoincrement.

Can't modify non-existent substring
: (P) The internal routine that does assignment to a *substr()* was handed a NULL.

Can't msgrcv to readonly var
: (F) The target of a msgrcv must be modifiable in order to be used as a receive buffer.

Can't open %s: %s
: (S) An inplace 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 "open2.pl". 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 the *perlsub* manpage.

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 only defined on 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 %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 only valid 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 string ("%s") as %s ref while "strict refs" in use

(F) Only hard references are allowed by "strict refs". Symbolic references are disallowed. See the *perlref* manpage.

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 delimiter brackets within expression

(F) The ${name} construct is for disambiguating identifiers in strings, not in ordinary code.

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 only be tied to 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 readonly 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 reaon while trying to process a **–e** switch. Maybe your /tmp partition is full, or clobbered.

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 the `connect` entry in the *perlfunc* manpage.

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.

Do you need to predeclare %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. Since 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 a goto, or a loop control statement.

Exiting subroutine via %s

(W) You are exiting a subroutine by unconventional means, such as a a goto, or a loop control statement.

Exiting substitution via %s

(W) You are exiting a substitution by unconventional means, such as a 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 only for 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 only intended to write the file, use ">" or ">>". See the `open` entry in the *perlfunc* manpage.

Filehandle only opened for 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 only intended to write the file, use ">" or ">>". See the `open` entry in the *perlfunc* manpage.

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 the `goto` entry in the *perlfunc* manpage.

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.

Identifier "%s::%s" used only once: possible typo

(W) Typographical errors often show up as unique identifiers. If you had a good reason for having a unique identifier, then just mention it again somehow to suppress the message.

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.

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 the *perlsec* manpage 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 the *perlsec* manpage.

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 the *perlsec* manpage.

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`, in order to determine whether the current call to `exec` should be affect the current script or a subprocess (see the `exec` entry in the *perlvms* manpage). 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 the *perlre* manpage.

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 the `last` entry in the *perlfunc* manpage.

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 the `last` entry in the *perlfunc* manpage.

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 the `last` entry in the *perlfunc* manpage.

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 the `listen` entry in the *perlfunc* manpage.

Literal @%s now requires backslash
: (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 putting 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.)

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 somehow point to a valid method. See the *perlovl* manpage.

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", since 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.

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 parens. So things like ** or +* or ?* are illegal.

Note, however, that the minimal matching quantifiers, *?, +? and ?? appear to be nested quantifiers, but aren't. See the *perlre* manpage.

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 the *perlsec* manpage.

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.

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, since Perl comes with SDBM. See the *SDBM_File* manpage.

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 the *perlref* manpage.

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 "type glob" (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 the *perlref* manpage.

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 the *perlref* manpage.

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 the *perlref* manpage.

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 the *perlref* manpage.

Not a subroutine reference in %OVERLOAD
: (F) An attempt was made to specify an entry in an overloading table that doesn't somehow point to a valid subroutine. See the *perlovl* manpage.

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 the *perlref* manpage.

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 the *perlform* manpage.

Null filename used
: (F) You can't require the null filename, especially since on many machines that means the current directory! See the `require` entry in the *perlfunc* manpage.

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 bigtime.

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, since hash lists come in key/value pairs.

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' %s: no method found,
: (F) An attempt was made to use an entry in an overloading table that somehow no longer points to a valid method. See the *perlovl* manpage.

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) The *malloc()* function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request.

page overflow
: (W) A single call to *write()* produced more lines than can fit on a page. See the *perlform* manpage.

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 readonly 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.

Parens 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 the `require` entry in the *perlfunc* manpage.

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 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 the `ioctl` entry in the *perlfunc* manpage.

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 parens 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 predeclared (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 expression 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 the *perlre* manpage.

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 the *perlform* manpage.

Scalar value @%s[%s] better written as $%s[%s]
: (W) You've used an array slice (indicated by @) to select a single value 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 only expecting 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, since Perl will not magically convert between scalars and lists for you. See the *perlref* manpage.

Script is not setuid/setgid in suidperl
: (F) Oddly, the suidperl program was invoked on a script with its setuid or setgid bit set. This doesn't make much sense.

Search pattern not terminated
: (F) The lexer couldn't find the final delimiter of a // or m{} construct. Remember that bracketing delimiters count nesting level.

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 parens aren't allowed. See the *perlre* manpage.

Sequence (?%s...) not implemented
: (F) A proposed regular expression extension has the character reserved but has not yet been written. See the *perlre* manpage.

Sequence (?%s...) not recognized
: (F) You used a regular expression extension that doesn't make sense. See the *perlre* manpage.

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 $<, 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 the `sort` entry in the *perlfunc* manpage.

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 the `sort` entry in the *perlfunc* manpage.

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 the `split` entry in the *perlfunc* manpage.

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.

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 the section on *Quote and Quotelike Operators* in the *perlop* manpage.

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 the `substr` entry in the *perlfunc* manpage.

suidperl is no longer needed since...
(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, since 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 the `semctl` entry in the *perlfunc* manpage, 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 the section on *−X* in the *perlfunc* manpage.

That use of $[is unsupported

(F) Assignment to $[is now strictly circumscribed, and interpreted as a compiler directive. You may only say 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 section on *$[* in the *perlvar* manpage.

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. Govermnment 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 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 only supports a maximum of 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 the *perlre* manpage.

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 the *perlref* manpage.

umask: argument is missing initial 0

(W) A umask of 222 is incorrect. It should be 0222, since 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 the *perlform* manpage.

Undefined sort subroutine "%s" called

(F) The sort comparison routine specified doesn't seem to exist. Perhaps it's in a different package? See the `sort` entry in the *perlfunc* manpage.

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 the `sort` entry in the *perlfunc* manpage.

Undefined top format "%s" called

(F) The format indicated doesn't seem to exist. Perhaps it's really in another package? See the *perlform* manpage.

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 byteswapping 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 paren. See the *perlre* manpage.

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 the *perlre* manpage.

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 the `chop` entry in the *perlfunc* manpage.

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 multiline 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 only from 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 the *perlref* manpage for more on this.

Variable "%s" is not exported

(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 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: unable to close filehandle %s properly.

(S) The implicit *close()* done by an *open()* got an error indication on the close(0. This usually indicates your filesystem ran out of disk space.

Warning: Use of "%s" without parens 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 parens 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 the `pack` entry in the *perlfunc* manpage.

x outside of string

(F) You had a pack template that specified a relative position after the end of the string being unpacked. See the `pack` entry in the *perlfunc* manpage.

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, since 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 the `getsockopt` entry in the *perlfunc* manpage.

\1 better written as $1

(W) Outside of patterns, backreferences live on as variables. The use of backslashes is grandfathered on the righthand 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;
```

NAME

perlform − Perl formats

DESCRIPTION

Perl has a mechanism to help you generate simple reports and charts. To facilitate this, Perl helps you lay out your output page in your code in a fashion that's 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, when to print page headers, etc. Keywords are borrowed from FORTRAN: *format()* to declare and *write()* to execute; see their entries in the *perlfunc* manpage. 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.

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 <perlfunc/*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 lower-case method names instead:

```
use FileHandle;
format_name     OUTF "My_Other_Format";
format_top_name OUTF "My_Top_Format";
```

Much better!

NOTES

Since 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 the `open()` entry in the *perlfunc* manpage) and always *write()* to MESELF instead of STDOUT. Have your child process post-processes 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;
```

WARNING

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.) Furthermore, lexical aliases will not be compiled correctly: see the `my` entry in the *perlfunc* manpage for other issues.

NAME

perlipc − Perl interprocess communication (signals, fifos, pipes, safe subprocceses, 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 an 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 with 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, just do 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 somtimes 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 {
    $SIG{CHLD} = \&REAPER;  # loathe sysV
    $waitedpid = wait;
}
$SIG{CHLD} = \&REAPER;
# now do something that forks...
```

or even the more elaborate:

```
use POSIX "wait_h";
sub REAPER {
    my $child;
    $SIG{CHLD} = \&REAPER;  # loathe sysV
    while ($child = waitpid(-1,WNOHANG)) {
        $Kid_Status{$child} = $?;
    }
}
$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, newsreader, finger program, etc.) tries to read from that file, the reading program will block and your program will supply the 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 sigs
    }
```

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 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 SPOOLER || die "bad netstat: $! $?";
```

If one can be sure that a particular program is a Perl script that is expecting filenames in @ARGV, the clever programmer can write something like this:

```
$ program f1 "cmd1|" - f2 "cmd2|" f3 < tmpfile
```

and irrespective of which shell it's called from, the Perl program will read from the file *f1*, the process *cmd1*, standard input (*tmpfile* in this case), the *f2* file, the *cmd2* command, and finally the *f3* file. Pretty nifty, eh?

You might notice that you could use backticks for much the same effect as opening a pipe for reading:

```
print grep { !/^(tcp|udp)/ } `netstat -an 2>&1`;
die "bad netstat" if $?;
```

While this is true on the surface, it's much more efficient to process the file one line or record at a time because then you don't have to read the whole thing into memory at once. It also gives you finer control of the whole process, letting you to kill off the child process early if you'd like.

Be careful to check both the *open()* and the *close()* return values. If you're *writing* to a pipe, you should also trap SIGPIPE. Otherwise, think of what happens when you start up a pipe to a command that doesn't exist: the *open()* will in all likelihood succeed (it only reflects the *fork()*'s success), but then your output will fail—spectacularly. Perl can't know whether the command worked because your command is actually running in a separate process whose *exec()* might have failed. Therefore, while readers of bogus commands just return 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;
```

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, "-|");
    unless (defined $pid) {
        warn "cannot fork: $!";
        die "bailing out" if $sleep_count++ > 6;
        sleep 10;
    }
} until defined $pid;

if ($pid) {  # parent
    print KID @some_data;
    close(KID) || 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 straigh-forward, but you can't use a pipe open or backticks safely. That's because there's no way to stop the shell from getting its hands on your arguments. Instead, use lower-level control to call *exec()* directly.

Here's a safe backtick or pipe open for read:

```
# add error processing as above
$pid = open(KID, "-|");

if ($pid) {   # parent
    while (<KID>) {
        # do something interesting
    }
    close(KID) || 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, "|-");
$SIG{ALRM} = sub { die "whoops, $program pipe broke" };

if ($pid) {  # parent
    for (@data) {
        print KID;
    }
    close(KID) || 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 multithreading. If you'd like to learn more about threading, see the *modules* file mentioned below in the the section on *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(KID, "| some program |")
```

and if you forgot 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 tridirectional I/O so you can also catch your child's STDERR, but doing so would then require an awkward *select()* loop and wouldn't allow you to use normal Perl input operations.

If you look at its source, you'll see that *open2()* uses low-level primitives like Unix *pipe()* and *exec()* 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 going to really ruin your day. Even though your `Writer` filehandle is autoflushed, 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 actually 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 hopefully its successor *IPC::Chat*) at your nearest CPAN archive as detailed in the the section on *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 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.

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 multihomed hosts:

```
#!/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');
socket(SERVER, PF_INET, SOCK_STREAM, $proto)        || die "socket: $!";
setsockopt(SERVER, SOL_SOCKET, SO_REUSEADDR, 1)     || die "setsockopt: $!";
bind(SERVER, sockaddr_in($port, INADDR_ANY))        || die "bind: $!";
listen(SERVER,5)                                    || die "listen: $!";

logmsg "server started on port $port";

my $waitedpid = 0;
my $paddr;

sub REAPER {
    $SIG{CHLD} = \&REAPER;  # loathe sysV
    $waitedpid = wait;
    logmsg "reaped $waitedpid" . ($? ? " with exit $?" : '');
}

$SIG{CHLD} = \&REAPER;

for ( $waitedpid = 0;
      ($paddr = accept(CLIENT,SERVER)) || $waitedpid;
      $waitedpid = 0, close CLIENT)
{
    next if $waitedpid;
    my($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 the *perlsec* manpage) 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 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: $!";
connect(SOCK, sockaddr_un($remote))         || 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,5)                    || 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 just use 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 that 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 that 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 of the *Socket.pm* module, which was/is/shall-be included in *perl5.001o*.

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 fo 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 filedescriptors 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 the *perlfunc* manpage, you should also check out the *modules* file at your nearest CPAN site. (See the *perlmod* manpage 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

perlsec – Perl security

DESCRIPTION

Perl is designed to make it easy to write secure setuid and setgid scripts. Unlike shells, which are based on multiple substitution passes on each line of the script, Perl uses a more conventional evaluation scheme with fewer hidden "gotchas". Additionally, since the language has more built-in functionality, it has to rely less upon external (and possibly untrustworthy) programs to accomplish its purposes.

Beyond the obvious problems that stem from giving special privileges to such flexible systems as scripts, on many operating systems, setuid scripts are inherently insecure right from the start. This is because that between the time that the kernel opens up the file to see what to run, and when the now setuid interpreter it ran turns around and reopens the file so it can interpret it, things 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.

