NAME

perlfaq7 - General Perl Language Issues

DESCRIPTION

This section deals with general Perl language issues that don't clearly fit into any of the other sections.

Can I get a BNF/yacc/RE for the Perl language?

There is no BNF, but you can paw your way through the yacc grammar in perly.y in the source distribution if you're particularly brave. The grammar relies on very smart tokenizing code, so be prepared to venture into toke.c as well.

In the words of Chaim Frenkel: "Perl's grammar can not be reduced to BNF. The work of parsing perl is distributed between yacc, the lexer, smoke and mirrors."

What are all these $@%&* punctuation signs, and how do I know when to use them?

They are type specifiers, as detailed in *perldata*:

```
$ for scalar values (number, string or reference)
@ for arrays
% for hashes (associative arrays)
& for subroutines (aka functions, procedures, methods)
* for all types of that symbol name. In version 4 you used them like
  pointers, but in modern perls you can just use references.
```

There are a couple of other symbols that you're likely to encounter that aren't really type specifiers:

```
<> are used for inputting a record from a filehandle.
\  takes a reference to something.
```

Note that <FILE> is *neither* the type specifier for files nor the name of the handle. It is the <> operator applied to the handle FILE. It reads one line (well, record--see *"$/" in perlvar*) from the handle FILE in scalar context, or *all* lines in list context. When performing open, close, or any other operation besides <> on files, or even when talking about the handle, do *not* use the brackets. These are correct: `eof(FH)`, `seek(FH, 0, 2)` and "copying from STDIN to FILE".

Do I always/never have to quote my strings or use semicolons and commas?

Normally, a bareword doesn't need to be quoted, but in most cases probably should be (and must be under `use strict`). But a hash key consisting of a simple word and the left-hand operand to the => operator both count as though they were quoted:

```
This                    is like this
------------            ---------------
$foo{line}              $foo{'line'}
bar => stuff            'bar' => stuff
```

The final semicolon in a block is optional, as is the final comma in a list. Good style (see *perlstyle*) says to put them in except for one-liners:

```
if ($whoops) { exit 1 }
my @nums = (1, 2, 3);

if ($whoops) {
    exit 1;
}
```

```
my @lines = (
    "There Beren came from mountains cold",
    "And lost he wandered under leaves",
);
```

How do I skip some return values?

One way is to treat the return values as a list and index into it:

```
$dir = (getpwnam($user))[7];
```

Another way is to use undef as an element on the left-hand-side:

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

You can also use a list slice to select only the elements that you need:

```
($dev, $ino, $uid, $gid) = ( stat($file) )[0,1,4,5];
```

How do I temporarily block warnings?

If you are running Perl 5.6.0 or better, the `use warnings` pragma allows fine control of what warnings are produced. See *perllexwarn* for more details.

```
{
    no warnings;          # temporarily turn off warnings
    $x = $y + $z;         # I know these might be undef
}
```

Additionally, you can enable and disable categories of warnings. You turn off the categories you want to ignore and you can still get other categories of warnings. See *perllexwarn* for the complete details, including the category names and hierarchy.

```
{
    no warnings 'uninitialized';
    $x = $y + $z;
}
```

If you have an older version of Perl, the `$^W` variable (documented in *perlvar*) controls runtime warnings for a block:

```
{
    local $^W = 0;        # temporarily turn off warnings
    $x = $y + $z;         # I know these might be undef
}
```

Note that like all the punctuation variables, you cannot currently use my() on `$^W`, only local().

What's an extension?

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

Why do Perl operators have different precedence than C operators?

Actually, they don't. All C operators that Perl copies have the same precedence in Perl as they do in C. The problem is with operators that C doesn't have, especially functions that give a list context to everything on their right, eg. print, chmod, exec, and so on. Such functions are called "list operators" and appear as such in the precedence table in *perlop*.

