## NAME

perlmod - Perl modules (packages and symbol tables)

## DESCRIPTION

### Packages

Perl provides a mechanism for alternative namespaces to protect packages from stomping on each other's variables. In fact, there's really no such thing as a global variable in Perl. The package statement declares the compilation unit as being in the given namespace. The scope of the package declaration is from the declaration itself through the end of the enclosing block, `eval`, or file, whichever comes first (the same scope as the my() and local() operators). Unqualified dynamic identifiers will be in this namespace, except for those few identifiers that if unqualified, default to the main package instead of the current one as described below. A package statement affects only dynamic variables--including those you've used local() on--but *not* lexical variables created with my(). Typically it would be the first declaration in a file included by the `do`, `require`, or `use` operators. You can switch into a package in more than one place; it merely influences which symbol table is used by the compiler for the rest of that block. You can refer to variables and filehandles in other packages by prefixing the identifier with the package name and a double colon: `$Package::Variable`. If the package name is null, the `main` package is assumed. That is, `$::sail` is equivalent to `$main::sail`.

The old package delimiter was a single quote, but double colon is now the preferred delimiter, in part because it's more readable to humans, and in part because it's more readable to **emacs** macros. It also makes C++ programmers feel like they know what's going on--as opposed to using the single quote as separator, which was there to make Ada programmers feel like they knew what was going on. Because the old-fashioned syntax is still supported for backwards compatibility, if you try to use a string like `"This is $owner's house"`, you'll be accessing `$owner::s`; that is, the $s variable in package `owner`, which is probably not what you meant. Use braces to disambiguate, as in `"This is ${owner}'s house"`.

Packages may themselves contain package separators, as in `$OUTER::INNER::var`. This implies nothing about the order of name lookups, however. There are no relative packages: all symbols are either local to the current package, or must be fully qualified from the outer package name down. For instance, there is nowhere within package `OUTER` that `$INNER::var` refers to `$OUTER::INNER::var`. `INNER` refers to a totally separate global package.

Only identifiers starting with letters (or underscore) are stored in a package's symbol table. All other symbols are kept in package `main`, including all punctuation variables, like `$_`. In addition, when unqualified, the identifiers STDIN, STDOUT, STDERR, ARGV, ARGVOUT, ENV, INC, and SIG are forced to be in package `main`, even when used for other purposes than their built-in ones. If you have a package called `m`, `s`, or `y`, then you can't use the qualified form of an identifier because it would be instead interpreted as a pattern match, a substitution, or a transliteration.

Variables beginning with underscore used to be forced into package main, but we decided it was more useful for package writers to be able to use leading underscore to indicate private variables and method names. However, variables and functions named with a single `_`, such as `$_` and `sub _`, are still forced into the package `main`. See also *"Technical Note on the Syntax of Variable Names" in perlvar*.

`eval`ed strings are compiled in the package in which the eval() was compiled. (Assignments to `$SIG{}`, however, assume the signal handler specified is in the `main` package. Qualify the signal handler name if you wish to have a signal handler in a package.) For an example, examine *perldb.pl* in the Perl library. It initially switches to the `DB` package so that the debugger doesn't interfere with variables in the program you are trying to debug. At various points, however, it temporarily switches back to the `main` package to evaluate various expressions in the context of the `main` package (or wherever you came from). See *perldebug*.

The special symbol `__PACKAGE__` contains the current package, but cannot (easily) be used to

construct variable names.

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

## Symbol Tables

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

The value in each entry of the hash is what you are referring to when you use the `*name` typeglob notation. In fact, the following have the same effect, though the first is more efficient because it does the symbol table lookups at compile time:

```
local *main::foo    = *main::bar;
local $main::{foo}  = $main::{bar};
```

(Be sure to note the **vast** difference between the second line above and `local $main::foo = $main::bar`. The former is accessing the hash `%main::`, which is the symbol table of package `main`. The latter is simply assigning scalar `$bar` in package `main` to scalar `$foo` of the same package.)

You can use this to print out all the variables in a package, for instance. The standard but antiquated *dumpvar.pl* library and the CPAN module Devel::Symdump make use of this.