If, however, 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. See the program **wrapsuid** in the *eg* directory of your Perl distribution for how to go about doing this.

There are some systems on which setuid scripts are free of this inherent security bug. For example, recent releases of Solaris are like this. 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 mettling, 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.

When executing a setuid script, or when you have turned on taint checking explicitly using the **–T** flag, Perl takes special precautions to prevent you from falling into any obvious traps. (In some ways, a Perl script is more secure than the corresponding C program.) Any command line argument, environment variable, or input is marked as "tainted", and may not be used, directly or indirectly, in any command that invokes a subshell, or in any command that modifies files, directories, or processes. Any variable that is set within an expression that has previously referenced a tainted value also becomes tainted (even if it is logically impossible for the tainted value to influence the variable). For example:

```
$foo = shift;               # $foo is tainted
$bar = $foo,'bar';          # $bar is also tainted
$xxx = <>;                  # Tainted
$path = $ENV{'PATH'};       # Tainted, but see below
$abc = 'abc';               # Not tainted

system "echo $foo";         # Insecure
system "/bin/echo", $foo;   # Secure (doesn't use sh)
system "echo $bar";         # Insecure
system "echo $abc";         # Insecure until PATH set

$ENV{'PATH'} = '/bin:/usr/bin';
$ENV{'IFS'} = '' if $ENV{'IFS'} ne '';

$path = $ENV{'PATH'};       # Not tainted
system "echo $abc";         # Is secure now!
```

```
open(FOO,"$foo");            # OK
open(FOO,">$foo");           # Not OK

open(FOO,"echo $foo|");      # Not OK, but...
open(FOO,"-|") || exec 'echo', $foo;        # OK

$zzz = `echo $foo`;          # Insecure, zzz tainted

unlink $abc,$foo;            # Insecure
umask $foo;                  # Insecure

exec "echo $foo";            # Insecure
exec "echo", $foo;           # Secure (doesn't use sh)
exec "sh", '-c', $foo;       # Considered secure, alas
```

The taintedness is associated with each scalar value, so some elements of an array can be tainted, and others not.

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 call or exec, but only by explicitly doing something like the last example above. You can also bypass the tainting mechanism by referencing subpatterns—Perl presumes that if you reference a substring using `$1`, `$2`, etc, you knew what you were doing when you wrote the pattern:

```
$ARGV[0] =~ /^-P(\w+)$/;
$printer = $1;               # Not tainted
```

This is fairly secure since `\w+` doesn't match shell metacharacters. Use of `/.+/` would have been insecure, but Perl doesn't check for that, so you must be careful with your patterns. This is the *ONLY* mechanism for untainting user supplied filenames if you want to do file operations on them (unless you make `$>` equal to `$<`).

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 the world. A frequently voiced gripe is that you can get this message even if the pathname to an executable is fully qualified. But Perl can't know that the executable in question isn't going to execute some other program depending on 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.

This gives us a reasonably safe way to open a file or pipe: just reset the id set to the original IDs. Here's a way to do backticks reasonably safely. Notice how the *exec()* is not called with a string that the shell could expand. By the time we get to the *exec()*, tainting is turned off, however, so be careful what you call and what you pass it.

```
die unless defined $pid = open(KID, "-|");
if ($pid) {           # parent
    while (<KID>) {
        # do something
    }
    close KID;
} else {
    $> = $<;
    $) = $(;  # BUG: initgroups() not called
    exec 'program', 'arg1', 'arg2';
    die "can't exec program: $!";
}
```

For those even more concerned about safety, see the *Safe* and *Safe CGI* modules at a CPAN site near you. See the *perlmod* manpage for a list of CPAN sites.

NAME

perltrap − Perl traps for the unwary

DESCRIPTION

The biggest trap of all is forgetting to use the **−w** switch; see the *perlrun* manpage. 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 `$RS`) as though they were in **awk**; see the *perlvar* manpage for details.
- Semicolons are required after all simple statements in Perl (except at the end of a block). Newline is not a statement delimiter.
- Curly brackets are required on `if`s and `while`s.
- Variables begin with "$" 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.
- Associative array 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 *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 the *perlvar* manpage.
- $<*digit*> does not refer to fields—it refers to substrings matched by the last match pattern.
- The *print()* statement does not add field and record separators unless you set `$,` 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, since the third slash would be interpreted as a division operator—the tokener 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      $#
OFS       $,
ORS       $\
RLENGTH   length($&)
RS        $/
RSTART    length($`)
SUBSEP    $;
```

- You cannot set `$RS` to a pattern, only a string.
- When in doubt, run the **awk** construct through **a2p** and see what it gives you.

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 "$" or "@" in Perl.
- *printf()* does not implement the "*" format for interpolating field widths, but it's trivial to use interpolation of double-quoted strings to achieve the same effect.
- Comments begin with "#", not "/*".
- You can't take the address of anything, although a similar operator in Perl 5 is the backslash, which creates a reference.
- `ARGV` must be capitalized. `$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 backtick operator does variable interpretation without regard to the presence of single quotes in the command.
- The backtick operator does no translation of the return value, unlike **csh**.
- Shells (especially **csh**) do several levels of substitution on each command line. Perl does substitution only in certain constructs such as double quotes, backticks, angle brackets, and search patterns.
- Shells interpret scripts a little bit at a time. Perl compiles the entire program before executing it (except for `BEGIN` blocks, which execute at compile time).
- The arguments are available via `@ARGV`, not `$1`, `$2`, etc.
- The environment is not automatically made available as separate scalar variables.

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 the *perldata* manpage for details.
- Avoid barewords if you can, especially all lower-case ones. You can't tell just by looking at it whether a bareword is a function or a string. By using quotes on strings and parens 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 **only** be list operators, never unary ones.) See the *perlop* manpage.
- People have a hard time remembering that some functions default to `$_`, or `@ARGV`, or whatever, but that others which you might expect to do not.
- The <FH> construct is not the name of the filehandle, it is a readline operation on that handle. The data read is only assigned to `$_` 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 the *perlform* manpage 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.

Perl4 Traps

Penitent Perl 4 Programmers should take note of the following incompatible changes that occurred between release 4 and release 5:

- `@` now always interpolates an array in double-quotish strings. Some programs may now need to use backslash to protect any `@` that shouldn't interpolate.

- 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. For example:

  ```
  sub SeeYa { die "Hasta la vista, baby!" }
  $SIG{'QUIT'} = SeeYa;
  ```

  In Perl 4, that set the signal handler; in Perl 5, it actually calls the function! You may use the **−w** switch to find such places.

- Symbols starting with _ are no longer forced into package `main`, except for `$_` itself (and `@_`, etc.).

- Double-colon is now a valid package separator in an identifier. Thus these behave differently in perl4 vs. perl5:

  ```
  print "$a::$b::$c\n";
  print "$var::abc::xyz\n";
  ```

- `s'$lhs'$rhs'` now does no interpolation on either side. It used to interpolate `$lhs` but not `$rhs`.

- The second and third arguments of *splice()* are now evaluated in scalar context (as the book says) rather than list context.

- These are now semantic errors because of precedence:

  ```
  shift @list + 20;
  $n = keys %map + 20;
  ```

  Because if that were to work, then this couldn't:

  ```
  sleep $dormancy + 20;
  ```

- 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.

- `open FOO || die` is now incorrect. You need parens around the filehandle. While temporarily supported, using such a construct will generate a non-fatal (but non-suppressible) warning.

- The elements of argument lists for formats are now evaluated in list context. This means you can interpolate list values now.

- You can't do a `goto` into a block that is optimized away. Darn.

- 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.

- 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.
- `m//g` now attaches its state to the searched string rather than the regular expression.
- `reverse` is no longer allowed as the name of a sort subroutine.
- **taintperl** is no longer a separate executable. There is now a **–T** switch to turn on tainting when it isn't turned on automatically.
- Double-quoted strings may no longer end with an unescaped `$` or `@`.
- The archaic `while/if` BLOCK BLOCK syntax is no longer supported.
- Negative array subscripts now count from the end of the array.
- The comma operator in a scalar context is now guaranteed to give a scalar context to its arguments.
- The `**` operator now binds more tightly than unary minus. It was documented to work this way before, but didn't.
- Setting `$#array` lower now discards array elements.
- *delete()* is not guaranteed to return the old value for *tie()*d arrays, since this capability may be onerous for some modules to implement.
- The construct ''this is $$x'' used to interpolate the pid at that point, but now tries to dereference `$x`. `$$` by itself still works fine, however.
- Some error messages will be different.
- Some bugs may have been inadvertently 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 multiline 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 paren.
- Space after each comma.
- Long lines broken after an operator (except "and" and "or").
- Space after last paren 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 doen'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";
```

since 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 parens 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.
- 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 filesystems' 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 punctuational operators like `&&` and `||`. Call your subroutines as if they were functions or list operators to avoid excessive ampersands and parens.
- 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

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 the *perlxstut* manpage 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. Specifically, the *rpcb_gettime()* function will be 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 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 may not be needed.

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
```

In many of the examples shown here the CODE: block (and other blocks) will often be contained within braces ({ and }). This protects the CODE: block from complex INPUT typemaps and ensures the resulting C code is legal.

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
```

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
     CODE:
     {
     char *host = "localhost";

     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
     PPCODE:
     {
     time_t  timep;
     bool_t  status;
     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 simply return `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 `void` return type to disable the generation of the RETVAL variable 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.

```
void
rpcb_gettime(host)
     char *  host
     CODE:
     {
     time_t  timep;
     bool_t x;
     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.

```
void
rpcb_gettime(host)
     char *  host
     CODE:
     {
     time_t  timep;
     bool_t x;
     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
     PPCODE:
     {
     time_t  timep;
     if( rpcb_gettime( host, &timep ) )
          PUSHs(sv_2mortal(newSViv(timep)));
     else{
     /* Nothing pushed on stack, so an empty */
     /* list is implicitly returned. */
     }
     }
```

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 only compile with **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");
```

Inserting Comments and C Preprocessor Directives

Comments and C preprocessor directives are allowed within CODE:, PPCODE:, BOOT:, and CLEANUP: blocks. The compiler will pass the preprocessor directives through untouched and will remove the commented lines. Comments can be added to XSUBs by placing a `#` at the beginning of the line. Care should be taken to avoid making the comment look like a C preprocessor directive, lest it be interpreted as such.

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 colors {
     public:
     colors();
     ~colors();
     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 referen
            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

void
rpcb_gettime(host="localhost")
     char *host
     CODE:
     {
     time_t  timep;
     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";
```

AUTHOR

Dean Roehrich *<roehrich@cray.com>* Dec 10, 1995

NAME

perlXStut – Tutorial for XSUB's

DESCRIPTION

This tutorial will educate the reader on the steps involved in creating a Perl 5 extension. The reader is assumed to have access to the *perlguts* manpage and the *perlxs* manpage.

This tutorial starts with very simple examples and becomes more complex, bringing in more features that are available. Thus, certain statements towards the beginning may be incomplete. The reader is encouraged to read the entire document before lambasting the author about apparent mistakes.

This tutorial is still under construction. Constructive comments are welcome.

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 terminate.

Run `h2xs -A -n Test1`. This creates a directory named Test1, possibly under ext/ if it exists in the current working directory. Four files will be created in the Test1 dir: MANIFEST, Makefile.PL, Test1.pm, Test1.xs.

The MANIFEST file should contain the names of the four 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'      => 'Test1',
    'VERSION'   => '0.1',
    'LIBS'      => [''],   # e.g., '-lm'
    'DEFINE'    => '',     # e.g., '-DHAVE_SOMETHING'
    'INC'       => '',     # e.g., '-I/usr/include/other'
);
```

The file Test1.pm should look something like this:

```
package Test1;

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(

);
bootstrap Test1;

# Preloaded methods go here.

# Autoload methods go after __END__, and are processed by the autosplit prog

1;
__END__
```

And the Test1.xs file should look something like this:

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

MODULE = Test1          PACKAGE = Test1
```

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. It's output looks something like:

```
% perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Test1
%
```

Now, running make will produce output that looks something like this:

```
% make
mkdir ./blib
mkdir ./blib/auto
mkdir ./blib/auto/Test1
        perl xsubpp -typemap typemap Test1.xs >Test1.tc && mv Test1.tc Test1
        cc -c Test1.c
Running Mkbootstrap for Test1 ()
        chmod 644 Test1.bs
        LD_RUN_PATH="" ld -o ./blib/auto/Test1/Test1.sl -b Test1.o
        chmod 755 ./blib/auto/Test1/Test1.sl
        cp Test1.bs ./blib/auto/Test1/Test1.bs
        chmod 644 ./blib/auto/Test1/Test1.bs
        cp Test1.pm ./blib/Test1.pm
        chmod 644 ./blib/Test1.pm
```

Now we'll create a test script, test1.pl in the Test1 directory. It should look like this:

```
#! /usr/local/bin/perl

BEGIN { unshift(@INC, "./blib") }

use Test1;