A common mistake is to write:

```
unlink $file || die "snafu";
```

This gets interpreted as:

```
unlink ($file || die "snafu");
```

To avoid this problem, either put in extra parentheses or use the super low precedence `or` operator:

```
(unlink $file) || die "snafu";
unlink $file or die "snafu";
```

The "English" operators (`and`, `or`, `xor`, and `not`) deliberately have precedence lower than that of list operators for just such situations as the one above.

Another operator with surprising precedence is exponentiation. It binds more tightly even than unary minus, making `-2**2` produce a negative four and not a positive one. It is also right-associating, meaning that `2**3**2` is two raised to the ninth power, not eight squared.

Although it has the same precedence as in C, Perl's `?:` operator produces an lvalue. This assigns $x to either $if_true or $if_false, depending on the trueness of $maybe:

```
($maybe ? $if_true : $if_false) = $x;
```

How do I declare/create a structure?

In general, you don't "declare" a structure. Just use a (probably anonymous) hash reference. See *perlref* and *perldsc* for details. Here's an example:

```
$person = {};                   # new anonymous hash
$person->{AGE}  = 24;           # set field AGE to 24
$person->{NAME} = "Nat";        # set field NAME to "Nat"
```

If you're looking for something a bit more rigorous, try *perlootut*.

How do I create a module?

perlnewmod is a good place to start, ignore the bits about uploading to CPAN if you don't want to make your module publicly available.

ExtUtils::ModuleMaker and *Module::Starter* are also good places to start. Many CPAN authors now use *Dist::Zilla* to automate as much as possible.

Detailed documentation about modules can be found at: *perlmod*, *perlmodlib*, *perlmodstyle*.

If you need to include C code or C library interfaces use h2xs. h2xs will create the module distribution structure and the initial interface files. *perlxs* and *perlxstut* explain the details.

How do I adopt or take over a module already on CPAN?

Ask the current maintainer to make you a co-maintainer or transfer the module to you.

If you can not reach the author for some reason contact the PAUSE admins at modules@perl.org who may be able to help, but each case it treated separately.

- Get a login for the Perl Authors Upload Server (PAUSE) if you don't already have one: *http://pause.perl.org*
- Write to modules@perl.org explaining what you did to contact the current maintainer. The PAUSE admins will also try to reach the maintainer.

- Post a public message in a heavily trafficked site announcing your intention to take over the module.
- Wait a bit. The PAUSE admins don't want to act too quickly in case the current maintainer is on holiday. If there's no response to private communication or the public post, a PAUSE admin can transfer it to you.

How do I create a class?

(contributed by brian d foy)

In Perl, a class is just a package, and methods are just subroutines. Perl doesn't get more formal than that and lets you set up the package just the way that you like it (that is, it doesn't set up anything for you).

See also *perlootut*, a tutorial that covers class creation, and *perlobj*.

How can I tell if a variable is tainted?

You can use the tainted() function of the Scalar::Util module, available from CPAN (or included with Perl since release 5.8.0). See also *"Laundering and Detecting Tainted Data" in perlsec*.

What's a closure?

Closures are documented in *perlref*.

Closure is a computer science term with a precise but hard-to-explain meaning. Usually, closures are implemented in Perl as anonymous subroutines with lasting references to lexical variables outside their own scopes. These lexicals magically refer to the variables that were around when the subroutine was defined (deep binding).

Closures are most often used in programming languages where you can have the return value of a function be itself a function, as you can in Perl. Note that some languages provide anonymous functions but are not capable of providing proper closures: the Python language, for example. For more information on closures, check out any textbook on functional programming. Scheme is a language that not only supports but encourages closures.

Here's a classic non-closure function-generating function:

```
sub add_function_generator {
    return sub { shift() + shift() };
}

my $add_sub = add_function_generator();
my $sum = $add_sub->(4,5);                # $sum is 9 now.
```

The anonymous subroutine returned by add_function_generator() isn't technically a closure because it refers to no lexicals outside its own scope. Using a closure gives you a *function template* with some customization slots left out to be filled later.