Assignment to a typeglob performs an aliasing operation, i.e.,

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

causes variables, subroutines, formats, and file and directory handles accessible via the identifier `richard` also to be accessible via the identifier `dick`. If you want to alias only a particular variable or subroutine, assign a reference instead:

```
*dick = \$richard;
```

Which makes $richard and $dick the same variable, but leaves @richard and @dick as separate arrays. Tricky, eh?

There is one subtle difference between the following statements:

```
*foo = *bar;
*foo = \$bar;
```

`*foo = *bar` makes the typeglobs themselves synonymous while `*foo = \$bar` makes the SCALAR portions of two distinct typeglobs refer to the same scalar value. This means that the following code:

```
$bar = 1;
*foo = \$bar;       # Make $foo an alias for $bar

{
    local $bar = 2; # Restrict changes to block
    print $foo;     # Prints '1'!
}
```

Would print '1', because `$foo` holds a reference to the *original* `$bar` -- the one that was stuffed away by `local()` and which will be restored when the block ends. Because variables are accessed through the typeglob, you can use `*foo = *bar` to create an alias which can be localized. (But be

aware that this means you can't have a separate @foo and @bar, etc.)

What makes all of this important is that the Exporter module uses glob aliasing as the import/export mechanism. Whether or not you can properly localize a variable that has been exported from a module depends on how it was exported:

```
    @EXPORT = qw($FOO); # Usual form, can't be localized
    @EXPORT = qw(*FOO); # Can be localized
```

You can work around the first case by using the fully qualified name ($Package::FOO) where you need a local value, or by overriding it by saying *FOO = *Package::FOO in your script.

The *x = \$y mechanism may be used to pass and return cheap references into or from subroutines if you don't want to copy the whole thing. It only works when assigning to dynamic variables, not lexicals.

```
    %some_hash = ();   # can't be my()
    *some_hash = fn( \%another_hash );
    sub fn {
 local *hashsym = shift;
 # now use %hashsym normally, and you
 # will affect the caller's %another_hash
 my %nhash = (); # do what you want
 return \%nhash;
    }
```

On return, the reference will overwrite the hash slot in the symbol table specified by the *some_hash typeglob. This is a somewhat tricky way of passing around references cheaply when you don't want to have to remember to dereference variables explicitly.

Another use of symbol tables is for making "constant" scalars.

```
    *PI = \3.14159265358979;
```

Now you cannot alter $PI, which is probably a good thing all in all. This isn't the same as a constant subroutine, which is subject to optimization at compile-time. A constant subroutine is one prototyped to take no arguments and to return a constant expression. See *perlsub* for details on these. The use constant pragma is a convenient shorthand for these.

You can say *foo{PACKAGE} and *foo{NAME} to find out what name and package the *foo symbol table entry comes from. This may be useful in a subroutine that gets passed typeglobs as arguments:

```
    sub identify_typeglob {
        my $glob = shift;
        print 'You gave me ', *{$glob}{PACKAGE}, '::', *{$glob}{NAME},
"\n";
    }
    identify_typeglob *foo;
    identify_typeglob *bar::baz;
```

This prints

```
    You gave me main::foo
    You gave me bar::baz
```

The *foo{THING} notation can also be used to obtain references to the individual elements of *foo. See *perlref*.

Subroutine definitions (and declarations, for that matter) need not necessarily be situated in the package whose symbol table they occupy. You can define a subroutine outside its package by explicitly qualifying the name of the subroutine:

```
    package main;
    sub Some_package::foo { ... }   # &foo defined in Some_package
```

This is just a shorthand for a typeglob assignment at compile time:

```
    BEGIN { *Some_package::foo = sub { ... } }
```

and is *not* the same as writing:

```
    {
 package Some_package;
 sub foo { ... }
    }
```

In the first two versions, the body of the subroutine is lexically in the main package, *not* in Some_package. So something like this:

```
    package main;

    $Some_package::name = "fred";
    $main::name = "barney";

    sub Some_package::foo {
 print "in ", __PACKAGE__, ": \$name is '$name'\n";
    }

    Some_package::foo();
```

prints:

```
    in main: $name is 'barney'
```

rather than:

```
    in Some_package: $name is 'fred'
```

This also has implications for the use of the SUPER:: qualifier (see *perlobj*).