Test1::hello();
```

Now we run the script and we should see the following output:

```
% perl test1.pl
Hello, world!
%
```

EXAMPLE 2

Now let's create a simple extension that will take a single argument and return 0 if the argument is even, 1 if the argument is odd.

Run `h2xs -A -n Test2`. This will create a Test2 directory with a file Test2.xs underneath it. Add the following to the end of the XS file:

```
int
is_even(input)
        int     input

        CODE:
        RETVAL = input % 2;

        OUTPUT:
        RETVAL
```

(Note that the line after the declaration of is_even is indented one tab stop. Although there is a tab between ''int'' and ''input'', this can be any amount of white space. Also notice that there is no semi-colon following the ''declaration'' of the variable input)

Now perform the same steps before, generating a Makefile from the Makefile.PL file, and running make.

Our test file test2.pl will now look like:

```
BEGIN { unshift(@INC, "./blib"); }

use Test2;

$a = &Test2::is_even(2);
$b = &Test2::is_even(3);

print "\$a is $a, \$b is $b\n";
```

The output should look like:

```
% perl test2.pl
$a is 0, $b is 1
%
```

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 tells Perl how to load your extension.

Generating the invoking the Makefile created a directory blib 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.

Finally, our test scripts do two important things. First of all, they place the directory ''blib'' at the head of the `@INC` array. Placing this inside a BEGIN block assures us that Perl will look in the blib directory hierarchy before looking in the system directories. This could be important if you are upgrading an already-existing extension and do not want to disturb the system version until you are ready to install it.

Second, the test scripts tell Perl to `use extension;`. When Perl sees this, it searches for a .pm file of the same name in the various directories kept in the @INC array. If it cannot be found, perl will die with an error that will look something like:

```
Can't locate Test2.pm in @INC at ./test2.pl line 5.
BEGIN failed--compilation aborted at ./test2.pl line 5.
```

The .pm file tells perl that it will need the Exporter and Dynamic Loader extensions. It then sets the @ISA array, which is used for looking up methods that might not exist in the current package, and finally tells perl to bootstrap the module. Perl will call its dynamic loader routine and load the shared library.

The @EXPORT array in the .pm file tells Perl which of the extension's routines should be placed into the calling package's namespace. In our two examples so far, we have not modified the @EXPORT array, so our test scripts must call the routines by their complete name (e.g., Test1::hello). If we placed the name of the routine in the @EXPORT array, so that the

```
@EXPORT = qw( hello );
```

Then the hello routine would also be callable from the "main" package. We could therefore change test1.pl to look like:

```
#! /usr/local/bin/perl

BEGIN { unshift(@INC, "./blib") }

use Test1;

hello();
```

And we would get the same output, "Hello, world!".

Most of the time you do not want to export the names of your extension's subroutines, because they might accidentally clash with other subroutines from other extensions or from the calling program itself.

EXAMPLE 3

Our third extension will take one argument as its input, round off that value, and set the argument to the rounded value.

Run `h2xs -A -n Test3`. This will create a Test3 directory with a file Test3.xs underneath it. Add the following to the end of the XS file:

```
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 file Makefile.PL so that the corresponding line looks like this:

```
'LIBS'       => ['-lm'],   # e.g., '-lm'
```

Generate the Makefile and run make. The test script test3.pl looks like:

```
#! /usr/local/bin/perl

BEGIN { unshift(@INC, "./blib"); }

use Test3;

foreach $i (-1.4, -0.5, 0.0, 0.4, 0.5) {
        $j = $i;
        &Test3::round($j);
        print "Rounding $i results in $j\n";
}

print STDERR "Trying to round a constant -- ";
&Test3::round(2.0);
```

Notice the output from trying to send a constant in to the routine. Perl reports:

```
Modification of a read-only value attempted at ./test3.pl line 15.
```

Perl won't let you change the value of two to, say, three, unlike a FORTRAN compiler from long, long ago!

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, in this case 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. The list of parameters looks very C–like, but the lines must be indented by a tab stop, and each line may not have an ending semi-colon.

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. Otherwise, you specify which variables used in the XSUB function should be placed into the respective Perl variables passed in.

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 the .c file created for the Test3 extension.

```
/*
 * This file was generated automatically by xsubpp version 1.9 from the
 * contents of Test3.xs. Don't edit this file, edit Test3.xs instead.
 *
 *      ANY CHANGES MADE HERE WILL BE LOST!
 *
 */
```

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"


XS(XS_Test3_round)
{
    dXSARGS;
    if (items != 1) {
        croak("Usage: Test3::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);
        }

        sv_setnv(ST(0), (double)arg); /* XXXXX */
    }
    XSRETURN(1);
}

XS(boot_Test3)
{
    dXSARGS;
    char* file = __FILE__;

    newXS("Test3::round", XS_Test3_round, file);
    ST(0) = &sv_yes;
    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 agood idea to write extensions that modify their input parameters, as in Example 3. However, in order to better accomodate calling pre-existing C routines, which often do modify their input parameters, this behavior is tolerated. The next example will show to do this.

EXAMPLE 4

We’ll now show how we can call routines in libraries, such as the curses screen handling package, or a DBM module like GDBM. Each of these libraries has a header file from which we will generate an XS template that we’ll then fine-tune.

Rather than attempt to find a library that exists on all systems, we’ll first create our own C library, then create an XSUB to it.

Let’s create the files libtest4.h and libtest4.c as follows:

```
/* libtest4.h */

#define TESTVAL 4

extern int      test4(int, long, const char*);

/* libtest4.c */

#include <stdlib.h>
#include "./libtest4.h"

int
test4(a, b, c)
int             a;
long            b;
const char *    c;
{
        return (a + b + atof(c) + TESTVAL);
}
```

Now let's compile it into a library. Since we'll be eventually using this archive to create a shared library, be sure to use the correct flags to generate position-independent code. In HP−UX, that's:

```
% cc −Aa −D_HPUX_SOURCE −c +z libtest4.c
% ar cr libtest4.a libtest4.o
```

Now let's move the libtest4.h and libtest.a files into a sub-directory under /tmp, so we don't interfere with anything.

```
% mkdir /tmp/test4
% mkdir /tmp/test4/include
% mkdir /tmp/test4/lib
% cp libtest4.h /tmp/test4/include
% cp libtest4.a /tmp/test4/lib
```

Okay, now that we have a header file and a library, let's begin actually writing the extension.

Run `h2xs −n Test4 /tmp/test4/include/libtest4.h` (notice we are no longer specifying **−A** as an argument). This will create a Test4 directory with a file *Test4.xs* underneath it. If we look at it now, we'll see some interesting things have been added to the various files.

- In the .xs file, there's now a #include declaration with the full path to the libtest4.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 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 only going to be used by the C routines themselves, and not by the user, they should be removed from the `@EXPORT` array. Alternately, if you don't mind using the "fully qualified name" of a variable, you could remove most or all of the items in the `@EXPORT` array.

Let's now add a definition for the routine in our library. Add the following code to the end of the .xs file:

```
int
test4(a,b,c)
int             a
long            b
const char *    c
```

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 we must tell our Makefile template where our new library is. Edit the Makefile.PL and change the following line:

```
'LIBS'      => ['-ltest4 -L/tmp/test4'],   # e.g., '-lm'
```

This specifies that we want the library test4 linked into our XSUB, and that it should also look in the directory /tmp/test4.

Let's also change the following line in the Makefile.PL to this:

```
'INC'       => '-I/tmp/test/include',     # e.g., '-I/usr/include/other'
```

and also change the #include in test4.xs to be:

```
#include <libtest4.h>
```

Now we don't have to specify the absolute path of the header file in the header files. This is generally considered a Good Thing.

Okay, let's create the Makefile, and run make. You can ignore a message that may look like:

```
Warning (non-fatal): No library found for -ltest4
```

If you forgot to create the typemap file, you might see output that looks like this:

```
Error: 'const char *' not in typemap in test4.xs, line 102
```

This error means that you have used a C datatype that xsubpp doesn't know how to convert between Perl and C. You'll have to create a typemap file to tell xsubpp how to do the conversions.

Author

Jeff Okamoto

Last Changed

1995/11/20

Jeff Okamoto *<okamoto@hpcc123.corp.hp.com>*

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.

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 large enough to hold either an integer or a pointer.

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.

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 not be terminated by a NUL.

If you simply 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.

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("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, 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 **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. Since 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 just use the `Sv*V` macros.

Working with AV's

There are two ways to create and load an AV. The first method just 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*); /* Returns highest index value in array */

SV**  av_fetch(AV*, I32 key, I32 lval);
        /* Fetches value at key offset, but it stores an undef value
           at the offset if lval is non-zero */
SV**  av_store(AV*, I32 key, SV* val);
        /* Stores val at offset key */
```

Take note that `av_fetch` and `av_store` return `SV**`'s, not `SV*`'s.

```
void  av_clear(AV*);
        /* Clear out all elements, but leave the array */
void  av_undef(AV*);
        /* Undefines the array, removing all elements */
void  av_extend(AV*, I32 key);
        /* Extend the array to a total of key elements */
```

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("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. 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.

Remember that `hv_store` and `hv_fetch` return `SV**`'s and not just `SV*`. In order 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);
```

And more miscellaneous functions:

```
void   hv_clear(HV*);
        /* Clears all entries in hash table */
void   hv_undef(HV*);
        /* Undefines the hash table */
```

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("varname", FALSE);
```

This returns NULL if the variable does not exist.

The hash algorithm, for those who are interested, is:

```
i = klen;
hash = 0;
s = key;
while (i--)
    hash = hash * 33 + *s++;
```

References

References are a special type of scalar that point to other data types (including references).

To create a reference, use the following command:

```
SV* newRV((SV*) thing);
```

The `thing` argument can be any of an `SV*`, `AV*`, or `HV*`. 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 actually discover what the reference refers to, you must use the following macro and then check the value returned.

```
SvTYPE(SvRV(SV*))
```

The most useful types that will be returned are:

```
SVt_IV    Scalar
SVt_NV    Scalar
SVt_PV    Scalar
SVt_PVAV  Array
SVt_PVHV  Hash
SVt_PVCV  Code
SVt_PVMG  Blessed Scalar
```

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 the *Stashes* manpage 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 pointer (*not a 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, which can be accessed from your Perl script, use the following routines, depending on the variable type.

```
SV*  perl_get_sv("varname", TRUE);
AV*  perl_get_av("varname", TRUE);
HV*  perl_get_hv("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 bits that may be OR'ed with the TRUE argument to enable certain extra features. Those bits are:

```
0x02  Marks the variable as multiply defined, thus preventing the
      "Indentifier <varname> used only once: possible typo" warning.
0x04  Issues a "Had to create <varname> unexpectedly" warning if
      the variable didn't actually exist.  This is useful if
      you expected the variable to already exist and want to propagate
      this warning back to the user.
```

If the `varname` argument does not contain a package specifier, it is created in the current package.

XSUB's 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 timezone'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.

For more information, consult the *perlxs* manpage.

Mortality

In Perl, values are normally "immortal" -- that is, they are not freed unless explicitly done so (via the Perl `undef` call or other routines in Perl itself).

Add cruft about reference counts. int SvREFCNT(SV* sv); void SvREFCNT_inc(SV* sv); void SvREFCNT_dec(SV* sv);

In the above example with `tzname`, we needed to create two new SV's to push onto the argument stack, that being the two strings. However, we don't want these new SV's to stick around forever because they will eventually be copied into the SV's that hold the two scalar variables.

An SV (or AV or HV) that is ''mortal'' acts in all ways as a normal ''immortal'' SV, AV, or HV, but is only valid in the ''current context''. When the Perl interpreter leaves the current context, the mortal SV, AV, or HV is automatically freed. Generally the ''current context'' means a single Perl statement.

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, 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 (and casting them to `SV*`) to the `sv_2mortal` or `sv_mortalcopy` routines.

>From Ilya: Beware that the *sv_2mortal()* call is eventually equivalent to *svREFCNT_dec()*. A value can happily be mortal in two different contexts, and it will be *svREFCNT_dec()*ed twice, once on exit from these contexts. It can also be mortal twice in the same context. This means that you should be very careful to make a value mortal exactly as many times as it is needed. The value that go to the Perl stack *should* be mortal.

You should be careful about creating mortal variables. It is possible for strange things to happen should you make the same value mortal within multiple contexts.

Stashes

A stash is a hash table (associative array) 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 called a GV (for 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
```

Perl stores various stashes in a separate GV structure (for global variable) but represents them with an HV structure. The keys in this larger GV are the various package names; the values are the `GV*`'s which are stashes. It may help to think of a stash purely as an HV, and that the term ''GV'' means the global variable hash.

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 return a blessed value to your Perl script, 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 the *perlref* manpage.

Magic

[This section still under construction. Ignore everything here. Post no bills. Everything not permitted is forbidden.]

Version 6, 1995/1/27

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 overriden, 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 magicalness
-------  ------              -------------------
\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?
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???
*        vtbl_glob           GV???
#        vtbl_arylen         Array Length
.        vtbl_pos            $. scalar variable
~        Reserved for extensions, but multiple extensions may clash
```

When an upper-case and lower-case letter both exist in the table, then the upper-case letter is used to represent some kind of composite type (a list or a hash), and the lower-case letter is used to represent an element of that composite type.

