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)
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 and nested data structures
perlobj     Perl objects
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
perlapi     Perl application programming interface
perlguts    Perl internal functions for those doing extensions
perlcall    Perl from C calling conventions
perlovl     Perl overloading semantics
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.)

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.

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()*.

If your stdio requires an 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.

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

NAME

perldata – Perl data structures

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

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.

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.

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;
```

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 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. Also note that a single-quoted string must be separated from a preceding word by a space, since single quote is a valid (though discouraged) 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__.

A word that doesn't have any 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, 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 << x 10;  # Legal but discouraged.  Use <<"".
Merry Christmas!

    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 an 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. 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];

# 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);
```

The final element may be an array or a hash:

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

You can actually put an array anywhere in the list, but the first array 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);
```

It is often more readable to use the => operator between key/value pairs (the => operator is actually nothing more than a more visually distinctive synonym for a comma):

```
%map = (
         'red'   => 0x00f,
         'blue'  => 0x0f0,
         'green' => 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
```

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.

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**.)

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.

As of Perl 5, 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 a unary operator, so be careful to use `or` instead of `||` there.

Subroutines declarations can also be imported by a `use` statement.

Also as of Perl 5, 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.

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 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 {
    $_ = <STDIN>;
    ...
} until $_ 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 (ARRAY) 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` (see below). 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).

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. (This feature continues to work in Perl 5 but is deprecated. Please change any occurrences of "if BLOCK" to "if (do BLOCK)".)

The C−style `for` loop works exactly like the corresponding `while` loop:

```
for ($i = 1; $i < 10; $i++) {
    ...
}
```

is the same as

```
$i = 1;
while ($i < 10) {
    ...
} continue {
    $i++;
}
```

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 (unless declared previously with `my`), and regains its former value upon exiting the loop. 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 ARRAY 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. Examples:

```
for (@ary) { s/foo/bar/; }

foreach $elem (@elements) {
    $elem *= 2;
}

for ((10,9,8,7,6,5,4,3,2,1,'BOOM')) {
    print $_, "\n"; sleep(1);
}

for (1..15) { print "Merry Christmas\n"; }

foreach $item (split(/:[\\\n:]*/, $ENV{'TERMCAP'})) {
    print "Item: $item\n";
}
```

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. This 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 one 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 }
```

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.)

```
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.

DESCRIPTIONS

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.

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. Scalar or list context propagates downward into the 2nd or 3rd argument, whichever is selected. 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;
```

Note that this is not 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;
```

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.

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.

?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 most recently executed (and successfully compiled) 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 most recently executed (and successfully compiled) 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
```

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//`.

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 `$_`. (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 ($_ = <STDIN>) { print; }
while (<STDIN>) { print; }
for (;<STDIN>;) { print; }
print while $_ = <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.

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.

If the string inside angle brackets is not a filehandle, 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 `<$foo>` because that's an indirect filehandle as explained in the previous paragraph. You could insert curly brackets to force interpretation as a filename glob: `<${foo}>`. (Alternately, you can 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*).

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

Matching Operations

For a description of how to use regular expressions with the matching and substitution operators, see `m//` and `s///` in the *perlop* manpage. These 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   Use extended regular expressions.
```

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. Together with the capability of embedding comments described later, this goes 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.)

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.

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}`. There is no limit to the size of n or m, but large numbers will chew up more memory.

By default, a quantified subpattern is "greedy", that is, it will match as many times as possible without causing the rest of the pattern not to match. If you want to match the *MINIMUM* number of times possible, follow the quantifier with a "?". Note that the meanings don't change, just their "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
\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
```

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. Within a character class, the only other metacharacter is a backslash (and therefore backslash plus one following character). Everything else, like "." and "*", become literals.

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.

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. (But see `(?...)` below.) Note also that the pattern `(fee|fie|foe)` differs from the pattern `[fee|fie|foe]` in that the former matches "fee", "fie", or "foe" in the target string, while the latter matches anything matched by the classes `[fee]`, `[fie]`, or `[foe]` (i.e. the class `[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 same 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.

Outside of the pattern, always use "$" instead of "\" in front of the digit. The scope of $<*digit*> (and `` $` ``, `$&`, and `$'`) extends to the end of the enclosing BLOCK or eval string, or to the next pattern match with subexpressions. If you want to use parentheses to delimit a subpattern (e.g. a set of alternatives) without saving it as a subpattern, use the appropriate `(?` construct (see below). For backwards compatibility with **sed**, The \<*digit*> notation sometimes works outside the current pattern (in the RHS of a substitute), but should not be relied upon. 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 compatibility) \10 is the same as \010, a backspace, and \11 the same as \011, a tab. And so on. (\1 through \9 are always backreferences.)

`$+` returns whatever the last bracket match matched. `$&` returns the entire matched string. (In ancient versions of Perl, `$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, Perl has 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 governs the function of the extension. Several extensions are already supported:

(?#text) A comment. The text is ignored, but you may not embed extra parens in the text.

(?: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 `/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";
$gotcha += ($var =~ /$pattern/i);
# more flexible:
$pattern = "(?i)foobar";
$gotcha += ($var =~ /$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

2) whenever you see one, you should stop and ‘‘question’’ exactly what is going on.

That’s psychology...

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.

–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. Ordinarily these checks are done only when running setuid or setgid. 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 a an array as though it were a scalar, if your subroutines recurse more than 100 deep, and innumeriable 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.

-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.

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 *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 sleeps 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.

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 example 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, it may be imperative. If FILEHANDLE is an expression, the value is taken as the name of the filehandle.

bless REF,PACKAGE

bless REF

This function tells the referenced object (passed as REF) that it is now an object in PACKAGE— or the current package if no PACKAGE is specified, which is the usual case. It returns the reference for convenience, since a *bless()* is often the last thing in a constructor. 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.

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. 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.

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;
```

chr NUMBER

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

chroot FILENAME

Does the same as the system call of that name. If you don't know what it does, don't worry about it. 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 package address of the appropriate type for the socket. See example in the *perlipc* manpage.

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. 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 at best at best.

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) or *ndbm* (3) 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 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. If your system has neither DBM nor ndbm, calling *dbmopen()* produces a fatal error.

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);
```

defined EXPR

Returns a boolean value saying whether the lvalue 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)` (‘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.

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.

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

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.) An `eof` without an argument returns the eof status for the last file read. Empty parentheses () may be used to indicate the pseudo file formed of the files listed on the command line, i.e. `eof()` is reasonable to use inside a while (<>) loop to detect the end of only the last file. Use `eof(ARGV)` or eof without the parentheses to test *EACH* file in a while (<>) loop. Examples:

```
# insert dashes just before last line of last file
while (<>) {
    if (eof()) {
        print "--------------\n";
    }
    print;
}
```

```
# reset line numbering on each input file
while (<>) {
    print "$.\t$_";
    if (eof) {      # Not eof().
        close(ARGV);
    }
}
```

Practical hint: you almost never need to use `eof` in Perl, because the input operators return undef when they run out of data.

eval EXPR

eval BLOCK

EXPR is parsed and executed as if it were a little Perl program. It is executed in the context of the current Perl program, so that any variable settings, 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 *dbmopen()* 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 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 *flock* (2). 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();
```

Note that *flock()* can't lock things over the network. You need to do locking with *fcntl()* for that.

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);
```

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, `$^A`. Eventually, when a *write()* is done, the contents of `$^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. 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.

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.

getlogin   Returns the current login from */etc/utmp*, if any. If null, use *getpwuid()*.