## BEGIN, CHECK, INIT and END

Four specially named code blocks are executed at the beginning and at the end of a running Perl program. These are the `BEGIN`, `CHECK`, `INIT`, and `END` blocks.

These code blocks can be prefixed with `sub` to give the appearance of a subroutine (although this is not considered good style). One should note that these code blocks don't really exist as named subroutines (despite their appearance). The thing that gives this away is the fact that you can have **more than one** of these code blocks in a program, and they will get **all** executed at the appropriate moment. So you can't execute any of these code blocks by name.

A `BEGIN` code block is executed as soon as possible, that is, the moment it is completely defined, even before the rest of the containing file (or string) is parsed. You may have multiple `BEGIN` blocks within a file (or eval'ed string) -- they will execute in order of definition. Because a `BEGIN` code block executes immediately, it can pull in definitions of subroutines and such from other files in time to be

visible to the rest of the compile and run time. Once a BEGIN has run, it is immediately undefined and any code it used is returned to Perl's memory pool.

It should be noted that BEGIN code blocks **are** executed inside string `eval()`'s. The CHECK and INIT code blocks are **not** executed inside a string eval, which e.g. can be a problem in a mod_perl environment.

An END code block is executed as late as possible, that is, after perl has finished running the program and just before the interpreter is being exited, even if it is exiting as a result of a die() function. (But not if it's morphing into another program via `exec`, or being blown out of the water by a signal--you have to trap that yourself (if you can).) You may have multiple END blocks within a file--they will execute in reverse order of definition; that is: last in, first out (LIFO). END blocks are not executed when you run perl with the `-c` switch, or if compilation fails.

Note that END code blocks are **not** executed at the end of a string `eval()`: if any END code blocks are created in a string `eval()`, they will be executed just as any other END code block of that package in LIFO order just before the interpreter is being exited.

Inside an END code block, $? contains the value that the program is going to pass to `exit()`. You can modify $? to change the exit value of the program. Beware of changing $? by accident (e.g. by running something via `system`).

CHECK and INIT code blocks are useful to catch the transition between the compilation phase and the execution phase of the main program.

CHECK code blocks are run just after the **initial** Perl compile phase ends and before the run time begins, in LIFO order. CHECK code blocks are used in the Perl compiler suite to save the compiled state of the program.

INIT blocks are run just before the Perl runtime begins execution, in "first in, first out" (FIFO) order. For example, the code generators documented in *perlcc* make use of INIT blocks to initialize and resolve pointers to XSUBs.

When you use the **-n** and **-p** switches to Perl, BEGIN and END work just as they do in **awk**, as a degenerate case. Both BEGIN and CHECK blocks are run when you use the **-c** switch for a compile-only syntax check, although your main code is not.

The **begincheck** program makes it all clear, eventually:

```
#!/usr/bin/perl

# begincheck

print         " 8. Ordinary code runs at runtime.\n";

END { print   "14.   So this is the end of the tale.\n" }
INIT { print  " 5. INIT blocks run FIFO just before runtime.\n" }
CHECK { print " 4.   So this is the fourth line.\n" }

print         " 9.   It runs in order, of course.\n";

BEGIN { print " 1. BEGIN blocks run FIFO during compilation.\n" }
END { print   "13.   Read perlmod for the rest of the story.\n" }
CHECK { print " 3. CHECK blocks run LIFO at compilation's end.\n" }
INIT { print  " 6.   Run this again, using Perl's -c switch.\n" }

print         "10.   This is anti-obfuscated code.\n";
```

```
END { print   "12. END blocks run LIFO at quitting time.\n" }
BEGIN { print " 2.   So this line comes out second.\n" }
INIT { print  " 7.   You'll see the difference right away.\n" }

print         "11.   It merely _looks_ like it should be confusing.\n";

__END__
```

## Perl Classes

There is no special class syntax in Perl, but a package may act as a class if it provides subroutines to act as methods. Such a package may also derive some of its methods from another class (package) by listing the other package name(s) in its global @ISA array (which must be a package global, not a lexical).

For more on this, see *perltoot* and *perlobj*.