Contrast this with the following make_adder() function, in which the returned anonymous function contains a reference to a lexical variable outside the scope of that function itself. Such a reference requires that Perl return a proper closure, thus locking in for all time the value that the lexical had when the function was created.

```
sub make_adder {
    my $addpiece = shift;
    return sub { shift() + $addpiece };
}

my $f1 = make_adder(20);
```

```
my $f2 = make_adder(555);
```

Now `$f1->($n)` is always 20 plus whatever $n you pass in, whereas `$f2->($n)` is always 555 plus whatever $n you pass in. The $addpiece in the closure sticks around.

Closures are often used for less esoteric purposes. For example, when you want to pass in a bit of code into a function:

```
my $line;
timeout( 30, sub { $line = <STDIN> } );
```

If the code to execute had been passed in as a string, `'$line = <STDIN>'`, there would have been no way for the hypothetical timeout() function to access the lexical variable $line back in its caller's scope.

Another use for a closure is to make a variable *private* to a named subroutine, e.g. a counter that gets initialized at creation time of the sub and can only be modified from within the sub. This is sometimes used with a BEGIN block in package files to make sure a variable doesn't get meddled with during the lifetime of the package:

```
BEGIN {
    my $id = 0;
    sub next_id { ++$id }
}
```

This is discussed in more detail in *perlsub*; see the entry on *Persistent Private Variables*.

What is variable suicide and how can I prevent it?

This problem was fixed in perl 5.004_05, so preventing it means upgrading your version of perl. ;)

Variable suicide is when you (temporarily or permanently) lose the value of a variable. It is caused by scoping through my() and local() interacting with either closures or aliased foreach() iterator variables and subroutine arguments. It used to be easy to inadvertently lose a variable's value this way, but now it's much harder. Take this code:

```
my $f = 'foo';
sub T {
    while ($i++ < 3) { my $f = $f; $f .= "bar"; print $f, "\n" }
}

T;
print "Finally $f\n";
```

If you are experiencing variable suicide, that `my $f` in the subroutine doesn't pick up a fresh copy of the `$f` whose value is `'foo'`. The output shows that inside the subroutine the value of `$f` leaks through when it shouldn't, as in this output:

```
foobar
foobarbar
foobarbarbar
Finally foo
```

The $f that has "bar" added to it three times should be a new `$f` `my $f` should create a new lexical variable each time through the loop. The expected output is:

```
foobar
foobar
```

```
foobar
Finally foo
```

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

You need to pass references to these objects. See *"Pass by Reference" in perlsub* for this particular question, and *perlref* for information on references.

Passing Variables and Functions

Regular variables and functions are quite easy to pass: just pass in a reference to an existing or anonymous variable or function:

```
func( \$some_scalar );

func( \@some_array  );
func( [ 1 .. 10 ]   );

func( \%some_hash   );
func( { this => 10, that => 20 }   );

func( \&some_func   );
func( sub { $_[0] ** $_[1] }   );
```

Passing Filehandles

As of Perl 5.6, you can represent filehandles with scalar variables which you treat as any other scalar.

```
open my $fh, $filename or die "Cannot open $filename! $!";
func( $fh );

sub func {
    my $passed_fh = shift;

    my $line = <$passed_fh>;
}
```

Before Perl 5.6, you had to use the `*FH` or `\*FH` notations. These are "typeglobs"--see *"Typeglobs and Filehandles" in perldata* and especially *"Pass by Reference" in perlsub* for more information.

Passing Regexes

Here's an example of how to pass in a string and a regular expression for it to match against. You construct the pattern with the `qr//` operator:

```
sub compare {
    my ($val1, $regex) = @_;
    my $retval = $val1 =~ /$regex/;
    return $retval;
}
$match = compare("old McDonald", qr/d.*D/i);
```

Passing Methods

To pass an object method into a subroutine, you can do this:

```
call_a_lot(10, $some_obj, "methname")
sub call_a_lot {
    my ($count, $widget, $trick) = @_;
    for (my $i = 0; $i < $count; $i++) {
```

```
            $widget->$trick();
        }
    }
```

Or, you can use a closure to bundle up the object, its method call, and arguments:

```
my $whatnot = sub { $some_obj->obfuscate(@args) };
func($whatnot);
sub func {
    my $code = shift;
    &$code();
}
```

You could also investigate the can() method in the UNIVERSAL class (part of the standard perl distribution).