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

getpeername SOCKET

Returns the packed sockaddr address of other end of the SOCKET connection.

```
# An internet sockaddr
$sockaddr = 'S n a4 x8';
$hersockaddr = getpeername(S);
($family, $port, $heraddr) = unpack($sockaddr,$hersockaddr);
```

getpgrp PID

Returns 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 *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 produce a fatal error 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.

```
# An internet sockaddr
$sockaddr = 'S n a4 x8';
$mysockaddr = getsockname(S);
($family, $port, $myaddr) =
                unpack($sockaddr,$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.

gmtime EXPR

Converts a time as returned by the time function to a 9-element array with the time analyzed for the 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 &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-&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

Returns the decimal value of EXPR interpreted as an hex string. (To interpret 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 documentbelow and the *perlmod* 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. 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 /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.) 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';
$sgttyb_t = "ccccs";                # 4 chars and a short
if (ioctl(STDIN,$TIOCGETP,$sgttyb)) {
    @ary = unpack($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";
}
```

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.

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.

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.

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 *perlipc* manpage.

local EXPR
In general, you should be using "my" instead of "local", because it's faster and safer. Format variables have to use "local" though, as do any other variables whose local value must be visible to called subroutines. This is known as dynamic scoping. Lexical scoping is done with "my", which works more like C's auto declarations.

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. All the listed elements must be legal lvalues. This operator works by saving the current values of those variables in 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 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:

```
sub RANGEVAL {
    local($min, $max, $thunk) = @_;
    local $result = '';
    local $i;

    # Presumably $thunk makes reference to $i

    for ($i = $min; $i < $max; $i++) {
        $result .= eval $thunk;
    }
```

```
        $result;
    }

    if ($sw eq '-v') {
        # init local array with global array
        local @ARGV = @ARGV;
        unshift(@ARGV,'echo');
        system @ARGV;
    }
    # @ARGV restored

    # temporarily add to digits associative array
    if ($base12) {
        # (NOTE: not claiming this is efficient!)
        local(%digits) = (%digits,'t',10,'e',11);
        parse_num();
    }
```

Note that *local()* is a run-time command, and so gets executed every time through a loop. In Perl 4 it used up more stack storage each time until the loop was exited. Perl 5 reclaims the space each time through, but it's still more efficient to declare your variables outside the loop.

When you assign to a localized EXPR, the local doesn't change whether EXPR 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.

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 {&key($_), $_} @array;
```

is just a funny way to write

```
%hash = ();
foreach $_ (@array) {
    $hash{&key($_)} = $_;
}
```

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. 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. 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”. If more than one value is listed, the list must be placed in parens. All the listed elements must be legal lvalues. Only alphanumeric identifiers may be lexically scoped— magical builtins like $/ must be localized with “local” instead. In particular, you’re not allowed to say

```
my $_;        # Illegal.
```

Unlike the “local” declaration, variables declared with “my” are totally hidden from the outside world, including any called subroutines (even if it's the same subroutine— 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 EXPR 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 an undefined value.) Commonly this is used to name the parameters to a subroutine. Examples:

```
sub RANGEVAL {
    my($min, $max, $thunk) = @_;
    my $result = '';
    my $i;

    # Presumably $thunk makes reference to $i

    for ($i = $min; $i < $max; $i++) {
        $result .= eval $thunk;
    }

    $result;
}

if ($sw eq '-v') {
    # init my array with global array
    my @ARGV = @ARGV;
    unshift(@ARGV,'echo');
    system @ARGV;
}
# Outer @ARGV again visible
```

When you assign to the EXPR, the "my" doesn't change whether EXPR 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.

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'".

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

Returns the decimal value of EXPR interpreted as an octal string. (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.) If the filename begins with "|", the filename is interpreted as a command to which output is to be piped, and if the filename ends with a "|", the filename is interpreted as command which pipes input to us. (You may not have a command that pipes both in and out.) 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. 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(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. 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. 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;
```

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");
```

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.

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.

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.

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 non-zero if successful. FILEHANDLE may be a scalar variable name, in which case the variable contains the name of 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.

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.

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.

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 documentbelow, and also the *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.

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.

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 is usin the POSIX module. Returns 1 upon success, 0 otherwise.

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);
```

With Perl 5, filehandles are objects with methods, and the last example is preferably written

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

select RBITS,WBITS,EBITS,TIMEOUT
: This calls the select *system*(2) 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);
```

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.

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 *Lsetpriority* (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.

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 *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 `$a` and `$b` (see example below). They are passed by reference, so don't modify `$a` and `$b`.

Examples:

```
# sort lexically
@articles = sort @files;

# same thing, but with explicit sort routine
@articles = sort {$a cmp $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
```

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 (`/[ \t\n]+/`). 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 C<//., 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)
```

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 space (' ') will split on white space just as split with no arguments does, but leading white space does *NOT* produce a null first field. 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.

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. (The * character for an indirectly specified length is not supported, but you can get the same effect by interpolating a variable into the pattern.)

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)`. 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 comp.security.unix.

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";
}
```

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. 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.

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,PACKAGENAME,LIST

This function binds a variable to a package that will provide the implementation for the variable. VARIABLE is the name of the variable to be enchanted. PACKAGENAME is the name of a package implementing objects of correct type. Any additional arguments are passed to the "new" method of the package. Typically these are arguments such as might be passed to the *dbm_open()* function of C.

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
tie(%HIST, NDBM_File, '/usr/lib/news/history', 1, 0);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
untie(%HIST);
```

A package implementing an associative array should have the following methods:

```
TIEHASH objectname, LIST
DESTROY this
FETCH this, key
STORE this, key, value
DELETE this, key
EXISTS this, key
FIRSTKEY this
NEXTKEY this, lastkey
```

A package implementing an ordinary array should have the following methods:

```
TIEARRAY objectname, LIST
DESTROY this
FETCH this, key
STORE this, key, value
[others TBD]
```

A package implementing a scalar should have the following methods:

```
TIESCALAR objectname, LIST
DESTROY this
FETCH this,
STORE this, value
```

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.

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.

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; }
```

If you don't want your namespace altered, use require instead.

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.

Because this is a wide-open interface, pragmas (compiler directives) are also implemented this way. Currently implemented pragmas are:

```
use integer;
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 throw 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 $ˆ 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. If you attempt to do so, 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($_)
```

(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. This variable should be considered read-only. Remember that only 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()*). (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. (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

$ˆ 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 ˆ) 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

$ˆL 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"?)

$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()*.

$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`. 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.

%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. See <perlsubs>.

NAME

perlsub – Perl subroutines

SYNOPSIS

To declare subroutines:

```
sub NAME;       # A "forward" declaration.
sub NAME BLOCK # A declaration and a definition.
```

To import subroutines:

```
use PACKAGE qw(NAME1 NAME2 NAME3);
```

To call subroutines:

```
&NAME           # Passes current @_ to subroutine.
&NAME(LIST);    # Parens required with & form.
NAME(LIST);     # & is optional with parens.
NAME LIST;      # Parens optional if predeclared/imported.
```

DESCRIPTION

Any arguments passed to the routine come in as array `@_`, that is ($_[0], $_[1], ...). The array `@_` is a local array, but its values are references to the actual scalar parameters. The return value of the subroutine is the value of the last expression evaluated, and can be either an array value or a scalar value. Alternately, a return statement may be used to specify the returned value and exit the subroutine. To create local variables see the *local()* and *my()* operators.

A subroutine may 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.)

Example:

```
sub MAX {
    my $max = pop(@_);
    foreach $foo (@_) {
        $max = $foo if $max < $foo;
    }
    $max;
}

...
$bestday = &MAX($mon,$tue,$wed,$thu,$fri);
```

Example:

```
# get a line, combining continuation lines
#  that start with whitespace
```

```
sub get_line {
    $thisline = $lookahead;
    LINE: while ($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.

Subroutines may be called recursively. If a subroutine is called using the "&" form, the argument list is optional. If omitted, no `@_` array is set up for the subroutine; the `@_` array at the time of the call is visible to subroutine instead.

```
&foo(1,2,3);         # pass three arguments
foo(1,2,3);          # the same

foo();               # pass a null list
&foo();              # the same
&foo;                # pass no arguments--more efficient
```

Passing Symbol Table Entries

[Note: The mechanism described in this section works fine in Perl 5, but the new reference mechanism is generally easier to work with. See the *perlref* manpage.]

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 the 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.

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. A good example of this is the standard Shell module, which can treat undefined subroutine calls as calls to Unix programs.