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 upper-case letter, then the mg_obj is copied to `nsv`, but the mg_type field is changed to be the lower-case letter.

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`.

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 more information, consult the *perlcall* manpage.

Memory Allocation

It is strongly 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 more quickly satisfy allocation requests. 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 (Larry has switched to using a run-time memory checker), so this argument can be any number.

The second argument `pointer` 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).

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
Creats 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.

```
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.

```
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.

```
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 the *perlxs* manpage.

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. 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. See `DBsingle`. The sub name can be found by

```
SvPV( GvSV( DBsub ), na )
```

dMARK Declare a stack marker for the XSUB. See `MARK` and `dORIGMARK`.

dORIGMARK
Saves the original stack mark for the XSUB. See `ORIGMARK`.

dSP Declares a stack pointer 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.

ENTER Opening bracket on a callback. See `LEAVE` and the *perlcall* manpage.

```
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 the *perlcall* manpage.

```
FREETMPS;
```

G_ARRAY
Used to indicate array context. See `GIMME` and the *perlcall* manpage.

G_DISCARD
Indicates that arguments returned from a callback should be discarded. See the *perlcall* manpage.

G_EVAL
Used to force a Perl `eval` wrapper around a callback. See the *perlcall* manpage.

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 the *perlcall* manpage.

G_SCALAR
Used to indicate scalar context. See `GIMME` and the *perlcall* manpage.

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.

he_free Releases a hash entry from an iterator. See `hv_iternext`.

hv_clear Clears a hash, making it empty.

```
void    hv_clear _((HV* tb));
```

hv_delete
Deletes a key/value pair in the hash. The value SV is removed from the hash and returned to the caller. The `lken` 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_exists
Returns a boolean indicating whether the specified hash key exists. The `lken` is the length of the key.

```
bool    hv_exists _((HV* tb, char* key, U32 klen));
```

hv_fetch Returns the SV which corresponds to the specified key in the hash. The `lken` 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_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_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_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 alphanumeric 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 the *perlxs* manpage.

LEAVE Closing bracket on a callback. See `ENTER` and the *perlcall* manpage.

```
LEAVE;
```

MARK Stack marker 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 refcount is set to 1.

```
AV*     newAV _((void));
```

newHV Creates a new HV. The refcount is set to 1.

```
HV*     newHV _((void));
```

newRV Creates an RV wrapper for an SV. The refcount for the original SV is incremented.

```
SV*     newRV _((SV* ref));
```

newSV Creates a new SV. The `len` parameter indicates the number of bytes of pre-allocated string space the SV should have. The refcount for the new SV is set to 1.

```
SV*     newSV _((STRLEN len));
```

newSViv
Creates a new SV and copies an integer into it. The refcount for the SV is set to 1.

```
SV*     newSViv _((IV i));
```

newSVnv
Creates a new SV and copies a double into it. The refcount for the SV is set to 1.

```
SV*     newSVnv _((NV i));
```

newSVpv
Creates a new SV and copies a string into it. The refcount 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 one. If `classname` is non-null then the new SV will be blessed in the specified package. The new SV is returned and its refcount is 1.

```
SV*     newSVrv _((SV* rv, char* classname));
```

newSVsv
Creates a new SV which is an exact duplicate of the orignal 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 the *perlembed* manpage.

perl_call_argv
Performs a callback to the specified Perl sub. See the *perlcall* manpage.

```
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 the *perlcall* manpage.

```
I32     perl_call_method _((char* methname, I32 flags));
```

perl_call_pv
Performs a callback to the specified Perl sub. See the *perlcall* manpage.

```
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 the *perlcall* manpage.

```
I32     perl_call_sv _((SV* sv, I32 flags));
```

perl_construct
Initializes a new Perl interpreter. See the *perlembed* manpage.

perl_destruct
Shuts down a Perl interpreter. See the *perlembed* manpage.

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 the *perlembed* manpage.

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 the *perlembed* manpage.

perl_require_pv
: Tells Perl to `require` a module.

```
void    perl_require_pv _((char* pv));
```

perl_run
: Tells a Perl interpreter to run. See the *perlembed* manpage.

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 the *perlcall* manpage.

```
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 the *perlcall* manpage 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 *perlxs* manpage.

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 the *perlcall* manpage.

```
SAVETMPS;
```

SP Stack pointer. This is usually handled by `xsubpp`. See `dSP` and `SPAGAIN`.

SPAGAIN
Refetch the stack pointer. Used after a callback. See the *perlcall* manpage.

```
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 refcount 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 Concatentates the string from SV `ssv` onto the end of the string in SV `dsv`.

```
void    sv_catsv _((SV* dsv, SV* ssv));
```

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 )
```

SvEND Returns a pointer to the last character in the string which is in the SV. See `SvCUR`. Access the character as

```
*SvEND(sv)
```

SvGROW
Expands the character buffer in the SV.

```
char * SvGROW( SV* sv, int len )
```

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)
```

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_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 refcount 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)
```

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)
```

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 refcount.

```
int SvREFCNT (SV* sv);
```

SvREFCNT_dec
Decrements the refcount of the given SV.

```
void SvREFCNT_dec (SV* sv)
```

SvREFCNT_inc
Increments the refcount 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 an SV, optionally blessing the SV. The SV must be an RV. 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 refcount of 1.

```
SV*     sv_setref_iv _((SV *rv, char *classname, IV iv));
```

sv_setref_nv
: Copies a double into an SV, optionally blessing the SV. The SV must be an RV. 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 refcount of 1.

```
SV*     sv_setref_nv _((SV *rv, char *classname, double nv));
```

sv_setref_pv
: Copies a pointer into an SV, optionally blessing the SV. The SV must be an RV. 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 refcount 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 an SV, optionally blessing the SV. The lenth of the string must be specified with `n`. The SV must be an RV. 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 refcount 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`.

```
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. See `svtype`.

sv_undef
This is the `undef` SV. Always refer to this as `&sv_undef`.

sv_usepvn
Tells an SV to use `ptr` to find its string value. Normally the string is stored inside the SV; this allows the SV to use an outside string. 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 the *perlxs* manpage.

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)
```

XSRETURN
: Return from XSUB, indicating number of items on the stack. This is usually handled by `xsubpp`.

```
XSRETURN(x);
```

XSRETURN_EMPTY
: Return from an XSUB immediately.

```
XSRETURN_EMPTY;
```

XSRETURN_NO
: Return `false` from an XSUB immediately.

```
XSRETURN_NO;
```

XSRETURN_UNDEF
: Return `undef` from an XSUB immediately.

```
XSRETURN_UNDEF;
```

XSRETURN_YES
: Return `true` from an XSUB immediately.

```
XSRETURN_YES;
```

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 );
```

AUTHOR

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, and Spider Boardman.

API Listing by Dean Roehrich <roehrich@cray.com>.

DATE

Version 20: 1995/12/14

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 your 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 the *perlembed* manpage.

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 – the *perlxs* manpage and the *perlguts* manpage.

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 actually 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 the section on *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 the *perlobj* manpage for more information on static and virtual methods and the section on *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 may 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 may items have been returned by the Perl subroutine.

If 0, the 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 explicitly dispose of these temporaries 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.

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. Hopefully 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 actually 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 explicitly use the `dSP` macro – 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 automatically count the number of parameters you are pushing. 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. Since 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 section the section on *XSUB'S and the Argument Stack* in the *perlguts* manpage 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*, which takes 2 integer parameters and simply returns their sum.

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

Since 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 re-allocated 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 ever happen.

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 ;
    SV * sv ;

    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 */
    sv = GvSV(gv_fetchpv("@", TRUE, SVt_PV));
    if (SvTRUE(sv))
    {
        printf ("Uh oh - %s\n", SvPV(sv, 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

   ```
   sv = GvSV(gv_fetchpv("@", TRUE, SVt_PV));
   if (SvTRUE(sv))
   {
       printf ("Uh oh - %s\n", SvPVx(sv, na)) ;
       POPs ;
   }
   ```

   is the direct equivalent of this bit of Perl

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

3. Note that the stack is popped using `POPs` in the block where `SvTRUE(sv)` 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*. Since we want the program to continue after detecting this error, it is essential that the stack is tidied up by removing the *undef*.

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 only as 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) ;
```

Since 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 to just 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 `CallSaved–Sub1`, 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 dependant on the version of Perl you are using)

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

The variable `$ref` may have referred to the subroutine `fred` whenever the call to `SaveSub1` was made but by the time `CallSavedSub1` gets called it now holds the number `47`. Since we saved only a pointer to the original SV in `SaveSub1`, any changes to `$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) ;
```

In order 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 just implements 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`, simply prints out 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 actually 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 explicitly dispose 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 1 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)
char *      buffer ;
{
    ...
}
```

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

In this case a possible way around this problem is to 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 recompiling. None the less, as long as the number of functions is chosen with some care, it is still a workable solution and in some cases is the only one available.

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

1. Ignore the problem – Allow only 1 callback
   For a lot of situations, like interfacing to an error handler, this may be a perfectly adequate solution.

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

3. Use a parameter to map to the Perl callback
   A hash is an ideal mechanism to store the mapping between C and Perl.

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 the *perlxs* manpage 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

the *perlxs* manpage, the *perlguts* manpage, the *perlembed* manpage

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 and Larry Wall.

DATE

Version 1.1, 17th May 1995

NAME

perlembed – how to embed perl in your C program

DESCRIPTION

PREAMBLE

Do you want to:

Use C from Perl?
Read the *perlcall* manpage and the *perlxs* manpage.

Use a UNIX program from Perl?
Read about backquotes and the `system` entry in the *perlfunc* manpage and the `exec` entry in the *perlfunc* manpage.

Use Perl from Perl?
Read about the `do` entry in the *perlfunc* manpage and the `eval` entry in the *perlfunc* manpage and the `use` entry in the *perlmod* manpage and the `require` entry in the *perlmod* manpage.

Use C from C?
Rethink your design.

Use Perl from C?
Read on...

ROADMAP

the section on *Compiling your C program*

There's one example in each of the five sections:

the section on *Adding a Perl interpreter to your C program*

the section on *Calling a Perl subroutine from your C program*

the section on *Evaluating a Perl statement from your C program*

the section on *Performing Perl pattern matches and substitutions from your C program*

the section on *Fiddling with the Perl stack from your C program*

This documentation is UNIX specific.

Compiling your C program

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.)

Your C program will—usually—allocate, "run", and deallocate a *PerlInterpreter* object, which is defined in the perl library.

If your copy of Perl is recent enough to contain this documentation (5.002 or later), then the perl library (and *EXTERN.h* and *perl.h*, which you'll also need) will reside in a directory resembling 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 -e 'use Config; print $Config{archlib}'
```

Here's how you might compile the example in the next section, the section on *Adding a Perl interpreter to your C program*, on a DEC Alpha running the OSF operating system:

```
% cc -o interp interp.c -L/usr/local/lib/perl5/alpha-dec_osf/CORE
-I/usr/local/lib/perl5/alpha-dec_osf/CORE -lperl -lm
```

You'll have to choose the appropriate compiler (*cc*, *gcc*, et al.) and 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* or *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 -e 'use Config; print $Config{libs}'
```

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 <stdio.h>
#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, env);
    perl_run(my_perl);
    perl_destruct(my_perl);
    perl_free(my_perl);
}
```

Now compile this program (I'll call it *interp.c*) into an executable:

```
% cc -o interp interp.c -L/usr/local/lib/perl5/alpha-dec_osf/CORE
-I/usr/local/lib/perl5/alpha-dec_osf/CORE -lperl -lm
```

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'll need to remove the call to *perl_run()* and replace it with a call to *perl_call_argv()*.

That's shown below, in a program I'll call *showtime.c*.

```
#include <stdio.h>
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

int main(int argc, char **argv, char **env)
{
    my_perl = perl_alloc();
    perl_construct(my_perl);

    perl_parse(my_perl, NULL, argc, argv, env);

                                  /*** This replaces perl_run() ***/
    perl_call_argv("showtime", G_DISCARD | G_NOARGS, argv);
    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 the *perlcall* manpage.

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 -L/usr/local/lib/perl5/alpha-dec_osf/CORE
-I/usr/local/lib/perl5/alpha-dec_osf/CORE -lperl -lm

% 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.

If you want to pass some arguments to the Perl subroutine, or you want to access the return value, you'll need to manipulate the Perl stack, demonstrated in the last section of this document: the section on *Fiddling with the Perl stack from your C program*

Evaluating a Perl statement from your C program

NOTE: This section, and the next, employ some very brittle techniques for evaluting strings of Perl code. Perl 5.002 contains some nifty features that enable A Better Way (such as with the `perl_eval_sv` entry in the *perlguts* manpage). Look for updates to this document soon.