How do I create a static variable?

(contributed by brian d foy)

In Perl 5.10, declare the variable with `state`. The `state` declaration creates the lexical variable that persists between calls to the subroutine:

```
sub counter { state $count = 1; $count++ }
```

You can fake a static variable by using a lexical variable which goes out of scope. In this example, you define the subroutine `counter`, and it uses the lexical variable `$count`. Since you wrap this in a BEGIN block, `$count` is defined at compile-time, but also goes out of scope at the end of the BEGIN block. The BEGIN block also ensures that the subroutine and the value it uses is defined at compile-time so the subroutine is ready to use just like any other subroutine, and you can put this code in the same place as other subroutines in the program text (i.e. at the end of the code, typically). The subroutine `counter` still has a reference to the data, and is the only way you can access the value (and each time you do, you increment the value). The data in chunk of memory defined by `$count` is private to `counter`.

```
BEGIN {
    my $count = 1;
    sub counter { $count++ }
}

my $start = counter();

.... # code that calls counter();

my $end = counter();
```

In the previous example, you created a function-private variable because only one function remembered its reference. You could define multiple functions while the variable is in scope, and each function can share the "private" variable. It's not really "static" because you can access it outside the function while the lexical variable is in scope, and even create references to it. In this example, `increment_count` and `return_count` share the variable. One function adds to the value and the other simply returns the value. They can both access `$count`, and since it has gone out of scope, there is no other way to access it.

```
BEGIN {
    my $count = 1;
    sub increment_count { $count++ }
```

```
    sub return_count    { $count }
}
```

To declare a file-private variable, you still use a lexical variable. A file is also a scope, so a lexical variable defined in the file cannot be seen from any other file.

See *"Persistent Private Variables" in perlsub* for more information. The discussion of closures in *perlref* may help you even though we did not use anonymous subroutines in this answer. See *"Persistent Private Variables" in perlsub* for details.

What's the difference between dynamic and lexical (static) scoping? Between local() and my()?

`local($x)` saves away the old value of the global variable `$x` and assigns a new value for the duration of the subroutine *which is visible in other functions called from that subroutine*. This is done at run-time, so is called dynamic scoping. local() always affects global variables, also called package variables or dynamic variables.

`my($x)` creates a new variable that is only visible in the current subroutine. This is done at compile-time, so it is called lexical or static scoping. my() always affects private variables, also called lexical variables or (improperly) static(ly scoped) variables.

For instance:

```
sub visible {
    print "var has value $var\n";
}

sub dynamic {
    local $var = 'local';    # new temporary value for the still-global
    visible();               #   variable called $var
}

sub lexical {
    my $var = 'private';     # new private variable, $var
    visible();               # (invisible outside of sub scope)
}

$var = 'global';

visible();              # prints global
dynamic();              # prints local
lexical();              # prints global
```

Notice how at no point does the value "private" get printed. That's because $var only has that value within the block of the lexical() function, and it is hidden from the called subroutine.

In summary, local() doesn't make what you think of as private, local variables. It gives a global variable a temporary value. my() is what you're looking for if you want private variables.

See *"Private Variables via my()" in perlsub* and *"Temporary Values via local()" in perlsub* for excruciating details.