There are mechanisms available for modules to help them split themselves up into autoloadable files to be used with the standard AutoLoader module. See the document on extensions.

NAME

perlmod – Perl modules (packages)

DESCRIPTION

Packages

Perl provides a mechanism for alternate namespaces to protect packages from stomping on each others variables. By default, a Perl script starts compiling into the package known as `main`. You can switch namespaces using the `package` declaration. The scope of the package declaration is from the declaration itself to the end of the enclosing block (the same scope as the *local()* operator). Typically it would be the first declaration in a file to be included by the `require` 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`. 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 since 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.)

*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 C<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.

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` 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 symbol table entry performs an aliasing operation, i.e.,

```
*dick = *richard;
```

causes variables, subroutines and filehandles 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?

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 wil 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 5, 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

In Perl 5, the notion of packages has been extended into the notion of modules. A module is a package that is defined in a library file of the same name, and is designed to be reusable. It may do this by providing a mechanism for exporting some of its symbols into the symbol table of any package using it. Or it may function as a class definition and make its semantics available implicitly through method calls on the class and its objects, without explicit exportation of any symbols. Or it can do a little of both.

Perl modules are included 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; }
```

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`. Therefore, if you're planning on the module altering your namespace, use `use`; otherwise, use `require`. Otherwise 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()
```

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 *perlapi* 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 libaries 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 if 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).

`integer` Perl pragma to compute arithmetic in integer instead of double

`less` Perl pragma to request less of something from the compiler

`sigtrap` Perl pragma to enable stack backtrace on unexpected signals

`strict` Perl pragma to restrict unsafe constructs

`subs` Perl pragma to predeclare sub names

Standard Modules

The following modules are all expacted to behave in a well-defined manner with respect to namespace pollution because they use the Exporter module. See their own documentation for details.

`Abbrev` create an abbreviation table from a list

`AnyDBM_File`
provide framework for multiple DBMs

`AutoLoader`
load functions only on demand

`AutoSplit` split a package for autoloading

`Basename` parse file anme and path from a specification

`Benchmark` benchmark running times of code

`Carp` warn or die of errors (from perspective of caller)

`CheckTree` run many filetest checks on a tree

`Collate` compare 8-bit scalar data according to the current locale

`Config` access Perl configuration option

`Cwd` get pathname of current working directory

`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` module to control namespace manipulations

`Fcntl` load the C Fcntl.h defines

`FileHandle`
supply object methods for filehandles

`Find` traverse a file tree

`Finddepth` traverse a directory structure depth-first

`Getopt` basic and extended *getopt* (3) processing

`MakeMaker` generate a Makefile for Perl extension

`Open2` open a process for both reading and writing

`Open3` open a process for reading, writing, and error handling

`POSIX` Perl interface to IEEE 1003.1 namespace

`Ping` check a host for upness

`Socket` load the C socket.h defines

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.

The following are popular C extension modules, which while available at Perl 5.0 release time, do not come not 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 their authors before randomly posting asking for their present condition and disposition. There's no guarantee that the names or addresses below have not changed since printing, and in fact, they probably have!

`Curses` Written by William Setzer <*William_Setzer@ncsu.edu*>, while not included with the standard distribution, this extension module ports to most systems. FTP from your nearest Perl archive site, or try

```
ftp://ftp.ncsu.edu/pub/math/wsetzer/cursperl5??.tar.gz
```

It is currently in alpha test, so the name and ftp location may change.

`DBI` This is the portable database interface written by <*Tim.Bunce@ig.co.uk*>. This supersedes the many perl4 ports for database extensions. The official archive for DBperl extensions is *ftp.demon.co.uk:/pub/perl/db*. This archive contains copies of perl4 ports for Ingres, Oracle, Sybase, Informix, Unify, Postgres, and Interbase, as well as rdb and shql and other non-SQL systems.

`DB_File` Fastest and most restriction-free of the DBM bindings, this extension module uses the popular Berkeley DB to *tie()* into your hashes. This has a standardly-distributed man page and dynamic loading extension module, but you'll have to fetch the Berkeley code yourself. See the *DB_File* manpage for where.

`Sx` This extension module is a front to the Athena and Xlib libraries for Perl GUI progamming, originally written by by Dominic Giampaolo <*dbg@sgi.com*>, then and rewritten for Sx by Frédéric Chauveau <*fmc@pasteur.fr*>. It's available for FTP from

```
ftp.pasteur.fr:/pub/Perl/Sx.tar.gz
```

`Tk` This extension module is an object-oriented Perl5 binding to the popular tcl/tk X11 package. However, you need know no TCL to use it! It was written by Malcolm Beattie <*mbeattie@sable.ox.ac.uk*>. If you are unable to locate it using *archie* (1L) or a similar tool, you may try retrieving it from */private/Tk-october.tar.gz* from Malcolm's machine listed above.

NAME

perlref − Perl references and nested data structures

DESCRIPTION

In Perl 4 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;
```

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’’.)

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,

```
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.)

For those who worry about these things, the current implementation uses shallow binding of *local()* variables; *my()* variables are not accessible. This precludes true closures. However, you can work around this with a run-time (rather than a compile-time) *eval()*:

```
{
    my $x = time;
    $coderef = eval "sub { \$x }";
}
```

Normally— if you'd used just `sub{}` or even `eval{}`--your unew sub would only have been able to access the global `$x`. But because you've used a run-time *eval()*, this will not only generate a brand new subroutine reference each time called, it will all grant access to the *my()* variable lexically above it rather than the global one. The particular `$x` accessed will be different for each new sub you create. This mechanism yields deep binding of variables. (If you don't know what closures, deep binding, or shallow binding are, don't worry too much about it.)

5. References are often returned by special subroutines called constructors. Perl objects are just reference 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.

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);
```

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);
```

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 type glob 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)}\n";
```

The way it works is that when the `${...}` is seen in the double-quoted string, it's evaluated as a block. The block executes the call to `mysub(1,2,3)`, and then takes a reference to that. So the whole block returns a reference to a scalar, which is then dereferenced by `${...}` and stuck into the double-quoted string.

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.

Further Reading

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.

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 that has been "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();
    $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 rebless 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');
```

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.

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.

SEE ALSO

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

NAME

perlbot – Bag'o Object Tricks For Perl5 (the BOT)

INTRODUCTION

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 comprehensive guide to Perl5'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.

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;
}

package Bar;

sub new {
        my $type = shift;
        my %params = @_;
        my $self = [];
        $self->[0] = $params{'Left'};
        $self->[1] = $params{'Right'};
        bless $self;
}

package main;

$a = new Foo ( 'High' => 42, 'Low' => 11 );
print "High=$a->{'High'}\n";
print "Low=$a->{'Low'}\n";

$b = new Bar ( 'Left' => 78, 'Right' => 40 );
print "Left=$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;
}

package main;

$a = new Foo 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 $self = {};
        $self->{'buz'} = 42;
        bless $self;
}

package Foo;
@ISA = qw( Bar );

sub new {
        my $self = new Bar;
        $self->{'biz'} = 11;
        bless $self;
}

package main;

$a = new Foo;
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 $self = {};
        $self->{'buz'} = 42;
        bless $self;
}

package Foo;
```

```
sub new {
        my $self = {};
        $self->{'Bar'} = new Bar ();
        $self->{'biz'} = 11;
        bless $self;
}

package main;

$a = new Foo;
print "buz = ", $a->{'Bar'}->{'buz'}, "\n";
print "biz = ", $a->{'biz'}, "\n";
```

OVERRIDING SUPERCLASS METHODS

The following example demonstrates how one might override a superclass method and then call the method after it has been overridden. The Foo::Inherit 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 );
@Foo::Inherit::ISA = @ISA;  # Access to overridden methods.

sub new { bless [] }
sub grr { print "grumble\n" }
sub goo {
        my $self = shift;
        $self->Foo::Inherit::goo();
}
sub mumble {
        my $self = shift;
        $self->Foo::Inherit::mumble();
}
sub google {
        my $self = shift;
        $self->Foo::Inherit::google();
}

package main;

$foo = new Foo;
$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.