One way to evaluate a Perl string is to define a function (we'll call ours *perl_eval()*) that wraps around Perl's the `eval` entry in the *perlfunc* manpage.

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 use the `require` entry in the *perlmod* manpage or the `do` entry in the *perlfunc* manpage 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 <stdio.h>
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

int perl_eval(char *string)
{
  char *argv[2];
  argv[0] = string;
  argv[1] = NULL;
  perl_call_argv("_eval_", 0, argv);
}

main (int argc, char **argv, char **env)
{
  char *embedding[] = { "", "-e", "sub _eval_ { eval $_[0] }" };
  STRLEN length;

  my_perl = perl_alloc();
  perl_construct( my_perl );

  perl_parse(my_perl, NULL, 3, embedding, env);

                                  /** 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 the *perlguts* manpage.

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 program might be represented as `"/\\b\\w*\\b/"`), 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]"), 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, 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:

```
#include <stdio.h>
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

int eval(char *string)
{
  char *argv[2];
  argv[0] = string;
  argv[1] = NULL;
  perl_call_argv("_eval_", 0, argv);
}

/** 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 **matches[])
{
  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 **/
  *matches = (char **) malloc(sizeof(char *) * num_matches);
  for (i = 0; i <= num_matches; i++) {
    current_match = av_shift(array);
    (*matches)[i] = SvPV(current_match, length);
  }
  return num_matches;
}

main (int argc, char **argv, char **env)
{
  char *embedding[] = { "", "-e", "sub _eval_ { eval $_[0] }" };
  char *text, **matches;
  int num_matches, i;
  int j;

  my_perl = perl_alloc();
  perl_construct( my_perl );

  perl_parse(my_perl, NULL, 3, embedding, env);

  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 so

  if (perl_match(text, "m/quarter/")) /** Does text contain 'quarter'? **/
    printf("perl_match: Text contains the word 'quarter'.\n\n");
  else
    printf("perl_match: Text doesn't contain the word 'quarter'.\n\n");

  if (perl_match(text, "m/eighth/")) /** Does text contain 'eighth'? **/
    printf("perl_match: Text contains the word 'eighth'.\n\n");
  else
    printf("perl_match: Text doesn't contain the word 'eighth'.\n\n");

                                    /** Match all occurrences of /wi../ **/
```

```
     num_matches = perl_matches(text, "m/(wi..)/g", &matches);

     printf("perl_matches: m/(wi..)/g found %d matches...\n", num_matches);
     for (i = 0; i < num_matches; i++)
       printf("match: %s\n", matches[i]);
     printf("\n");
     for (i = 0; i < num_matches; i++) {
       free(matches[i]);
     }
     free(matches);

                                  /** Remove all vowels from text **/
     num_matches = perl_substitute(&text, "s/[aeiou]//gi");
     if (num_matches) {
       printf("perl_substitute: s/[aeiou]//gi...%d substitutions made.\n",
        num_matches);
       printf("Now text is: %s\n\n", text);
     }

                                  /** Attempt a substitution
     if (!perl_substitute(&text, "s/Perl/C/")) {
       printf("perl_substitute: s/Perl/C...No substitution made.\n\n");
     }

     free(text);

     perl_destruct(my_perl);
     perl_free(my_perl);
   }
```

which produces the output

```
perl_match: Text contains the word 'quarter'.

perl_match: Text doesn't contain the word 'eighth'.

perl_matches: m/(wi..)/g found 2 matches...
match: will
match: with

perl_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

perl_substitute: s/Perl/C...No substitution made.
```

```
=head2 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 the *perlguts* manpage.

Then you'll need to know how to manipulate the Perl stack. That's described in the *perlcall* manpage.

Once you've understood those, embedding Perl in C is easy.

Since 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, since 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 <stdio.h>
#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, env);

  PerlPower(3, 4);                      /*** Compute 3 ** 4 ***/

  perl_destruct(my_perl);
  perl_free(my_perl);
}
```

Compile and run:

```
% cc -o power power.c -L/usr/local/lib/perl5/alpha-dec_osf/CORE
-I/usr/local/lib/perl5/alpha-dec_osf/CORE -lperl -lm

% power
3 to the 4th power is 81.
```

MORAL

You can sometimes *write faster code* in C, but you can always *write code faster* in Perl. Since you can use each from the other, combine them as you wish.

AUTHOR

Jon Orwant <*orwant@media.mit.edu*>, with contributions from Tim Bunce, Tom Christiansen, Dov Grobgeld, and Ilya Zakharevich.

December 18, 1995

Some of this material is excerpted from my book: *Perl 5 Interactive*, Waite Group Press, 1996 (ISBN 1-57169-064-6) and appears courtesy of Waite Group Press.

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
```

The "=pod" directive does nothing beyond telling the compiler to lay off of 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 on the same paragraph as "=headn" 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 indention. 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 you the first =item type to decide how to format the list.

And don't forget, when using any command, that that 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 it's 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>            manpage
                L<name/ident>      item in manpage
                L<name/"sec">      section in other manpage
                L<"sec">           section in this manpage
                                   (the quotes are optional)
                L</"sec">          ditto
F<file>     Used for filenames
X<index>    An index entry
Z<>         A zero-width character
```

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 beg, and end it with an =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.

SEE ALSO

the *pod2man* manpage and the section on *PODs: Embedded Documentation* in the *perlsyn* manpage

AUTHOR

Larry Wall

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.

See the section on *Pragmatic Modules* in the *perlmod* manpage and the `subs` entry in the *strict* manpage.

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 the *perlref* manpage).

```
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 the `my` entry in the *perlfunc* manpage and the `local` entry in the *perlfunc* manpage.

```
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 the section on *Pragmatic Modules* in the *perlmod* manpage.

NAME

sigtrap – Perl pragma to enable stack backtrace on unexpected signals

SYNOPSIS

```
use sigtrap;
use sigtrap qw(BUS SEGV PIPE SYS ABRT TRAP);
```

DESCRIPTION

The `sigtrap` pragma initializes some default signal handlers that print a stack dump of your Perl program, then sends itself a SIGABRT. This provides a nice starting point if something horrible goes wrong.

By default, handlers are installed for the ABRT, BUS, EMT, FPE, ILL, PIPE, QUIT, SEGV, SYS, TERM, and TRAP signals.

See the section on *Pragmatic Modules* in the *perlmod* manpage.

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. 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 the section on *MAGIC AUTOGENERATION*. (Mathemagical methods refer to methods triggered by an overloaded mathematical operator.)

Calling Conventions for Binary Operations

The functions specified in the `use overload ...` directive are called with three (in one particular case with four, see the section on *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.

• *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 the section on *SPECIAL SYMBOLS FOR* `use overload`.

See the section on *Fallback* for an explanation of when a missing method can be autogenerated.

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 the section on *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.

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`,

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.

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 as magic in the symbol table hash for the package. The table is rechecked for changes due to `use overload`, `no overload`, and `@ISA` only during `bless`ing; so if they are changed 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.)

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 and the current operation is overloadable 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.

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 the section on *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.

BUGS

Because it is used for overloading, the per-package associative array `%OVERLOAD` now has a special meaning in Perl.

As shipped, mathemagical properties are not inherited via the `@ISA` tree.

This document is confusing.

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

AddINC – optional module which deals with paths relative to the source file.

AUTHOR

Tim Bunce, 2nd June 1995.

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

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 the section on *Pragmatic Modules* in the *perlmod* manpage.

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 wtih the more explicative and endearing descriptions found in the *perldiag* manpage. Like the other pragmata, it affects to 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 the *perldiag* manpage introduction before any other diagnostics. The `$diagnostics::PRETTY` can generate nicer escape sequences for pgers.

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 theorectical

```
(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 `$SIG{__WARN__}` handler is discovered, it will continue to be honored, but only after the *diagnostic::splainthis()* function (the module's `$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 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

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 the *DB_File* manpage), 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 the *DB_File* manpage. 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 = (Exporter, AutoLoader);
```

DESCRIPTION

This module tells its users that functions in the FOOBAR package are to be autoloaded from *auto/$AUTOLOAD.al*. See the section on *Autoloading* in the *perlsub* manpage.

NAME

AutoSplit – split a package for autoloading

SYNOPSIS

```
 perl -e 'use AutoSplit; autosplit_modules(@ARGV)'  ...
```

DESCRIPTION

This function will split up your program into files that the AutoLoader module can handle. Normally only used to build autoloading Perl library modules, especially extensions (like POSIX). You should look at how they're built out for details.

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, $t0);
print "the code took:",timestr($dt),"\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

Config − access Perl configuration option

SYNOPSIS

```
use Config;
if ($Config{'cc'} =~ /gcc/) {
    print "built by gcc\n";
}
```

DESCRIPTION

The Config module contains everything that was available to the `Configure` program at Perl build time. Shell variables from *config.sh* are stored in the readonly-variable `%Config`, indexed by their names.

EXAMPLE

Here's a more sophisticated example of using `%Config`:

```
use Config;

defined $Config{sig_name} || die "No sigs?";
foreach $name (split(' ', $Config{sig_name})) {
    $signo{$name} = $i;
    $signame[$i] = $name;
    $i++;
}

print "signal #17 = $signame[17]\n";
if ($signo{ALRM}) {
    print "SIGALRM is $signo{ALRM}\n";
}
```

NOTE

This module contains a good example of how to make a variable readonly to those outside of it.

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 the section on *Overriding builtin functions* in the *perlsub* manpage.) Note that it will only be kept up to date it all packages which use chdir import it from Cwd.

NAME

DB_File − Perl5 access to Berkeley DB

SYNOPSIS

```
use DB_File ;

[$X =] tie %hash,  DB_File, $filename [, $flags, $mode, $DB_HASH] ;
[$X =] tie %hash,  DB_File, $filename, $flags, $mode, $DB_BTREE ;
[$X =] tie @array, DB_File, $filename, $flags, $mode, $DB_RECNO ;

$status = $X->del($key [, $flags]) ;
$status = $X->put($key, $value [, $flags]) ;
$status = $X->get($key, $value [, $flags]) ;
$status = $X->seq($key, $value [, $flags]) ;
$status = $X->sync([$flags]) ;
$status = $X->fd ;

untie %hash ;
untie @array ;
```

DESCRIPTION

DB_File is a module which allows Perl programs to make use of the facilities provided by Berkeley DB. If you intend to use this module you should really have a copy of the Berkeley DB manualpage at hand. The interface defined here mirrors the Berkeley DB interface closely.

Berkeley DB is a C library which provides a consistent interface to a number of database formats. **DB_File** provides an interface to all three of the database types currently supported by Berkeley DB.

The file types are:

DB_HASH
: This database type allows arbitrary key/data pairs to be stored in data files. This is equivalent to the functionality provided by other hashing packages like DBM, NDBM, ODBM, GDBM, and SDBM. Remember though, the files created using DB_HASH are not compatible with any of the other packages mentioned.

  A default hashing algorithm, which will be adequate for most applications, is built into Berkeley DB. If you do need to use your own hashing algorithm it is possible to write your own in Perl and have **DB_File** use it instead.

DB_BTREE
: The btree format allows arbitrary key/data pairs to be stored in a sorted, balanced binary tree.

  As with the DB_HASH format, it is possible to provide a user defined Perl routine to perform the comparison of keys. By default, though, the keys are stored in lexical order.

DB_RECNO
: DB_RECNO allows both fixed-length and variable-length flat text files to be manipulated using the same key/value pair interface as in DB_HASH and DB_BTREE. In this case the key will consist of a record (line) number.

How does DB_File interface to Berkeley DB?

DB_File allows access to Berkeley DB files using the *tie()* mechanism in Perl 5 (for full details, see the `tie()` entry in the *perlfunc* manpage). This facility allows **DB_File** to access Berkeley DB files using either an associative array (for DB_HASH & DB_BTREE file types) or an ordinary array (for the DB_RECNO file type).

In addition to the *tie()* interface, it is also possible to use most of the functions provided in the Berkeley DB API.

Differences with Berkeley DB

Berkeley DB uses the function *dbopen()* to open or create a database. Below is the C prototype for *dbopen()*.

```
DB*
dbopen (const char * file, int flags, int mode,
        DBTYPE type, const void * openinfo)
```

The parameter `type` is an enumeration which specifies which of the 3 interface methods (DB_HASH, DB_BTREE or DB_RECNO) is to be used. Depending on which of these is actually chosen, the final parameter, *openinfo* points to a data structure which allows tailoring of the specific interface method.

This interface is handled slightly differently in **DB_File**. Here is an equivalent call using **DB_File**.

```
tie %array, DB_File, $filename, $flags, $mode, $DB_HASH ;
```

The `filename`, `flags` and `mode` parameters are the direct equivalent of their *dbopen()* counterparts. The final parameter `$DB_HASH` performs the function of both the `type` and `openinfo` parameters in *dbopen()*.