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

If you know your package, you can just mention it explicitly, as in $Some_Pack::var. Note that the notation $::var is **not** the dynamic $var in the current package, but rather the one in the "main" package, as though you had written $main::var.

```
use vars '$var';
local $var = "global";
my    $var = "lexical";

print "lexical is $var\n";
print "global  is $main::var\n";
```

Alternatively you can use the compiler directive our() to bring a dynamic variable into the current lexical scope.

```
require 5.006; # our() did not exist before 5.6
use vars '$var';

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

print "lexical is $var\n";

{
    our $var;
    print "global  is $var\n";
}
```

What's the difference between deep and shallow binding?

In deep binding, lexical variables mentioned in anonymous subroutines are the same ones that were in scope when the subroutine was created. In shallow binding, they are whichever variables with the same names happen to be in scope when the subroutine is called. Perl always uses deep binding of lexical variables (i.e., those created with my()). However, dynamic variables (aka global, local, or package variables) are effectively shallowly bound. Consider this just one more reason not to use them. See the answer to *What's a closure?*.

Why doesn't "my($foo) = <$fh>;" work right?

`my()` and `local()` give list context to the right hand side of =. The <$fh> read operation, like so many of Perl's functions and operators, can tell which context it was called in and behaves appropriately. In general, the scalar() function can help. This function does nothing to the data itself (contrary to popular myth) but rather tells its argument to behave in whatever its scalar fashion is. If that function doesn't have a defined scalar behavior, this of course doesn't help you (such as with sort()).

To enforce scalar context in this particular case, however, you need merely omit the parentheses:

```
local($foo) = <$fh>;        # WRONG
local($foo) = scalar(<$fh>);   # ok
local $foo  = <$fh>;        # right
```

You should probably be using lexical variables anyway, although the issue is the same here:

```
my($foo) = <$fh>;    # WRONG
my $foo  = <$fh>;    # right
```

How do I redefine a builtin function, operator, or method?

Why do you want to do that? :-)

If you want to override a predefined function, such as open(), then you'll have to import the new

definition from a different module. See *"Overriding Built-in Functions" in perlsub*.

If you want to overload a Perl operator, such as + or **, then you'll want to use the `use overload` pragma, documented in *overload*.

If you're talking about obscuring method calls in parent classes, see *"Overriding methods and method resolution" in perlootut*.

What's the difference between calling a function as &foo and foo()?

(contributed by brian d foy)

Calling a subroutine as `&foo` with no trailing parentheses ignores the prototype of `foo` and passes it the current value of the argument list, `@_`. Here's an example; the `bar` subroutine calls `&foo`, which prints its arguments list:

```
sub foo { print "Args in foo are: @_\n"; }

sub bar { &foo; }

bar( "a", "b", "c" );
```

When you call `bar` with arguments, you see that `foo` got the same `@_`:

```
Args in foo are: a b c
```

Calling the subroutine with trailing parentheses, with or without arguments, does not use the current `@_`. Changing the example to put parentheses after the call to `foo` changes the program:

```
sub foo { print "Args in foo are: @_\n"; }

sub bar { &foo(); }

bar( "a", "b", "c" );
```

Now the output shows that `foo` doesn't get the `@_` from its caller.

```
Args in foo are:
```

However, using `&` in the call still overrides the prototype of `foo` if present:

```
sub foo ($$$) { print "Args infoo are: @_\n"; }

sub bar_1 { &foo; }
sub bar_2 { &foo(); }
sub bar_3 { foo( $_[0], $_[1], $_[2] ); }
# sub bar_4 { foo(); }
# bar_4 doesn't compile: "Not enough arguments for main::foo at ..."

bar_1( "a", "b", "c" );
# Args in foo are: a b c

bar_2( "a", "b", "c" );
# Args in foo are:

bar_3( "a", "b", "c" );
```

```
    # Args in foo are: a b c
```

The main use of the @_ pass-through feature is to write subroutines whose main job it is to call other subroutines for you. For further details, see *perlsub*.

How do I create a switch or case statement?