```
use SDBM_File;
use POSIX;

package Mydbm;

sub TIEHASH {
    my $self = shift;
    my $ref  = SDBM_File->new(@_);
    bless {'dbm' => $ref};
}
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;

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 { bless {} }
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 since *FOO::bar()* explicitly calls *FOO::private::BAZ()*.

```
package FOO;

sub new { bless {} }
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 { bless {} }

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 { bless {} }
sub bar {
        my $self = shift;
        $self->BAZ;
}
```

```
sub BAZ {
        print "in BAZ\n";
}

package GOOP;
@ISA = qw( FOO );

sub new { bless {} }
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 $self = {};
        $self->{'fizzle'} = \%fizzle;
        bless $self;
}

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 $self = Bar->new;
        $self->{'fizzle'} = \%fizzle;
        bless $self;
}

package main;

$a = Bar->new;
$b = Foo->new;
$a->enter;
$b->enter;
```

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 shift your `@_` arguments in a subroutine, the stack backtrace will not show the shifted off arguments.

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).
```

Optional warnings are enabled by using the **−w** switch. 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.

–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).

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.

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.

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
(F) An error peculiar to VMS.

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 they 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 pipe mailbox device name

(F) An error peculiar to VMS.

Can't get SYSGEN parameter value for MAXBUF

(F) An error peculiar to VMS.

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 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 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 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 as left arg of an implicit −>
: (F) The compiler tried to interpret a bracketed expression as a subscript to an array reference. But to the left of the brackets was an expression that didn't end in an arrow (−>), or look like a subscripted expression. Only subscripted expressions with multiple subscripts are allowed to omit the intervening arrow.

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 a string 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 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 $ instead of %?
(W) You probably said `%hash{$key}` when you meant `$hash{$key}`.

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.

END failed— cleanup aborted
(F) An untrapped exception was raised while executing an END subroutine. The interpreter is immediately exited.

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 $PUTMSG error: %d
(F) An error peculiar to VMS.

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?

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.

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 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 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 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 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 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.

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.

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;
```

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.

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 C<<=> 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.

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 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.

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.

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 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 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);
```

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.

[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.

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. The keyword used are borrowed from FORTRAN: *format()* to declare and *write()* to execute; see their entries in the *manfunc* 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.

Since use of 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 be 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 expression (for at fields, not caret fields), you can farm out any more sophisticated processing by 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 English;
    use Carp;
    sub swrite {
        croak "usage: swrite PICTURE ARGS" unless @ARG;
        local($ACCUMULATOR);
        formline(@ARG);
        return $ACCUMULATOR;
    }

    $string = swrite(<<'END', 1, 2 3);
 Check me out
 @<<<  @|||  @>>>
 END
    print $string;
```

WARNING

During the execution of a format, only global variables are visible, or dynamically-scoped ones declared with *local()*. Lexically scoped variables declared with *my()* are *NOT* available, as they are not considered to reside in the same lexical scope as the format.

NAME

perlipc − Perl interprocess communication

DESCRIPTION

The IPC facilities of Perl are built on the Berkeley socket mechanism. If you don’t have sockets, you can ignore this section. The calls have the same names as the corresponding system calls, but the arguments tend to differ, for two reasons. First, Perl file handles work differently than C file descriptors. Second, Perl already knows the length of its strings, so you don’t need to pass that information.

Client/Server Communication

Here’s a sample TCP client.

```
($them,$port) = @ARGV;
$port = 2345 unless $port;
$them = 'localhost' unless $them;

$SIG{'INT'} = 'dokill';
sub dokill { kill 9,$child if $child; }

use Socket;

$sockaddr = 'S n a4 x8';
chop($hostname = `hostname`);

($name, $aliases, $proto) = getprotobyname('tcp');
($name, $aliases, $port) = getservbyname($port, 'tcp')
    unless $port =~ /^\d+$/;
($name, $aliases, $type, $len, $thisaddr) =
                gethostbyname($hostname);
($name, $aliases, $type, $len, $thataddr) = gethostbyname($them);

$this = pack($sockaddr, &AF_INET, 0, $thisaddr);
$that = pack($sockaddr, &AF_INET, $port, $thataddr);

socket(S, &PF_INET, &SOCK_STREAM, $proto) || die "socket: $!";
bind(S, $this) || die "bind: $!";
connect(S, $that) || die "connect: $!";

select(S); $| = 1; select(stdout);

if ($child = fork) {
    while (<>) {
        print S;
    }
    sleep 3;
    do dokill();
}
else {
    while (<S>) {
        print;
    }
}
```

And here’s a server:

```
($port) = @ARGV;
$port = 2345 unless $port;

use Socket;

$sockaddr = 'S n a4 x8';

($name, $aliases, $proto) = getprotobyname('tcp');
($name, $aliases, $port) = getservbyname($port, 'tcp')
    unless $port =~ /^\d+$/;

$this = pack($sockaddr, &AF_INET, $port, "\0\0\0\0");

select(NS); $| = 1; select(stdout);

socket(S, &PF_INET, &SOCK_STREAM, $proto) || die "socket: $!";
bind(S, $this) || die "bind: $!";
listen(S, 5) || die "connect: $!";

select(S); $| = 1; select(stdout);

for (;;) {
    print "Listening again\n";
    ($addr = accept(NS,S)) || die $!;
    print "accept ok\n";

    ($af,$port,$inetaddr) = unpack($sockaddr,$addr);
    @inetaddr = unpack('C4',$inetaddr);
    print "$af $port @inetaddr\n";

    while (<NS>) {
        print;
        print NS;
    }
}
```

SysV IPC

Here's a small example showing shared memory usage:

```
$IPC_PRIVATE = 0;
$IPC_RMID = 0;
$size = 2000;
$key = shmget($IPC_PRIVATE, $size , 0777 );
die if !defined($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 "$!";
```

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 Perl is executing a setuid script, it 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 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.

NAME

perltrap – Perl traps for the unwary

DESCRIPTION

The biggest trap of all is forgetting to use the **–w** switch; see the *perlrun* manpage. Making your entire program runnable under

```
use strict;
```

can help make your program more bullet-proof, but sometimes it's too annoying for quick throw-away programs.

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.
- 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 type remembering that some functions default to `$_`, or `@ARGV`, or whatever, but that others which you might expect to do not.
- 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.

defined before the compiler sees them. For example:

- Barewords that used to look like strings to Perl will now look like subroutine calls if a subroutine by that name is

```
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 strict` `'subs'` to find such place.

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

- `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.
- Some error messages will be different.
- Some bugs may have been inadvertently removed.

NAME

perlstyle – Perl style guide

DESCRIPTION

Style

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.

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.
- 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)         || die "can't mkdir $tmpdir: $!";
chdir($tmpdir)              || die "can't chdir $tmpdir: $!";
mkdir('tmp',  0777)         || die "can't mkdir $tmpdir/tmp: $!";
```

- 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

perlapi − Perl 5 application programming interface for C extensions

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.

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<tab>x
```

The compiler expects a tab between the parameter name and its type, and any or no whitespace before the type. 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 parameter list of a function must not have whitespace after the open-parenthesis or before the close-parenthesis.

```
INCORRECT                      CORRECT

double                         double
sin( x )                       sin(x)
  double  x                      double  x
```

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. If function parameters are listed under OUTPUT: along with the RETVAL variable then the RETVAL variable must be the last one listed.

```
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<tab>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 only as an output value. The **xsubpp** compiler will normally generate code to read the values of all function parameters from the argument stack and assign them to C variables upon entry to the function. NO_INIT will tell the compiler that some parameters will be used for output rather than for input and that they will be handled before the function terminates.