In the example above `$DB_HASH` is actually a reference to a hash object. **DB_File** has three of these pre-defined references. Apart from `$DB_HASH`, there is also `$DB_BTREE` and `$DB_RECNO`.

The keys allowed in each of these pre-defined references is limited to the names used in the equivalent C structure. So, for example, the `$DB_HASH` reference will only allow keys called `bsize`, `cachesize`, `ffactor`, `hash`, `lorder` and `nelem`.

To change one of these elements, just assign to it like this

```
$DB_HASH{cachesize} = 10000 ;
```

RECNO

In order to make RECNO more compatible with Perl the array offset for all RECNO arrays begins at 0 rather than 1 as in Berkeley DB.

In Memory Databases

Berkeley DB allows the creation of in-memory databases by using NULL (that is, a C<(char *)0 in C) in place of the filename. **DB_File** uses `undef` instead of NULL to provide this functionality.

Using the Berkeley DB Interface Directly

As well as accessing Berkeley DB using a tied hash or array, it is also possible to make direct use of most of the functions defined in the Berkeley DB documentation.

To do this you need to remember the return value from the tie.

```
$db = tie %hash, DB_File, "filename"
```

Once you have done that, you can access the Berkeley DB API functions directly.

```
$db->put($key, $value, R_NOOVERWRITE) ;
```

All the functions defined in the *dbx(3X)* manpage are available except for *close()* and *dbopen()* itself. The **DB_File** interface to these functions have been implemented to mirror the the way Berkeley DB works. In particular note that all the functions return only a status value. Whenever a Berkeley DB function returns data via one of its parameters, the **DB_File** equivalent does exactly the same.

All the constants defined in the *dbopen* manpage are also available.

Below is a list of the functions available.

get Same as in `recno` except that the flags parameter is optional. Remember the value associated with the key you request is returned in the `$value` parameter.

put As usual the flags parameter is optional.

If you use either the R_IAFTER or R_IBEFORE flags, the key parameter will have the record number of the inserted key/value pair set.

del The flags parameter is optional.

fd As in *recno*.

seq The flags parameter is optional.

Both the key and value parameters will be set.

sync The flags parameter is optional.

EXAMPLES

It is always a lot easier to understand something when you see a real example. So here are a few.

Using HASH

```
use DB_File ;
use Fcntl ;

tie %h,  "DB_File", "hashed", O_RDWR|O_CREAT, 0640, $DB_HASH ;

# Add a key/value pair to the file
$h{"apple"} = "orange" ;

# Check for existence of a key
print "Exists\n" if $h{"banana"} ;

# Delete
delete $h{"apple"} ;

untie %h ;
```

Using BTREE

Here is sample of code which used BTREE. Just to make life more interesting the default comparision function will not be used. Instead a Perl sub, `Compare()`, will be used to do a case insensitive comparison.

```
use DB_File ;
use Fcntl ;

sub Compare
{
    my ($key1, $key2) = @_ ;

    "\L$key1" cmp "\L$key2" ;
}

$DB_BTREE->{compare} = 'Compare' ;

tie %h,  'DB_File', "tree", O_RDWR|O_CREAT, 0640, $DB_BTREE ;

# Add a key/value pair to the file
$h{'Wall'} = 'Larry' ;
$h{'Smith'} = 'John' ;
$h{'mouse'} = 'mickey' ;
$h{'duck'}   = 'donald' ;

# Delete
delete $h{"duck"} ;

# Cycle through the keys printing them in order.
# Note it is not necessary to sort the keys as
# the btree will have kept them in order automatically.
foreach (keys %h)
  { print "$_\n" }

untie %h ;
```

Here is the output from the code above.

```
mouse
Smith
Wall
```

Using RECNO

```
use DB_File ;
use Fcntl ;

$DB_RECNO->{psize} = 3000 ;

tie @h,  DB_File, "text", O_RDWR|O_CREAT, 0640, $DB_RECNO ;

# Add a key/value pair to the file
$h[0] = "orange" ;

# Check for existence of a key
print "Exists\n" if $h[1] ;

untie @h ;
```

Locking Databases

Concurrent access of a read-write database by several parties requires them all to use some kind of locking. Here's an example of Tom's that uses the *fd* method to get the file descriptor, and then a careful *open()* to give something Perl will *flock()* for you. Run this repeatedly in the background to watch the locks granted in proper order.

```
use Fcntl;
use DB_File;

use strict;

sub LOCK_SH { 1 }
sub LOCK_EX { 2 }
sub LOCK_NB { 4 }
sub LOCK_UN { 8 }

my($oldval, $fd, $db, %db, $value, $key);

$key = shift || 'default';
$value = shift || 'magic';

$value .= " $$";

$db = tie(%db, 'DB_File', '/tmp/foo.db', O_CREAT|O_RDWR, 0644)
        || die "dbcreat /tmp/foo.db $!";
$fd = $db->fd;
print "$$: db fd is $fd\n";
open(DB_FH, "+<&=$fd") || die "dup $!";

unless (flock (DB_FH, LOCK_SH | LOCK_NB)) {
    print "$$: CONTENTION; can't read during write update!
                Waiting for read lock ($!) ....";
    unless (flock (DB_FH, LOCK_SH)) { die "flock: $!" }
}
print "$$: Read lock granted\n";

$oldval = $db{$key};
print "$$: Old value was $oldval\n";
flock(DB_FH, LOCK_UN);

unless (flock (DB_FH, LOCK_EX | LOCK_NB)) {
    print "$$: CONTENTION; must have exclusive lock!
                Waiting for write lock ($!) ....";
    unless (flock (DB_FH, LOCK_EX)) { die "flock: $!" }
}

print "$$: Write lock granted\n";
$db{$key} = $value;
sleep 10;

flock(DB_FH, LOCK_UN);
untie %db;
close(DB_FH);
print "$$: Updated db to $key=$value\n";
```

HISTORY

0.1 First Release.

0.2 When **DB_File** is opening a database file it no longer terminates the process if *dbopen* returned an error. This allows file protection errors to be caught at run time. Thanks to Judith Grass <grass@cybercash.com> for spotting the bug.

0.3 Added prototype support for multiple btree compare callbacks.

1.0 **DB_File** has been in use for over a year. To reflect that, the version number has been incremented to 1.0.

Added complete support for multiple concurrent callbacks.

Using the *push* method on an empty list didn't work properly. This has been fixed.

1.01 Fixed a core dump problem with SunOS.

The return value from TIEHASH wasn't set to NULL when dbopen returned an error.

WARNINGS

If you happen find any other functions defined in the source for this module that have not been mentioned in this document -- beware. I may drop them at a moments notice.

If you cannot find any, then either you didn't look very hard or the moment has passed and I have dropped them.

BUGS

Some older versions of Berkeley DB had problems with fixed length records using the RECNO file format. The newest version at the time of writing was 1.85 – this seems to have fixed the problems with RECNO.

I am sure there are bugs in the code. If you do find any, or can suggest any enhancements, I would welcome your comments.

AVAILABILITY

Berkeley DB is available at your nearest CPAN archive (see the section on *CPAN* in the *perlmod* manpage for a list) in *src/misc/db.1.85.tar.gz*, or via the host *ftp.cs.berkeley.edu* in */ucb/4bsd/db.tar.gz*. It is *not* under the GPL.

SEE ALSO

the *perl(1)* manpage, the *dbopen(3)* manpage, the *hash(3)* manpage, the *recno(3)* manpage, the *btree(3)* manpage

Berkeley DB is available from *ftp.cs.berkeley.edu* in the directory */ucb/4bsd*.

AUTHOR

The DB_File interface was written by Paul Marquess <pmarquess@bfsec.bt.co.uk>. Questions about the DB system itself may be addressed to Keith Bostic <bostic@cs.berkeley.edu>.

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

DynaLoader – Dynamically load C libraries into Perl code

dl_error(), *dl_findfile()*, *dl_expandspec()*, *dl_load_file()*, *dl_find_symbol()*, *dl_undef_symbols()*, *dl_install_xsub()*, *boostrap()* – routines used by DynaLoader modules

SYNOPSIS

```
package YourPackage;
require DynaLoader;
@ISA = qw(... DynaLoader ...);
bootstrap YourPackage;
```

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
$dl_debug
                                                     Implemented in:
bootstrap($modulename)                                   Perl
@filepaths = dl_findfile(@names)                         Perl

$libref  = dl_load_file($filename)                       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_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)
```

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.

(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 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])
```

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_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.

boostrap()

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_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.

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 the *perlvar* manpage 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-associative-array 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 − provide inport/export controls for Perl modules

SYNOPSIS

use Module; use Module qw(name1 name2 :tag /pattern/ !name);

DESCRIPTION

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.
```

Application says:

```
use Module qw(:DEFAULT :T2 !B3 A3);
use Socket qw(!/^[AP]F_/ !SOMAXCONN !SOL_SOCKET);
use POSIX  qw(/^S_/ acos asin atan /^E/ !/^EXIT/);
```

You can set `$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.

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.

SEE ALSO

the *ExtUtils::MakeMaker* manpage

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.

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'];
```

What's new in version 5 of MakeMaker

MakeMaker 5 is pure object oriented. This allows us to write an unlimited number of Makefiles with a single perl process. 'perl Makefile.PL' with MakeMaker 5 goes through all subdirectories immediately and evaluates any Makefile.PL found in the next level subdirectories. The benefit of this approach comes in useful for both single and multi directories extensions.

Multi directory extensions have an immediately visible speed advantage, because there's no startup penalty for any single subdirectory Makefile.

Single directory packages benefit from the much improved *needs_linking()* method. As the main Makefile knows everything about the subdirectories, a *needs_linking()* method can now query all subdirectories if there is any linking involved down in the tree. The speedup for PM−only Makefiles seems to be around 1 second on my Indy 100 MHz.

Incompatibilities between MakeMaker 5.00 and 4.23

There are no incompatibilities in the short term, as all changes are accompanied by short-term workarounds that guarantee full backwards compatibility.

You are likely to face a few warnings that expose deprecations which will result in incompatibilities in the long run:

You should not use `%att` directly anymore. Instead any subroutine you override in the MY package will be called by the object method, so you can access all object attributes directly via the object in `$_[0]`.

You should not call the class methos MM−>something anymore. Instead you should call the superclass. Something like

```
sub MY::constants {
    my $self = shift;
    $self->MM::constants();
}
```

Especially the *libscan()* and *exescan()* methods should be altered towards OO programming, that means do not expect that $_ to contain the path but rather $_[1].

You should program with more care. Watch out for any MakeMaker variables. Do not try to alter them, somebody else might depend on them. E.g. do not overwrite the ExtUtils::MakeMaker::VERSION variable (this happens if you import it and then set it to the version number of your package), do not expect that the INST_LIB variable will be “blib/FindBin.pm"). Do not croak in your Makefile.PL, let it fail with a warning instead.

Try to build several extensions simultanously to debug your Makefile.PL. You can unpack a bunch of distributed packages, so your directory looks like

```
Alias-1.00/          Net-FTP-1.01a/       Set-Scalar-0.001/
ExtUtils-Peek-0.4/   Net-Ping-1.00/       SetDualVar-1.0/
Filter-1.06/         NetTools-1.01a/      Storable-0.1/
GD-1.00/             Religion-1.04/       Sys-Domain-1.05/
MailTools-1.03/      SNMP-1.5b/           Term-ReadLine-0.7/
```

and write a dummy Makefile.PL that contains nothing but

```
use ExtUtils::MakeMaker;
WriteMakefile();
```

That’s actually fun to watch :)

Final suggestion: Try to delete all of your MY:: subroutines and watch, if you really still need them. MakeMaker might already do what you want without them. That’s all about it.

Default Makefile Behaviour

The automatically 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`. If the user wants to work with a different perl than the default, this can be achieved with

```
perl Makefile.PL PERL=/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 dist       # see below the Distribution Support section
```

Special case make install

make alone puts all relevant files into directories that are named by the macros INST_LIB, INST_ARCHLIB, INST_EXE, INST_MAN1DIR, and INST_MAN3DIR. All these default to ./blib or 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_EXE is not defined.

The *install* target of the generated Makefile is a recursive call to make which sets

```
INST_LIB     to INSTALLPRIVLIB
INST_ARCHLIB to INSTALLARCHLIB
INST_EXE     to INSTALLBIN
INST_MAN1DIR to INSTALLMAN1DIR
INST_MAN3DIR to INSTALLMAN3DIR
```

The INSTALL... macros in turn default to their `%Config` ($Config{installprivlib}, `$Config`{installarchlib}, etc.) counterparts.

The recommended way to proceed is to set only the INSTALL* macros, not the INST_* targets. In doing so, you give room to the compilation process without affecting important directories. Usually a make test will succeed after the make, and a make install can finish the game.

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.