In Perl 5.10, use the `given-when` construct described in *perlsyn*:

```
use 5.010;

given ( $string ) {
    when( 'Fred' )        { say "I found Fred!" }
    when( 'Barney' )      { say "I found Barney!" }
    when( /Bamm-?Bamm/ )  { say "I found Bamm-Bamm!" }
    default               { say "I don't recognize the name!" }
};
```

If one wants to use pure Perl and to be compatible with Perl versions prior to 5.10, the general answer is to use `if-elsif-else`:

```
for ($variable_to_test) {
    if    (/pat1/)  { }     # do something
    elsif (/pat2/)  { }     # do something else
    elsif (/pat3/)  { }     # do something else
    else            { }     # default
}
```

Here's a simple example of a switch based on pattern matching, lined up in a way to make it look more like a switch statement. We'll do a multiway conditional based on the type of reference stored in $whatchamacallit:

```
SWITCH: for (ref $whatchamacallit) {

    /^$/           && die "not a reference";

    /SCALAR/       && do {
                       print_scalar($$ref);
                       last SWITCH;
                     };

    /ARRAY/        && do {
                       print_array(@$ref);
                       last SWITCH;
                     };

    /HASH/         && do {
                       print_hash(%$ref);
                       last SWITCH;
                     };

    /CODE/         && do {
                       warn "can't print function ref";
                       last SWITCH;
                     };
```

```
        # DEFAULT

        warn "User defined type skipped";

    }
```

See *perlsyn* for other examples in this style.

Sometimes you should change the positions of the constant and the variable. For example, let's say you wanted to test which of many answers you were given, but in a case-insensitive way that also allows abbreviations. You can use the following technique if the strings all start with different characters or if you want to arrange the matches so that one takes precedence over another, as `"SEND"` has precedence over `"STOP"` here:

```
chomp($answer = <>);
if    ("SEND"  =~ /^\Q$answer/i) { print "Action is send\n"  }
elsif ("STOP"  =~ /^\Q$answer/i) { print "Action is stop\n"  }
elsif ("ABORT" =~ /^\Q$answer/i) { print "Action is abort\n" }
elsif ("LIST"  =~ /^\Q$answer/i) { print "Action is list\n"  }
elsif ("EDIT"  =~ /^\Q$answer/i) { print "Action is edit\n"  }
```

A totally different approach is to create a hash of function references.

```
my %commands = (
    "happy" => \&joy,
    "sad",  => \&sullen,
    "done"  => sub { die "See ya!" },
    "mad"   => \&angry,
);

print "How are you? ";
chomp($string = <STDIN>);
if ($commands{$string}) {
    $commands{$string}->();
} else {
    print "No such command: $string\n";
}
```

Starting from Perl 5.8, a source filter module, `Switch`, can also be used to get switch and case. Its use is now discouraged, because it's not fully compatible with the native switch of Perl 5.10, and because, as it's implemented as a source filter, it doesn't always work as intended when complex syntax is involved.

How can I catch accesses to undefined variables, functions, or methods?

The AUTOLOAD method, discussed in *"Autoloading" in perlsub* lets you capture calls to undefined functions and methods.

When it comes to undefined variables that would trigger a warning under `use warnings`, you can promote the warning to an error.

```
use warnings FATAL => qw(uninitialized);
```

Why can't a method included in this same file be found?

Some possible reasons: your inheritance is getting confused, you've misspelled the method name, or the object is of the wrong type. Check out *perlootut* for details about any of the above cases. You may

also use `print ref($object)` to find out the class `$object` was blessed into.

Another possible reason for problems is that you've used the indirect object syntax (eg, `find Guru "Samy"`) on a class name before Perl has seen that such a package exists. It's wisest to make sure your packages are all defined before you start using them, which will be taken care of if you use the `use` statement instead of `require`. If not, make sure to use arrow notation (eg., `Guru->find("Samy")`) instead. Object notation is explained in *perlobj*.

Make sure to read about creating modules in *perlmod* and the perils of indirect objects in *"Method Invocation" in perlobj*.

How can I find out my current or calling package?