The following example shows a variation of the *rpcb_gettime()* function. This function uses the timep variable only as an output variable and does not care about its initial contents.

```
bool_t
rpcb_gettime(host,timep)
     char *  host
     time_t  &timep = NO_INIT
     OUTPUT:
     timep
```

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 statments.

```
$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. Occassionally 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(newSVnv(status)));
        PUSHs(sv_2mortal(newSVnv(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

Occassionally 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(newSVnv(timep)));
     else{
     /* Nothing pushed on stack, so an empty */
     /* list is implicitly returned. */
     }
     }
```

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_setptrobj()* macro. The blessing of the object by Perl can be handled by the T_PTROBJ typemap.

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.

Perl Variables

The following demonstrates how the Perl variable `$host` can be accessed from an XSUB. The function **perl_get_sv()** is used to obtain a pointer to the variable, known as an **SV** (Scalar Variable) internally. The package name `RPC` will be added to the name of the variable so *perl_get_sv()* will know in which package `$host` can be found. If the package name is not supplied then *perl_get_sv()* will search package `main` for the variable. The macro **SvPVX()** is then used to dereference the SV to obtain a `char*` pointer to its contents.

```
void
rpcb_gettime()
     PPCODE:
     {
     char *host;
     SV *hostsv;
     time_t timep;

     hostsv = perl_get_sv( "RPC::host", FALSE );
     if( hostsv != NULL ){
          host = SvPVX( hostsv );
          if( rpcb_gettime( host, &timep ) )
               PUSHs(sv_2mortal(newSVnv(timep)));
     }
     }
```

This Perl code can be used to call that XSUB.

```
$RPC::host = "localhost";
$timep = rpcb_gettime();
```

In the above example the SV contained a C `char*` but a Perl scalar variable may also contain numbers and references. If the SV is expected to have a C `int` then the macro **SvIVX()** should be used to dereference the SV. When the SV contains a C double then **SvNVX()** should be used.

The macro **SvRV()** can be used to dereference an SV when it is a Perl reference. The result will be another SV which points to the actual Perl variable. This can then be dereferenced with *SvPVX()*, *SvNVX()*, or *SvIVX()*. The following XSUB will use *SvRV()*.

```
void rpcb_gettime()
      PPCODE:
      {
      char *host;
      SV *rv;
      SV *hostsv;
      time_t timep;

      rv = perl_get_sv( "RPC::host", FALSE );
      if( rv != NULL ){
           hostsv = SvRV( rv );
           host = SvPVX( hostsv );
           if( rpcb_gettime( host, &timep ) )
                PUSHs(sv_2mortal(newSVnv(timep)));
      }
      }
```

This Perl code will create a variable `$RPC::host` which is a reference to `$MY::host`. The variable `$MY::host` contains the hostname which will be used.

```
$MY::host = "localhost";
$RPC::host = \$MY::host;
$timep = rpcb_gettime();
```

The second argument to *perl_get_sv()* will normally be **FALSE** as shown in the above examples. An argument of **TRUE** will cause variables to be created if they do not already exist. One should not use TRUE unless steps are taken to deal with a possibly empty SV.

XSUBs may use **perl_get_av**(), **perl_get_hv**(), and **perl_get_cv()** to access Perl arrays, hashes, and code values.

Interface Stategy

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 only by 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.

The Perl Module

The Perl module is the link between the extension library, which was generated from XS code, and the Perl interpreter. The module is used to tell Perl what the extension library contains. The name and package of the module should match the name of the library.

The following is a Perl module for an extension containing some ONC+ RPC bind library functions.

```
package RPC;

require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
@EXPORT = qw( rpcb_gettime rpcb_getmaps rpcb_getaddr
                rpcb_rmtcall rpcb_set rpcb_unset );

bootstrap RPC;
1;
```

The RPC extension contains the functions found in the `@EXPORT` list. By using the `Exporter` module the RPC module can make these function names visible to the rest of the Perl program. The `DynaLoader` module will allow the RPC module to bootstrap the extension library. To load this extension and make the functions available, the following Perl statement should be used.

```
use RPC;
```

For more information about the DynaLoader consult its documentation in the ext/DynaLoader directory in the Perl source.

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 since 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.

C Headers and Perl

The **h2xs** compiler is designed to convert C header files in /usr/include into Perl extensions. This compiler will create a directory under the `ext` directory of the Perl source and will populate it with a Makefile, a Perl Module, an XS source file, and a MANIFEST file.

The following command will create an extension called `Rusers` from the <rpcsvc/rusers.h> header.

```
h2xs rpcsvc/rusers
```

When the Rusers extension has been compiled and installed Perl can use it to retrieve any `#define` statements which were in the C header.

```
use Rusers;
print "RPC program number for rusers service: ";
print &RUSERSPROG, "\n";
```

Creating A New Extension

The **h2xs** compiler can generate template source files and Makefiles. These templates offer a suitable starting point for most extensions. The following example demonstrates how one might use **h2xs** to create an extension containing the RPC functions in this document.

The extension will not use autoloaded functions and will not define constants, so the **−A** option will be given to **h2xs**. When run from the Perl source directory, the **h2xs** compiler will create the directory ext/RPC and will populate it with files called RPC.xs, RPC.pm, Makefile.PL, and MANIFEST. The XS code for the RPC functions should be added to the RPC.xs file. The `@EXPORT` list in RPC.pm should be updated to include the functions from RPC.xs.

```
h2xs -An RPC
```

To compile the extension for dynamic loading the following command should be executed from the ext/RPC directory.

```
make dynamic
```

If the extension will be statically linked into the Perl binary then the makefile (use `makefile`, not `Makefile`) in the Perl source directory should be edited to add `ext/RPC/RPC.a` to the `static_ext` variable. Before making this change Perl should have already been built. After the makefile has been updated the following command should be executed from the Perl source directory.

```
make
```

Perl's **Configure** script can also be used to add extensions. The extension should be placed in the `ext` directory under the Perl source before Perl has been built and prior to running Configure. When Configure is run it will find the extension along with the other extensions in the `ext` directory and will add it to the list of extensions to be built. When make is run the extension will be built along with the other extensions.

Configure recognizes extensions if they have an XS source file which matches the name of the extension directory. If the extension directory includes a MANIFEST file Configure will search that file for any **.SH** files and extract them after it extracts all the other .SH files listed in the main MANIFEST. The main Perl Makefile will then run **make** in the extension's directory if it finds an XS file matching the name of the extension's directory.

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> September 27, 1994

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 type.

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 a special typedef I32 which will always be a 32-bit integer.

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* scalar, 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 allow Perl to calculate the length by using `sv_setpv`. Be warned, though, that `sv_setpv` determines 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)
```

But note that these are valid only if `SvPOK()` is true.

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**.

Private and Public Values

Recall that the usual method of determining the type of scalar you have is to use `Sv[INP]OK` macros. Since a scalar can be both a number and a string, usually these macros will always return TRUE and calling the `Sv[INP]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.

In general, though, it's best to just use the `Sv[INP]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 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 length of array */

SV**  av_fetch(AV*, I32 key, I32 lval);
        /* Fetches value at key offset, but it seems to
           set the value to lval if lval is non-zero */
SV**  av_store(AV*, I32 key, SV* val);
        /* Stores val at offset key */

void  av_clear(AV*);
        /* Clear out all elements, but leave the array */
void  av_undef(AV*);
        /* Undefines the array, removing all 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);
```

And more miscellaneous functions:

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

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 */
```

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 scalar types (including references). To treat an AV or HV as a scalar, it is simply a matter of casting an AV or HV to an SV.

To create a reference, use the following command:

```
SV*  newRV((SV*) pointer);
```

Once you have a reference, you can use the following macro with a cast to the appropriate typedef (SV, AV, HV):

```
SvRV(SV*)
```

then call the appropriate routines, casting the returned `SV*` to either an `AV*` or `HV*`.

To determine, after dereferencing a reference, if you still have a reference, you can use the following macro:

```
SvROK(SV*)
```

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.

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).

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.

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.

Stashes and Objects

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 hash table 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). The 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 GV structure (for global variable) but represents them with an HV structure.

To get the HV 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.

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, it is legal to 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.

Magic

[This section under construction]

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 `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 should be `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 tells the appropriate routine what, if any, variables are being returned by the Perl subroutine.

All four routines return the number of arguments that the subroutine returned on the Perl stack.

When using these four routines, the programmer must manipulate the Perl stack. These include the following macros and functions:

```
dSP
PUSHMARK()
PUTBACK
SPAGAIN
ENTER
SAVETMPS
FREETMPS
LEAVE
XPUSH*()
```

For more information, consult the *perlcall* manpage.

Memory Allocation

[This section under construction]

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, and Neil Bowers.

DATE

Version 12: 1994/10/16

NAME

perlcall − Calling Perl from C

DESCRIPTION

WARNING: This document is still under construction. There are bound to be a number of inaccuracies, so tread very carefully for now.

The purpose of this document is to show you how to write *callbacks*, i.e. how to call Perl from C. The main focus is on how to interface back to Perl from a bit of C code that has itself been run by Perl, i.e. the 'main' program is a Perl script; you are using it to execute a section of code written in C; that bit of C code wants you to do something with a particular event, so you want a Perl sub to be executed whenever it happens.

Examples where this is necessary include

- 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 get called whenever something nasty occurs. What we would like is for a Perl sub to be called instead.

- The classic example of where callbacks are used is in an event driven program like for X−windows. In this case your register functions to be called whenever a specific events occur, e.g. a mouse button is pressed.

Although the techniques described are applicable to embedding Perl in a C program, this is not the primary goal of this document. For details on embedding Perl in C refer to the *perlembed* manpage (currently unwritten).

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 *perlapi* manpage and the *perlguts* manpage.

This stuff is easier to explain using examples. But first here are a few definitions anyway.

Definitions

Perl has a number of C functions which allow you to call Perl subs. 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 make use of *perl_call_sv* to do what they do.

perl_call_sv takes two parameters, the first is an SV*. This allows you to specify the Perl sub to be called either as a C string (which has first been converted to an SV) or a reference to a sub. Example 7, shows you how you can make use of *perl_call_sv*. The second parameter, `flags`, is a general purpose option command. This parameter is common to all the *perl_call_** functions. It is discussed in the next section.

The function, *perl_call_pv*, is similar as *perl_call_sv* except it expects it's first parameter has to be a C char* which identifies the Perl sub you want to call, e.g. `perl_call_pv("fred", 0)`.

The function *perl_call_method* expects its first argument to contain a blessed reference to a class. Using that reference it looks up and calls `methname` from that class. See example 9.

perl_call_argv calls the Perl sub specified by the `subname` parameter. It also takes the usual `flags` parameter. The final parameter, `argv`, consists of a list of C strings to be sent to the Perl sub. See example 8.

All the functions return a number. This is a count of the number of items returned by the Perl sub on the stack.

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

Flag Values

The `flags` parameter in all the *perl_call_** functions consists of any combination of the symbols defined below, OR'ed together.

G_SCALAR
Calls the Perl sub in a scalar context.

Whatever the Perl sub actually returns, we only want a scalar. If the perl sub does return a scalar, the return value from the *perl_call_** function will be 1 or 0. If 1, then the value actually returned by the Perl sub will be contained on the top of the stack. If 0, then the sub has probably called *die* or you have used the G_DISCARD flag.

If the Perl sub returns a list, the *perl_call_** function will still only return 1 or 0. If 1, then the number of elements in the list will be stored on top of the stack. The actual values of the list will not be accessible.

G_SCALAR is the default flag setting for all the functions.

G_ARRAY
Calls the Perl sub in a list context.

The return code from the *perl_call_** functions will indicate how many elements of the stack are used to store the array.

G_DISCARD
If you are not interested in the values returned by the Perl sub then setting this flag will make Perl get rid of them automatically for you. This will take precedence to either G_SCALAR or G_ARRAY.

If you do not set this flag then you may need to explicitly get rid of temporary values. See example 3 for details.

G_NOARGS
If you are not passing any parameters to the Perl sub, you can save a bit of time by setting this flag. It has the effect of of not creating the `@_` array for the Perl sub.

A point worth noting is that if this flag is specified the Perl sub called can still access an `@_` array from a previous Perl sub. This functionality can be illustrated with the perl code below

```
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
If the Perl sub you are calling has the ability to terminate abnormally, e.g. by calling *die* or by not actually existing, and you want to catch this type of event, specify this flag setting. It will put an *eval { }* around the sub call.

Whenever control returns from the *perl_call_** function you need to check the `$@` variable as you would in a normal Perl script. See example 6 for details of how to do this.

EXAMPLES

Enough of the definition talk, lets have a few examples.

Perl provides many macros to assist in accessing the Perl stack. These macros should always be used when interfacing to Perl internals. Hopefully this should make the code less vulnerable to changes made to Perl in the future.

Another point worth noting is that in the first series of examples I have only made use of the *perl_call_pv* function. This has only been done to ease you into the topic. Wherever possible, if the choice is between using *perl_call_pv* and *perl_call_sv*, I would always try to use *perl_call_sv*.

The code for these examples is stored in the file *perlcall.tar*. (Once this document settles down, all the example code will be available in the file).

Example1: No Parameters, Nothing returned

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

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

and here is the C to call it

```
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. We aren't passing any parameters to *PrintUID* so G_NOARGS can be specified.
2. Ignore `dSP` and `PUSHMARK(sp)` for now. They will be discussed in the next example.
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. Because we specified G_DISCARD, it is not necessary to check the value returned from *perl_call_sv*. It will always be 0.
5. As *perl_call_pv* is being used, the Perl sub is specified as a C string.

Example 2: Passing Parameters

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

So the Perl sub 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. The only flag specified this time is G_DISCARD. As we are passing 2 parameters to the Perl sub this time, we have not specified G_NOARGS.

2. Parameters are passed to the Perl sub using the Perl stack. This is the purpose of the code beginning with the line `dSP` and ending with the line `PUTBACK`.

3. 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 initialises 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.

   If you are calling a Perl sub directly from an XSUB function, it is not necessary to explicitly use the `dSP` macro – it will be declared for you.

4. 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 sub, 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 in Example 1) 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.

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 *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.

Example 3: Returning a Scalar

Now for an example of dealing with the values returned from a Perl sub.

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

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

    $a + $b ;
}
```

As we are now concerned with the return value from *Adder*, the C function 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 sub, plus it will also dump the mortal SV's we created. Having `ENTER`/`SAVETMPS` at the beginning of the code makes sure that no other mortals are destroyed.

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 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 only updated our local copy of the stack pointer. Remember, `PUTBACK` sets the global stack pointer to be the same as our local copy.

Example 4: Returning a list of values

Now, lets extend the previous example to return both the sum of the parameters and the difference.

Here is the Perl sub

```
    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 ;
}
```

Notes

1. We wanted array context, so we used G_ARRAY.
2. Not surprisingly there are 2 POPi's this time because we were retrieving 2 values from the stack. The main point to note is that they came off the stack in reverse order.

Example 5: 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 sub, *Inc*, below takes 2 parameters and increments each.

```
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 return value 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*.

Example 6: Using G_EVAL

Now an example using G_EVAL. Below is a Perl sub which computes the difference of its 2 parameters. If this would result in a negative result, the sub 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);

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

    SPAGAIN ;

    if (count != 1)
        croak ("call_Subtract : expected 1 return 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.
2. The code

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

is the equivalent of this bit of Perl

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

Example 7: Using perl_call_sv

In all the previous examples I have 'hard-wried' the name of the Perl sub to be called from C. Sometimes though, it is necessary to be able to specify the name of the Perl sub from within the Perl script.

Consider the Perl code below

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

    CallSub("fred") ;
```

here is a snippet of XSUB which defines *CallSub*.

```
    void
    CallSub(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, it only allows the Perl sub to be specified as a string. For perl 4 this was adequate, but Perl 5 allows references to subs and anonymous subs. This is where *perl_call_sv* is useful.

The code below for *CallSub* is identical to the previous time except that the `name` parameter is now defined as an SV* and we use *perl_call_sv* instead of *perl_call_pv*.