The usual relationship between INSTALLPRIVLIB and INSTALLARCHLIB is that the latter is a subdirectory of the former with the name `$Config{archname}`, MakeMaker supports the user who sets INSTALLPRIVLIB. If INSTALLPRIVLIB is set, but INSTALLARCHLIB not, then MakeMaker defaults the latter to be INSTALLPRIVLIB/ARCHNAME if that directory exists, otherwise it defaults to INSTALLPRIVLIB.

PREFIX attribute

The PREFIX attribute can be used to set the INSTALL* attributes (except INSTALLMAN1DIR) in one go. The quickest way to install a module in a non-standard place

```
perl Makefile.PL PREFIX=~
```

is identical to

```
perl Makefile.PL INSTALLPRIVLIB=~/perl5/lib INSTALLBIN=~/bin \
                 INSTALLMAN3DIR=~/perl5/man/man3
```

Note, that the tilde expansion is done by MakeMaker, not by perl by default, nor by make.

It is important to know, that the INSTALL* macros should be absolute paths, never relativ ones. Packages with multiple Makefile.PLs in different directories get the contents of the INSTALL* macros propagated verbatim. (The INST_* macros will be corrected, if they are relativ paths, but not the INSTALL* macros.)

If the user has superuser privileges, and is not working on AFS (Andrew File System) or relatives, then the defaults for INSTALLPRIVLIB, INSTALLARCHLIB, INSTALLBIN, 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 is an experimental feature. It 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 INSTALLPRIVLIB=/afs/here/today \
    INSTALLBIN=/afs/there/now INSTALLMAN3DIR=/afs/for/manpages
make
```

In nested extensions with many subdirectories, the INSTALL* arguments will get propagated to the subdirectories. Be careful to repeat this procedure every time you recompile an extension, unless you are sure the AFS istallation 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 (which usually is `./blib`) 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 INST_ARCHLIB and PERL_ARCHLIB 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.

Note, that there is a `makeaperl` scipt in the perl distribution, that supports the linking of a new perl binary in a similar fashion, but with more options.

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 is rather mighty and will 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 install files into), 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 an installed copy of the perl library. The recommended way is to build extensions after you have run 'make install' on perl itself. Do that in a directory 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 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     = $archlib/CORE
PERL_LIB     = $privlib
PERL_ARCHLIB = $archlib
INST_LIB     = ./blib
INST_ARCHLIB = ./blib/<archname>
```

If perl has not yet been installed then PERL_SRC can be defined on the command line as shown in the previous section.

Useful Default Makefile Macros

FULLEXT = Pathname for extension directory (eg DBD/Oracle).

BASEEXT = Basename part of FULLEXT. May be just equal FULLEXT.

ROOTEXT = Directory part of FULLEXT with leading slash (eg /DBD)

INST_LIBDIR = `$(INST_LIB)$(ROOTEXT)`

INST_AUTODIR = `$(INST_LIB)/auto/$(FULLEXT)`

INST_ARCHAUTODIR = `$(INST_ARCHLIB)/auto/$(FULLEXT)`

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 so

CONFIGURE
: CODE reference. Extension writers are requested to do all their initializing within that subroutine. 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)])

EXE_FILES
: Ref to array of executable files. The files will be copied to the INST_EXE directory. Make realclean will delete them from there again.

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"`

INSTALLARCHLIB
: Used by 'make install', which sets INST_ARCHLIB to this value.

INSTALLBIN
: Used by 'make install' which sets INST_EXE to this value.

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 sets INST_LIB to this value.

INST_ARCHLIB
Same as INST_LIB for architecture dependent files.

INST_EXE
Directory, where executable scripts should be installed during 'make'. Defaults to "./blib/ARCHNAME", just to have a dummy location during testing. make install will set INST_EXE to INSTALLBIN.

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

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.

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.

NORECURS
Boolean. Experimental 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"

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 INSTALLMAN1DIR). They will have PREFIX as a common directory node and will branch from that node into lib/, lib/ARCHNAME, and bin/ unless you override one of them.

PREREQ
Placeholder, not yet implemented. Will eventually be a 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. Needs further evaluation, should probably allow to define prerequisites among header files, libraries, perl version, etc.

SKIP
Arryref. E.g. [qw(name1 name2)] skip (do not write) sections of the Makefile

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.

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.

Additional lowercase attributes

can be used to pass parameters to the methods which implement that part of the Makefile. These are not normally required:

clean

```
{FILES => "*.xyz foo"}
```

dist

```
{TARFLAGS => 'cvfF', COMPRESS => 'gzip', SUFFIX => 'gz',
SHAR => 'shar -m', DIST_CP => 'ln'}
```

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 => '...'}
```

installpm

```
{SPLITLIB => '$(INST_LIB)' (default) or '$(INST_ARCHLIB)'}
```

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 $self = shift;
            local *c_o;
            $_=$self->MM::c_o;
            s/old text/new text/;
            $_;
        }
```

Both methods above are available for backwards compatibility with older Makefile.PLs.

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.

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 command $(PREOP) which defaults to a null command. Does a distdir next and runs `tar` on that directory into a tarfile. Then deletes the distdir. 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 command $(PREOP) which defaults to a null command. Does a distdir next and runs `shar` on that directory into a sharfile. Then deletes the distdir. 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 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         ('@ :')
RCS_LABEL     ('rcs -q -Nv$(VERSION_SYM):')
SHAR          ('shar')
SUFFIX        ('Z')
TAR           ('tar')
TARFLAGS      ('cvf')
```

An example:

```
WriteMakefile( 'dist' => { COMPRESS=>"gzip", SUFFIX=>"gz" })
```

AUTHORS

Andy Dougherty *<doughera@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@HMIVAX.HUMGEN.UPENN.EDU>*. Contact the makemaker mailing list `mailto:makemaker@franz.ww.tu-berlin.de`, if you have any questions.

MODIFICATION HISTORY

For a more complete documentation see the file Changes in the MakeMaker distribution package.

TODO

See the file Todo in the MakeMaker distribution package.

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.

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

the *ExtUtils::MakeMaker* manpage which has handy targets for most of the functionality.

AUTHOR

Andreas Koenig *<koenig@franz.ww.TU−Berlin.DE>*

NAME

ExtUtils::Miniperl.PL

SYNOPSIS

```
use ExtUtils::Miniperl;

writemain(@directories);
```

DESCRIPTION

This whole module is written when perl itself is built from a script called minimod.PL. In case you want to patch it, please patch minimod.PL in the perl distribution instead.

writemain() takes an argument list of directories containing archive libraries that relate to perl modules and should be linked into a new perl binary. It writes to STDOUT a corresponding perlmain.c file that is a plain C file containing all the bootstrap code to make the modules associated with the libraries available from within perl.

The typical usage is from within a Makefile generated by ExtUtils::MakeMaker. So under normal circumstances you won't have to deal with this module directly.

SEE ALSO

the *ExtUtils::MakeMaker* manpage

NAME

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

Fcntl – load the C Fcntl.h defines

SYNOPSIS

```
use Fcntl;
```

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.

NAME

Basename – parse file specifications

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", or "MacOS", 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 you haven't called *fileparse_set_fstype()*, the syntax is chosen by examining the "osname" entry from the `Config` package 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 be identical to 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',
$tail 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. It is provided for 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. 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

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. `$dir` contains the current directory name, and `$_` the current filename within that directory. `$name` contains `"$dir/$_"`. You are *chdir()*'d to `$dir` when the function is called. The function may set `$prune` to prune the tree.

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($_))) &&
    $dev < 0 &&
    ($prune = 1);
}
```

Set the variable `$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: $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. For the moment, `rmtree` expects Unix file specification syntax.
- 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.

AUTHORS

Tim Bunce <Tim.Bunce@ig.co.uk> Charles Bailey <bailey@genetics.upenn.edu>

REVISION

This document was last revised 08-Mar-1995, for perl 5.001

NAME

FileHandle – supply object methods for filehandles

cacheout – keep more files open than the system permits

SYNOPSIS

```
use FileHandle;
autoflush STDOUT 1;

cacheout($path);
print $path @data;
```

DESCRIPTION

See the *perlvar* manpage 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 the `print` entry in the *perlfunc* manpage.

$fh->printf
: See the `printf` entry in the *perlfunc* manpage.

$fh->getline
: This works like <$fh> described in the section on *I/O Operators* in the *perlop* manpage 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.

The *cacheout()* Library

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.

SEE ALSO

the *perlfunc* manpage, the section on *I/O Operators* in the *perlop* manpage, the section on *FileHandle* in the *POSIX* manpage

BUGS

sys/param.h lies with its `NOFILE` define on some systems, so you may have to set `$cacheout::maxopen` yourself.

Some of the methods that set variables (like *format_name()*) don't seem to work.

The POSIX functions that create FileHandle methods should be in this module instead.

Due to backwards compatibility, all filehandles resemble objects of class `FileHandle`, or actually classes derived from that class. They actually aren't. Which means you can't derive your own class from `FileHandle` and inherit those methods.

NAME

GetOptions − extended getopt processing

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 new syntax (long option names, no bundling). It tries to implement the better functionality of traditional, GNU and POSIX *getopt()* functions.

Each description should designate a valid Perl identifier, optionally followed by an argument specifier.

Values for argument specifiers are:

```
<none>   option does not take an argument
!        option does not take an argument and may be negated
=s :s    option takes a mandatory (=) or optional (:) string argument
=i :i    option takes a mandatory (=) or optional (:) integer argument
=f :f    option takes a mandatory (=) or optional (:) real number argument
```

If option ‘‘name’’ is set, it will cause the Perl variable `$opt_name` to be set to the specified value. The calling program can use this variable to detect whether the option has been set. Options that do not take an argument will be set to 1 (one).

Options that take an optional argument will be defined, but set to ‘’ if no actual argument has been supplied.

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`.

Options that do not take a value may have an ‘‘!’’ argument specifier to indicate that they may be negated. E.g. ‘‘foo!’’ will allow **−foo** (which sets `$opt_foo` to 1) and **−nofoo** (which will set `$opt_foo` to 0).

The option name may actually be a list of option names, separated by ‘|’s, e.g. B<"foo|bar|blech=s’’. In this example, options ‘bar’ and ‘blech’ will set `$opt_foo` instead.

Option names may be abbreviated to uniqueness, depending on configuration variable `$autoabbrev`.

Dashes in option names are allowed (e.g. pcc-struct-return) and will be translated to underscores in the corresponding Perl variable (e.g. `$opt_pcc_struct_return`). Note that a lone dash ‘‘−’’ is considered an option, corresponding Perl identifier is `$opt_` .

A double dash ‘‘--’’ signals end of the options list.

If the first option of the list consists of non-alphanumeric characters only, it is interpreted as a generic option starter. Everything starting with one of the characters from the starter will be considered an option.

The default values for the option starters are ‘‘−’’ (traditional), ‘‘--’’ (POSIX) and ‘‘+’’ (GNU, being phased out).

Options that start with ‘‘--’’ may have an argument appended, separated with an ‘‘=’’, e.g. ‘‘—foo=bar’’.

If configuration variable `$getopt_compat` is set to a non-zero value, options that start with ‘‘+’’ may also include their arguments, e.g. ‘‘+foo=bar’’.

A return status of 0 (false) indicates that the function detected one or more errors.

EXAMPLES

If option ‘‘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 ‘‘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 = '--'
```

$autoabbrev
: Allow option names to be abbreviated to uniqueness. Default is 1 unless environment variable POSIXLY_CORRECT has been set.

$getopt_compat
: Allow '+' to start options. Default is 1 unless environment variable POSIXLY_CORRECT has been set.

$option_start
: Regexp with option starters. Default is (--|-) if environment variable POSIXLY_CORRECT has been set, (--|-|\+) otherwise.

$order
: Whether non-options are allowed to be mixed with options. Default is `$REQUIRE_ORDER` if environment variable POSIXLY_CORRECT has been set, `$PERMUTE` otherwise.

$ignorecase
: Ignore case when matching options. Default is 1.

$debug
: Enable debugging output. Default is 0.

NOTE

Does not yet use the Exporter— or even packages!! Thus, it's not a real module.

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.
getopts('oif:');  # -o & -i are boolean flags, -f takes an argument
                  # Sets opt_* as a side effect.
```

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.

NAME

Collate − compare 8-bit scalar data according to the current locale

SYNOPSIS

```
use Collate;
setlocale(LC_COLLATE, 'locale-of-your-choice');
$s1 = new Collate "scalar_data_1";
$s2 = new Collate "scalar_data_2";
```

DESCRIPTION

This module provides you with objects that will collate according to your national character set, providing the POSIX *setlocale()* function should be 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 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. Due to `cmp` and overload magic, `lt`, `le`, `eq`, `ge`, and `gt` work also. 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`. Not all the locales that your vendor supports are necessarily installed: please consult your operating system's documentation.