(contributed by brian d foy)

To find the package you are currently in, use the special literal `__PACKAGE__`, as documented in *perldata*. You can only use the special literals as separate tokens, so you can't interpolate them into strings like you can with variables:

```
my $current_package = __PACKAGE__;
print "I am in package $current_package\n";
```

If you want to find the package calling your code, perhaps to give better diagnostics as *Carp* does, use the `caller` built-in:

```
sub foo {
    my @args = ...;
    my( $package, $filename, $line ) = caller;

    print "I was called from package $package\n";
    );
```

By default, your program starts in package `main`, so you will always be in some package.

This is different from finding out the package an object is blessed into, which might not be the current package. For that, use `blessed` from *Scalar::Util*, part of the Standard Library since Perl 5.8:

```
use Scalar::Util qw(blessed);
my $object_package = blessed( $object );
```

Most of the time, you shouldn't care what package an object is blessed into, however, as long as it claims to inherit from that class:

```
my $is_right_class = eval { $object->isa( $package ) }; # true or false
```

And, with Perl 5.10 and later, you don't have to check for an inheritance to see if the object can handle a role. For that, you can use `DOES`, which comes from `UNIVERSAL`:

```
my $class_does_it = eval { $object->DOES( $role ) }; # true or false
```

You can safely replace `isa` with `DOES` (although the converse is not true).

How can I comment out a large block of Perl code?

(contributed by brian d foy)

The quick-and-dirty way to comment out more than one line of Perl is to surround those lines with Pod directives. You have to put these directives at the beginning of the line and somewhere where Perl expects a new statement (so not in the middle of statements like the # comments). You end the

comment with =cut, ending the Pod section:

```
=pod

my $object = NotGonnaHappen->new();

ignored_sub();

$wont_be_assigned = 37;

=cut
```

The quick-and-dirty method only works well when you don't plan to leave the commented code in the source. If a Pod parser comes along, your multiline comment is going to show up in the Pod translation. A better way hides it from Pod parsers as well.

The =begin directive can mark a section for a particular purpose. If the Pod parser doesn't want to handle it, it just ignores it. Label the comments with comment. End the comment using =end with the same label. You still need the =cut to go back to Perl code from the Pod comment:

```
=begin comment

my $object = NotGonnaHappen->new();

ignored_sub();

$wont_be_assigned = 37;

=end comment

=cut
```

For more information on Pod, check out *perlpod* and *perlpodspec*.

How do I clear a package?

Use this code, provided by Mark-Jason Dominus:

```
sub scrub_package {
    no strict 'refs';
    my $pack = shift;
    die "Shouldn't delete main package"
        if $pack eq "" || $pack eq "main";
    my $stash = *{$pack . '::'}{HASH};
    my $name;
    foreach $name (keys %$stash) {
        my $fullname = $pack . '::' . $name;
        # Get rid of everything with that name.
        undef $$fullname;
        undef @$fullname;
        undef %$fullname;
        undef &$fullname;
        undef *$fullname;
    }
}
```

Or, if you're using a recent release of Perl, you can just use the Symbol::delete_package() function instead.

How can I use a variable as a variable name?

Beginners often think they want to have a variable contain the name of a variable.

```
$fred    = 23;
$varname = "fred";
++$$varname;         # $fred now 24
```

This works *sometimes*, but it is a very bad idea for two reasons.

The first reason is that this technique *only works on global variables*. That means that if $fred is a lexical variable created with my() in the above example, the code wouldn't work at all: you'd accidentally access the global and skip right over the private lexical altogether. Global variables are bad because they can easily collide accidentally and in general make for non-scalable and confusing code.

Symbolic references are forbidden under the `use strict` pragma. They are not true references and consequently are not reference-counted or garbage-collected.