```
    void
    CallSub(name)
            SV*     name
            CODE:
            PUSHMARK(sp) ;
            perl_call_sv(name, G_DISCARD|G_NOARGS) ;
```

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

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

As you can see, *perl_call_sv* gives you greater flexibility in how you can specify the Perl sub.

Example 8: Using perl_call_argv

Here is a Perl sub 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*.

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

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

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

Example 9: Using perl_call_method

[This section is under construction]

Consider the following Perl code

```
{
  package Mine ;

  sub new     { bless [@_] }
  sub Display { print $_[0][1], "\n" }
}

$a = new Mine ('red', 'green', 'blue') ;
call_Display($a, 'Display') ;
```

The method `Display` just prints out the first element of the list. Here is a XSUB implementation of *call_Display*.

```
void
call_Display(ref, method)
    SV *    ref
    char *  method
    CODE:
    PUSHMARK(sp);
    XPUSHs(ref);
    PUTBACK;

    perl_call_method(method, G_DISCARD) ;
```

Strategies for storing Context Information

[This section is under construction]

One of the trickiest problems to overcome when designing a callback interface is figuring out how to store the mapping between the C callback functions and the Perl equivalent.

Consider the following example.

Alternate Stack Manipulation

[This section is under construction]

Although I have only made use of the POP* macros to access values returned from Perl subs, it is also possible to bypass these macros and read the stack directly.

The code below is example 4 recoded to

SEE ALSO

the *perlapi* 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.

DATE

Version 0.4, 17th October 1994

NAME

perlovl – perl overloading semantics

SYNOPSIS

```
package SomeThing;

%OVERLOAD = (
    '+' => \&myadd,
    '-' => \&mysub,
    # etc
);
...

package main;
$a = new SomeThing 57;
$b=5+$a;
```

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

```
package Number;
%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. Legal values of this hash array are values legal inside `&{ ... }` call, so the name of a subroutine, a reference to a subroutine, or an anonymous subroutine will all work.

The subroutine `$OVERLOAD{"+"}` will be called to execute `$a+$b` if `$a` is a reference to an object blessed into the package `Number`, or `$a` is not an object from a package with defined mathemagic addition, but `$b` is a reference to a `Number`. It can be called also 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 in `values %OVERLOAD` 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 be always an object in the package, so in the situation of `7+$a`, the order of arguments is interchanged. Most probably it does not matter for implementation of the addition method, but whether the arguments are reversed is vital for the subtraction method. The subroutine 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 `$OVERLOAD{"++"}` is called with arguments `($a,undef,'')` when `$a++` is executed.

Overloadable Operations

The following keys of `%OVERLOAD` are recognized:

• *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 operations "−" 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 `%OVERLOAD`.

• *Bit operations*

```
"&", "^", "|", "neg", "!", "~",
```

"`neg`" stands for unary minus. If the method for `neg` is not specified, it can be autogenerated using on 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 KEYS OF %OVERLOAD*.

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

SPECIAL KEYS OF `%OVERLOAD`

Three keys are recognized by Perl that are not covered by the above description.

Last Resort

`$OVERLOAD{"nomethod"}` is a reference to a function of four parameters. If defined, it is called when the overloading mechanism cannot find a method for some operation. The first three arguments of this function coincide with arguments for the corresponding method if it were found, the fourth argument is the key of `%OVERLOAD` corresponding to the missing method. If several methods are tried, the last one is used. Say, `1-$a` can be equivalent to

```
&{ $Pack::OVERLOAD{"nomethod"} }($a,1,1,"-").
```

If some operation cannot be resolved, and there is no `$OVERLOAD{"nomethod"}`, then an exception will be raised via *die()* -- unless `$OVERLOAD{"fallback"}` is true.

Fallback

`$OVERLOAD{"fallback"}` governs what to do if a method for a particular operation is not found. Three different cases are possible depending on value of `$OVERLOAD{"fallback"}`:

- `undef` Perl tries to use a substituted method (see the section on *MAGIC AUTOGENERATION*). If this fails, it then tries to calls `$OVERLOAD{"nomethod"}`; if missing, an exception will be raised.
- TRUE The same as for the `undef` value, but no exception is raised. Instead, it silently reverts to what it would have done were there no `%OVERLOAD` is present.
- defined, but FALSE
  No autogeneration is tried. Perl tries to call `$OVERLOAD{"nomethod"}`, and if this is missing, raises an exception.

Copy Constructor

`$OVERLOAD{"="}` is a reference to a function with three arguments, i.e., it looks like a usual value of `%OVERLOAD`. What is special about this subroutine is that it should not return a blessed reference into a package (as most other methods are expected to), but rather a freshly made copy of its dereferenced argument (see the section on *BUGS*, though). 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 to `$a` and not to change `$b`, a freshly made copy of `$$a` is made, and `$a` is assigned a reference to this object. This operation is executed during `$a++`, (so before this `$$a` coincides with `$$b`), and only if ++ is expressed via `$OPERATOR{'++'}` or `$OPERATOR{'+='}`. Note that if this operation is expressed via '+', i.e., as

```
$a=$b;
$a=$a+1;
```

then `$$a` and `$$b` do not appear as lvalues.

If the copy constructor is required during execution of some mutator, but `$OPERATOR{'='}` is missing, it can be autogenerated as a string copy if an object of the package is a plain scalar.

MAGIC AUTOGENERATION

If a method for an operation is not found, and `$OVERLOAD{"fallback"}` is TRUE or undefined, Perl tries to to autogenerate a substitute method for the missing operation based on defined operations. Autogenerated method substitutions are possible for the following operations:

Assignment forms of arithmetic operations
: `$a=+$b` can use the `$OVERLOAD{"+"}` method if `$OVERLOAD{"+="}` 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 assignment to the dereferenced value, if this value is scalar but 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 becomes promoted to a string first, and its mathemagical qualities is lost. The same can happen with other operations as well.

IMPLEMENTATION

The table of methods for all operations is cached as a magic for the symbol table hash of the package. It is rechecked for changes of `%OVERLOAD` and `@ISA` only during `bless`ing; so if it is changed dynamically, you’ll need an additional fake `bless`ing to update the table.

(Every SVish thing has a magic queue, and a 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 with `%OVERLOAD`, it carries a special flag. Thus the only speed penalty during arithmetic operations without overload is the check of this flag.

In fact, if no `%OVERLOAD` is ever accessed, there is almost no overhead for overloadable operations, so most programs should not suffer measurable performance penalties. Considerable effort was made minimize overhead when `%OVERLOAD` is accessed and the current operation is overloadable but the arguments in question do not belong to packages with `%OVERLOAD`. When in doubt, test your speed with

`%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 there is no `%OVERLOAD`.

The copying like `$a=$b` is shallow; however, a one-level-deep copying is carried out before any operation that can imply an assignment to the object `$b` (or `$a`) refers to, like `$b++`. You can override this behavior by defining your 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's used for overloading, the per-package associative array `%OVERLOAD` now has a special meaning in Perl.

Although the copy constructor is specially designed to make overloading operations with references to an array simpler, as it now works it's useless for this because a subroutine cannot return an array in the same way as it returns a scalar (from the point of view of Perl internals). Expect a change of interface for the copy constructor.

As shipped, `%OVERLOAD` is not inherited via the `@ISA` tree. A patch for this is available from the author.

This document is confusing.

NAME

perlbook − Perl book information

DESCRIPTION

You can order Perl books from O'Reilly & Associates, 1-800-998-9938. Local/overseas is 1-707-829-0515. If you can locate an O'Reilly order form, you can also fax to 1-707-829-0104. *Programming Perl* is a reference work that covers nearly all of Perl (version 4, alas), while *Learning Perl* is a tutorial that covers the most frequently used subset of the language.

```
Programming Perl (the Camel Book):
    ISBN 0-937175-64-1      (English)
    ISBN 4-89052-384-7      (Japanese)

Learning Perl (the Llama Book):
    ISBN 1-56592-042-2      (English)
```

Camel Book Errata and Addenda

The first edition of the Perl book, *Programming Perl*, has the following omissions and goofs.