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

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");
```

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 the *open3* manpage for an alternative that handles STDERR as well.

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.

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.

All caveats from *open2()* continue to apply. See the *open2* manpage for details.

NAME

Net::Ping, pingecho – check a host for upness

SYNOPSIS

```
use Net::Ping;
print "'jimmy' is alive and kicking\n" if pingecho('jimmy', 10) ;
```

DESCRIPTION

This module contains routines to test for the reachability of remote hosts. Currently the only routine implemented is *pingecho()*.

pingecho() uses a TCP echo (*not* an ICMP one) to determine if the remote host is reachable. This is usually adequate to tell that a remote host is available to *rsh* (1), *ftp* (1), or *telnet* (1) onto.

Parameters

hostname
: The remote host to check, specified either as a hostname or as an IP address.

timeout
: The timeout in seconds. If not specified it will default to 5 seconds.

WARNING

pingecho() uses alarm to implement the timeout, so don't set another alarm while you are using it.

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 the *perlfunc* manpage 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 `FileHandle::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 `FileHandle::close()` instead.

fcntl This is identical to Perl's builtin `fcntl()` function.

fdopen Use method `FileHandle::new_from_fd()` instead.

feof Use method `FileHandle::eof()` instead.

ferror Use method `FileHandle::error()` instead.

fflush Use method `FileHandle::flush()` instead.

fgetc Use method `FileHandle::getc()` instead.

fgetpos Use method `FileHandle::getpos()` instead.

fgets Use method `FileHandle::gets()` instead.

fileno Use method `FileHandle::fileno()` instead.

floor This is identical to the C function `floor()`.

fmod This is identical to the C function `fmod()`.

fopen Use method `FileHandle::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 `FileHandle::seek()` instead.

fsetpos Use method `FileHandle::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 `FileHandle::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.

```
$loc = POSIX::setlocale( &POSIX::LC_ALL, "C" );
```

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, isc
```

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 *strtod()* is C–specific.

strtok *strtok()* is C–specific.

strtol *strtol()* is C–specific.

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 `FileHandle::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 `FileHandle::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

FileHandle

new Open a file and return a Perl filehandle. The first parameter is the filename and the second parameter is the mode. The mode should be specified as `a` for append, `w` for write, and `<` or `""` for read.

Open a file for reading.

```
$fh = FileHandle->new( "foo", "" );
die "Unable to open foo for reading" unless $fh;
```

Open a file for writing.

```
$fh = FileHandle->new( "foo", "w" );
die "Unable to open foo for writing" unless $fh;
```

Use `FileHandle::close()` to close the file or let the FileHandle object's destructor perform the close.

clearerr Resets the error indicator and EOF indicator to zero.

```
$fh->clearerr;
```

close Close the file.

```
$fh->close;
```

eof Tests for end of file.

```
if( $fh->eof ){
        print "end of file\n";
}
```

error Returns non-zero if there has been an error while reading or writing a file.

```
if( $fh->error ){
        print "error\n";
}
```

fileno Returns the integer file descriptor associated with the file.

```
$fileno = $fh->fileno;
```

flush Flush the stream.

```
$fh->flush;
```

Returns `undef` on failure.

getc Get a character from the stream.

```
$ch = $fh->getc;
```

getpos Retrieve the file pointer position. The returned value can be used as an argument to `setpos()`.

```
$pos = $fh->getpos;
```

gets Retrieve a line from the open file.

```
$line = $fh->gets;
```

new_from_fd
Open a file using a file descriptor. Return a Perl filehandle. The first parameter should be a file descriptor, which can come from `POSIX::open()`. The second parameter, the mode, should be `a` for append, `w` for write, and `<` or `""` for read. The mode should match the mode which was used when the file descriptor was created.

```
$fd = POSIX::open( "typemap" );
$fh = FileHandle->new_from_fd( $fd, "<" );
die "FileHandle failed" unless $fh;
```

new_tmpfile
Creates a temporary file, opens it for writing, and returns a Perl filehandle. Consult your system's `tmpfile()` manpage for details.

```
$fh = FileHandle->new_tmpfile;
die "FileHandle failed" unless $fh;
```

seek Reposition file pointer.

```
$fh->seek( 2, &POSIX::SEEK_SET );
```

setbuf

setpos Set the file pointer position.

```
$pos = $fh->getpos;
$fh->setpos( $pos );
```

Returns `undef` on failure.

setvbuf Returns `undef` on failure.

tell Returns the current file position, in bytes.

```
$pos = $fh->tell;
```

ungetc

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. The third parameter contains the `sa_flags`.

```
$sigset = POSIX::SigSet->new;
$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( 1, &POSIX::VEOF );
```

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 EAGAIN EBADF EBUSY ECHILD EDEADLK EDOM EEXIST EFAULT EFBIG EINTR EINVAL EIO EISDIR EMFILE EMLINK ENAMETOOLONG ENFILE ENODEV ENOENT ENOEXEC ENOLCK ENOMEM ENOSPC ENOSYS ENOTDIR ENOTEMPTY ENOTTY ENXIO EPERM EPIPE ERANGE EROFS ESPIPE ESRCH 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 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 _IOFBF _IOLBF _IONBF

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 19951212.

NAME

Safe − Safe extension module for Perl

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 `$_` and `@_` (and, technically, the much less frequently used `%_`, the `_` filehandle and so on). This is because otherwise perl operators which default to `$_` 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.

By default, the operator mask for a newly created compartment masks out all operations which give ‘‘access to the system’’ in some sense. This includes masking off operators such as *system*, *open*, *chown*, and *shmget* but does not mask off operators such as *print*, *sysread* and *<HANDL>*. Those file operators are allowed since for the code in the compartment to have access to a filehandle, the code outside the compartment must have explicitly placed the filehandle variable inside the compartment.

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');
```

Operator masks

An operator mask exists at user-level as a string of bytes of length MAXO, each of which is either 0x00 or 0x01. Here, MAXO is the number of operators in the current version of perl. The subroutine *MAXO()* (available for export by package Safe) returns the number of operators in the current version of perl. Note that, unlike the beta versions of the Safe extension, this is a reliable count of the number of operators in the currently running perl executable. The presence of a 0x01 byte at offset **n** of the string indicates that operator number **n** should be masked (i.e. disallowed). The Safe extension makes available routines for converting from operator names to operator numbers (and *vice versa*) and for converting from a list of operator names to the corresponding mask (and *vice versa*).

Methods in class Safe

To create a new compartment, use

```
$cpt = new Safe;
```

Optional arguments are (NAMESPACE, MASK), where

NAMESPACE
: is the root namespace to use for the compartment (defaults to "Safe::Root000000000", auto-incremented for each new compartment); and

MASK
: is the operator mask to use (defaults to a fairly restrictive set).

The following methods can then be used on the compartment object returned by the above constructor. The object argument is implicit in each case.

root (NAMESPACE)
: This is a get-or-set method for the compartment's namespace. With the NAMESPACE argument present, it sets the root namespace for the compartment. With no NAMESPACE argument present, it returns the current root namespace of the compartment.

mask (MASK)
: This is a get-or-set method for the compartment's operator mask. With the MASK argument present, it sets the operator mask for the compartment. With no MASK argument present, it returns the current operator mask of the compartment.

trap (OP, ...)
: This sets bits in the compartment's operator mask corresponding to each operator named in the list of arguments. Each OP can be either the name of an operation or its number. See opcode.h or opcode.pl in the main perl distribution for a canonical list of operator names.

untrap (OP, ...)
: This resets bits in the compartment's operator mask corresponding to each operator named in the list of arguments. Each OP can be either the name of an operation or its number. See opcode.h or opcode.pl in the main perl distribution for a canonical list of operator names.

share (VARNAME, ...)
: This shares the *variable* (s) in the argument list with the compartment. Each VARNAME must be the **name** of a variable with a leading type identifier included. Examples of legal variable names are '$foo' for a scalar, '@foo' for an array, '%foo' for a hash, '&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).

varglob (VARNAME)
: This returns a glob 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). Any attempt by code in STRING to use an operator which is in the compartment's mask 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()**. Note that 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.

rdo (FILENAME)
: This evaluates the contents of file FILENAME inside the compartment. See above documentation on the **reval** method for further details.

Subroutines in package Safe

The Safe package contains subroutines for manipulating operator names and operator masks. All are available for export by the package. 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.

ops_to_mask (OP, ...)
: This takes a list of operator names and returns an operator mask with precisely those operators masked.

mask_to_ops (MASK)
: This takes an operator mask and returns a list of operator names corresponding to those operators which are masked in MASK.

opcode (OP, ...)
: This takes a list of operator names and returns the corresponding list of opcodes (which can then be used as byte offsets into a mask).

opname (OP, ...)
: This takes a list of opcodes and returns the corresponding list of operator names.

fullmask
: This just returns a mask which has all operators masked. It returns the string "\1" x *MAXO()*.

emptymask
: This just returns a mask which has all operators unmasked. It returns the string "\0" x *MAXO()*. This is useful if you want a compartment to make use of the namespace protection features but do not want the default restrictive mask.

MAXO
: This returns the number of operators (and hence the length of an operator mask). Note that, unlike the beta distributions of the Safe extension, this is derived from a genuine integer variable in the perl executable and not from a preprocessor constant. This means that the Safe extension is more robust in the presence of mismatched versions of the perl executable and the Safe extension.

op_mask
: This returns the operator mask which is actually in effect at the time the invocation to the subroutine is compiled. In general, this is probably not terribly useful.

AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk.

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 the section on *Autoloading* in the *perlsub* manpage.

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).

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__.

__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

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.

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_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 PATHANE 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

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

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

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 scxript is `"1..M"` with `M` being the number of tests that should be run within the test script. Test::Harness::*runscripts* (@tests) runs all the testscripts named as arguments and checks standard output for the expected `"ok N"` strings.

After all tests have been performed, *runscripts()* prints some performance statistics that are computed by the Benchmark module.

EXPORT

`&runscripts` 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 1 test, $pct% okay.`

`Failed %d/%d tests, %.2f%% okay.`

If not all tests were successful, the script dies with one of the above messages.

SEE ALSO

See the *Benchmerk* manpage for the underlying timing routines.

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.

NAME

abbrev – create an abbreviation table from a list

SYNOPSIS

```
use Abbrev;
abbrev *HASH, LIST
```

DESCRIPTION

Stores all unambiguous truncations of each element of LIST as keys key in the associative array indicated by `*hash`. The values are the original list elements.

EXAMPLE

```
abbrev(*hash,qw("list edit send abort gripe"));
```

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

SYNOPSIS

```
use Text::Tabs;

#$tabstop = 8; # Defaults
print expand("Hello\tworld");
print unexpand("Hello,        world");
$tabstop = 4;
print join("\n",expand(split(/\n/,
        "Hello\tworld,\nit's a nice day.\n"
        )));
```

DESCRIPTION

This module expands and unexpands tabs into spaces, as per the unix expand and unexpand programs. Either function should be passed an array of strings (newlines may *not* be included, and should be used to split an incoming string into separate elements.) which will be processed and returned.

AUTHOR

David Muir Sharnoff <muir@idiom.com>

NAME

Text::Wrap -- wrap text into a paragraph

SYNOPSIS

```
use Text::Wrap;

$Text::Wrap::columns = 20; # Default
print wrap("\t","",Hello, world, it's a nice day, isn't it?");
```

DESCRIPTION

This module is a simple paragraph formatter that wraps text into a paragraph and indents each line. The single exported function, *wrap()*, takes three arguments. The first is included before the first output line, and the second argument is included before each subsequest output line. The third argument is the text to be wrapped.

AUTHOR

David Muir Sharnoff <muir@idiom.com>

NAME

TieHash, TieHash::Std – base class definitions for tied hashes

SYNOPSIS

```
package NewHash;
require TieHash;

@ISA = (TieHash);

sub DELETE { ... }          # Provides needed method
sub CLEAR { ... }           # Overrides inherited method


package NewStdHash;
require TieHash;

@ISA = (TieHash::Std);

# 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 the `tie` entry in the *perlfunc* manpage for a list of the functions required in order to tie a hash to a package. The basic **TieHash** package provides a `new` method, as well as methods `TIEHASH`, `EXISTS` and `CLEAR`. The **TieHash::Std** package provides most methods required for hashes in the `tie` entry in the *perlfunc* manpage. It inherits from **TieHash**, 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:

TIEHASH classname, LIST
The method invoked by the command `tie %hash, class`. Associates a new hash instance with the specified class. `LIST` would represent additional arguments (along the lines of the *AnyDBM_File* manpage 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 the `tie` entry in the *perlfunc* manpage documentation includes a method called `DESTROY` as a necessary method for tied hashes. Neither **TieHash** nor **TieHash::Std** define a default for this method.

The `CLEAR` method provided by these two packages is not listed in the the `tie` entry in the *perlfunc* manpage section.

MORE INFORMATION

The packages relating to various DBM−related implemetations (*DB_File*, *NDBM_File*, etc.) show examples of general tied hashes, as does the the *Config* manpage module. While these do not utilize **TieHash**, they serve as good working examples.

NAME

Time::Local − efficiently compute tome 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.