The other reason why using a variable to hold the name of another variable is a bad idea is that the question often stems from a lack of understanding of Perl data structures, particularly hashes. By using symbolic references, you are just using the package's symbol-table hash (like `%main::`) instead of a user-defined hash. The solution is to use your own hash or a real reference instead.

```
$USER_VARS{"fred"} = 23;
my $varname = "fred";
$USER_VARS{$varname}++;  # not $$varname++
```

There we're using the %USER_VARS hash instead of symbolic references. Sometimes this comes up in reading strings from the user with variable references and wanting to expand them to the values of your perl program's variables. This is also a bad idea because it conflates the program-addressable namespace and the user-addressable one. Instead of reading a string and expanding it to the actual contents of your program's own variables:

```
$str = 'this has a $fred and $barney in it';
$str =~ s/(\$\w+)/$1/eeg;          # need double eval
```

it would be better to keep a hash around like %USER_VARS and have variable references actually refer to entries in that hash:

```
$str =~ s/\$(\w+)/$USER_VARS{$1}/g;   # no /e here at all
```

That's faster, cleaner, and safer than the previous approach. Of course, you don't need to use a dollar sign. You could use your own scheme to make it less confusing, like bracketed percent symbols, etc.

```
$str = 'this has a %fred% and %barney% in it';
$str =~ s/%(\w+)%/$USER_VARS{$1}/g;   # no /e here at all
```

Another reason that folks sometimes think they want a variable to contain the name of a variable is that they don't know how to build proper data structures using hashes. For example, let's say they wanted two hashes in their program: %fred and %barney, and that they wanted to use another scalar variable to refer to those by name.

```
$name = "fred";
$$name{WIFE} = "wilma";     # set %fred
```

```
$name = "barney";
$$name{WIFE} = "betty";    # set %barney
```

This is still a symbolic reference, and is still saddled with the problems enumerated above. It would be far better to write:

```
$folks{"fred"}{WIFE}   = "wilma";
$folks{"barney"}{WIFE} = "betty";
```

And just use a multilevel hash to start with.

The only times that you absolutely *must* use symbolic references are when you really must refer to the symbol table. This may be because it's something that one can't take a real reference to, such as a format name. Doing so may also be important for method calls, since these always go through the symbol table for resolution.

In those cases, you would turn off `strict 'refs'` temporarily so you can play around with the symbol table. For example:

```
@colors = qw(red blue green yellow orange purple violet);
for my $name (@colors) {
    no strict 'refs';  # renege for the block
    *$name = sub { "<FONT COLOR='$name'>@_</FONT>" };
}
```

All those functions (red(), blue(), green(), etc.) appear to be separate, but the real code in the closure actually was compiled only once.

So, sometimes you might want to use symbolic references to manipulate the symbol table directly. This doesn't matter for formats, handles, and subroutines, because they are always global--you can't use my() on them. For scalars, arrays, and hashes, though--and usually for subroutines-- you probably only want to use hard references.

What does "bad interpreter" mean?

(contributed by brian d foy)

The "bad interpreter" message comes from the shell, not perl. The actual message may vary depending on your platform, shell, and locale settings.

If you see "bad interpreter - no such file or directory", the first line in your perl script (the "shebang" line) does not contain the right path to perl (or any other program capable of running scripts). Sometimes this happens when you move the script from one machine to another and each machine has a different path to perl--/usr/bin/perl versus /usr/local/bin/perl for instance. It may also indicate that the source machine has CRLF line terminators and the destination machine has LF only: the shell tries to find /usr/bin/perl<CR>, but can't.

If you see "bad interpreter: Permission denied", you need to make your script executable.

In either case, you should still be able to run the scripts with perl explicitly:

```
% perl script.pl
```

If you get a message like "perl: command not found", perl is not in your PATH, which might also mean that the location of perl is not where you expect it so you need to adjust your shebang line.

Do I need to recompile XS modules when there is a change in the C library?

(contributed by Alex Beamish)

If the new version of the C library is ABI-compatible (that's Application Binary Interface compatible) with the version you're upgrading from, and if the shared library version didn't change, no re-compilation should be necessary.

AUTHOR AND COPYRIGHT