On page 5, the examples which read

```
eval "/usr/bin/perl
```

should read

```
eval "exec /usr/bin/perl
```

On page 195, the equivalent to the System V sum program only works for very small files. To do larger files, use

```
undef $/;
$checksum = unpack("%32C*",<>) % 65535;
```

The descriptions of alarm and sleep refer to signal SIGALARM. These should refer to SIGALRM.

The **−0** switch to set the initial value of $/ was added to Perl after the book went to press.

The **−l** switch now does automatic line ending processing.

The `qx//` construct is now a synonym for backticks.

`$0` may now be assigned to set the argument displayed by *ps* (1).

The new `@###.##` format was omitted accidentally from the description on formats.

It wasn't known at press time that `s///ee` caused multiple evaluations of the replacement expression. This is to be construed as a feature.

`(LIST) x $count` now does array replication.

There is now no limit on the number of parentheses in a regular expression.

In double-quote context, more escapes are supported: `\e`, `\a`, `\x1b`, `\c[`, `\l`, `\L`, `\u`, `\U`, `\E`. The latter five control upper and lower case translation.

The `$/` variable may now be set to a multi-character delimiter.

There is now a `g` modifier on ordinary pattern matching that causes it to iterate through a string finding multiple matches.

All of the `$^X` variables are new except for `$^T`.

The default top-of-form format for FILEHANDLE is now FILEHANDLE_TOP rather than top.

The `eval {}` and `sort {}` constructs were added in version 4.018.

The `v` and `V` (little-endian) template options for pack and unpack were added in 4.019.

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

less − Perl pragma to request less of something from the compiler

DESCRIPTION

Currently unimplemented, this may someday be a compiler directive to make certain trade-off, such as perhaps

```
use less 'memory';
use less 'CPU';
use less 'fat';
```

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

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 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’’, or ‘‘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 identifiers that's not a subroutine.

```
use strict 'subs';
$SIG{PIPE} = Plumber;       # blows up
$SIG{"PIPE"} = "Plumber";   # just fine
```

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

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

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

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 us 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);
```

This 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

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

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 according 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

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

SYNOPSYS

```
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

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

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 the more direct approach `ls /usr/lib/nls/loc` or `ls /usr/lib/nls`. The locale names are probably something like "xx_XX.(ISO)?8859-N".

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

```
require Cwd;
$dir = Cwd::getcwd()'

use Cwd;
$dir = getcwd()'

use Cwd 'chdir';
chdir "/tmp";
print $ENV{'PWD'};
```

DESCRIPTION

The *getcwd()* function re-implements the *getcwd*(3) (or *getwd*(3)) functions in Perl. 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.)

The *fastgetcwd()* 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.

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

```
require DynaLoader;
push (@ISA, 'DynaLoader');
```

DESCRIPTION

This specification 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.

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 allow pseudo-dynamic linking (using `ld -A` at runtime).

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.

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** (C<`$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(0 function in F<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.

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

AUTHOR

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, and others.

Larry Wall designed the elegant inherited bootstrap mechanism and implemented the first Perl 5 dynamic loader using it.

Tim Bunce, 11 August 1994.

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

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;
```

NAME

Exporter − module to control namespace manipulations

import − import functions into callers namespace

SYNOPSYS

```
package WhatEver;
require Exporter;
@ISA = (Exporter);
@EXPORT    = qw(func1, $foo, %tabs);
@EXPORT_OK = qw(sin cos);
...
use Whatever;
use WhatEver 'sin';
```

DESCRIPTION

The Exporter module is used by well-behaved Perl modules to control what they will export into their user's namespace. The WhatEver module above has placed in its export list the function `func1()`, the scalar `$foo`, and the hash `%tabs`. When someone decides to `use WhatEver`, they get those identifier grafted onto their own namespace. That means the user of package whatever can use the function *func1()* instead of fully qualifying it as *WhatEver::func1()*.

You should be careful of such namespace pollution. Of course, the user of the WhatEver module is free to use a `require` instead of a `use`, which will preserve the sanctity of their namespace.

In particular, you almost certainly shouldn't automatically export functions whose names are already used in the language. For this reason, the `@EXPORT_OK` list contains those function which may be selectively imported, as the *sin()* function was above. See the section on *Overriding builtin functions* in the *perlsub* manpage.

You can't import names that aren't in either the `@EXPORT` or the `@EXPORT_OK` list.

Remember that these two constructs are identical:

```
use WhatEver;

BEGIN {
    require WhatEver;
    import Module;
}
```

The *import()* function above is not predefined in the language. Rather, it's a method in the Exporter module. A sneaky library writer could conceivably have an *import()* method that behaved differently from the standard one, but that's not very friendly.

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

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:

```
print
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
```

The *cacheout()* function will make sure that there's a filehandle open for writing available as the pathname you give it. It automatically closes and re-opens files if you exceed your system file descriptor maximum.

BUGS

sys/param.h lies with it's `NOFILE` define on some systems, so you may have to set `$cacheout::maxopen` yourself.

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

find − traverse a file tree

SYNOPSYS

```
use File::Find;
find(\&wanted, '/foo','/bar');
sub wanted { ... }
```

DESCRIPTION

The *wanted()* function does whatever verificationsyou 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.

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

finddepth − traverse a directory structure depth-first

SYNOPSYS

```
use File::Finddepth;
finddepth(\&wanted, '/foo','/bar');
sub wanted { ... }
```

DESCRIPTION

This is just like `File::Find`, except that it does a depthfirst search uses *finddepth()* rather than *find()*, and performs a depth-first search.

NAME

getopt – Process single-character switches with switch clustering

getopts – Process single-character switches with switch clustering

GetOptions – extended getopt processing

SYNOPSIS

```
use Getopt::Std;
getopt('oDI');  # -o, -D & -I take arg.  Sets opt_* as a side effect.
getopts('oif:');  # likewise, but all of them

use Getopt::Long;
$result = GetOptions (...option-descriptions...);
```

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.

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

MakeMaker − generate a Makefile for Perl extension

SYNOPSIS

```
use ExtUtils::MakeMaker;
```

DESCRIPTION

This utility is designed to write a Makefile for an extension module from a Makefile.PL. It splits the task of generating the Makefile into several subroutines that can be individually overridden. Each subroutines returns the text it wishes to have written to the Makefile.

The best way to learn to use this is to look at how some of the extensions are generated, such as Socket.

AUTHOR

Andy Dougherty <*doughera@lafcol.lafayette.edu*>, Andreas Koenig <*k@franz.ww.TU−Berlin.DE*>, and Tim Bunce <*Tim.Bunce@ig.co.uk*>.

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. See the *open3* manpage for an alternative.

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.

All caveats from *open2()* continue to apply. See the *open2* manpage for details.

NAME

POSIX − Perl interface to IEEE 1003.1 namespace

SYNOPSIS

```
use POSIX;
use POSIX 'strftime';
```

DESCRIPTION

The POSIX module permits you to access all (or nearly all) the standard POSIX 1003.1 identifiers. 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.

To get a list of all the possible identifiers available to you—and their semantics—you should pick up a 1003.1 spec, or look in the *POSIX.pm* module.

EXAMPLES

```
printf "EENTR is %d\n", EINTR;

POSIX::setsid(0);

$fd = POSIX::open($path, O_CREAT|O_EXCL|O_WRONLY, 0644);
    # note: that's a filedescriptor, *NOT* a filehandle
```

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 because they're, supplying the Perl equivalent if one exists. 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.

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

Socket − load the C socket.h defines

SYNOPSIS

```
use Socket;

$proto = (getprotobyname('udp'))[2];
socket(Socket_Handle, PF_INET, SOCK_DGRAM, $proto);
```

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.

NOTE

Only `#define` symbols get translated; you must still correctly pack up your own arguments to pass to *bind()*, etc.