## Perl Modules

A module is just a set of related functions in a library file, i.e., a Perl package with the same name as the file. It is specifically designed to be reusable by other modules or programs. It may do this by providing a mechanism for exporting some of its symbols into the symbol table of any package using it, or it may function as a class definition and make its semantics available implicitly through method calls on the class and its objects, without explicitly exporting anything. Or it can do a little of both.

For example, to start a traditional, non-OO module called Some::Module, create a file called *Some/Module.pm* and start with this template:

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

use strict;
use warnings;

BEGIN {
    use Exporter   ();
    our ($VERSION, @ISA, @EXPORT, @EXPORT_OK, %EXPORT_TAGS);

    # set the version for version checking
    $VERSION     = 1.00;
    # if using RCS/CVS, this may be preferred
    $VERSION = sprintf "%d.%03d", q$Revision: 1.1 $ =~ /(\d+)/g;

    @ISA         = qw(Exporter);
    @EXPORT      = qw(&func1 &func2 &func4);
    %EXPORT_TAGS = ( );     # eg: TAG => [ qw!name1 name2! ],

    # your exported package globals go here,
    # as well as any optionally exported functions
    @EXPORT_OK   = qw($Var1 %Hashit &func3);
}
our @EXPORT_OK;

# exported package globals go here
our $Var1;
our %Hashit;
```

```
    # non-exported package globals go here
    our @more;
    our $stuff;

    # initialize package globals, first exported ones
    $Var1   = '';
    %Hashit = ();

    # then the others (which are still accessible as $Some::Module::stuff)
    $stuff  = '';
    @more   = ();

    # all file-scoped lexicals must be created before
    # the functions below that use them.

    # file-private lexicals go here
    my $priv_var    = '';
    my %secret_hash = ();

    # here's a file-private function as a closure,
    # callable as &$priv_func;  it cannot be prototyped.
    my $priv_func = sub {
        # stuff goes here.
    };

    # make all your functions, whether exported or not;
    # remember to put something interesting in the {} stubs
    sub func1      {}    # no prototype
    sub func2()    {}    # proto'd void
    sub func3($$)  {}    # proto'd to 2 scalars

    # this one isn't exported, but could be called!
    sub func4(\%)  {}    # proto'd to 1 hash ref

    END { }       # module clean-up code here (global destructor)

    ## YOUR CODE GOES HERE

    1;  # don't forget to return a true value from the file
```

Then go on to declare and use your variables in functions without any qualifications. See *Exporter* and the *perlmodlib* for details on mechanics and style issues in module creation.

Perl modules are included into your program by saying

```
    use Module;
```

or

```
    use Module LIST;
```

This is exactly equivalent to

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

or

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

As a special case

```
use Module ();
```

is exactly equivalent to

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

All Perl module files have the extension *.pm*. The `use` operator assumes this so you don't have to spell out "*Module.pm*" in quotes. This also helps to differentiate new modules from old *.pl* and *.ph* files. Module names are also capitalized unless they're functioning as pragmas; pragmas are in effect compiler directives, and are sometimes called "pragmatic modules" (or even "pragmata" if you're a classicist).

The two statements:

```
require SomeModule;
require "SomeModule.pm";
```

differ from each other in two ways. In the first case, any double colons in the module name, such as `Some::Module`, are translated into your system's directory separator, usually "/". The second case does not, and would have to be specified literally. The other difference is that seeing the first `require` clues in the compiler that uses of indirect object notation involving "SomeModule", as in `$ob = purge SomeModule`, are method calls, not function calls. (Yes, this really can make a difference.)

Because the `use` statement implies a `BEGIN` block, the importing of semantics happens as soon as the `use` statement is compiled, before the rest of the file is compiled. This is how it is able to function as a pragma mechanism, and also how modules are able to declare subroutines that are then visible as list or unary operators for the rest of the current file. This will not work if you use `require` instead of `use`. With `require` you can get into this problem:

```
require Cwd;  # make Cwd:: accessible
$here = Cwd::getcwd();

use Cwd;   # import names from Cwd::
$here = getcwd();

require Cwd;      # make Cwd:: accessible
$here = getcwd();   # oops! no main::getcwd()
```

In general, `use Module ()` is recommended over `require Module`, because it determines module availability at compile time, not in the middle of your program's execution. An exception would be if two modules each tried to `use` each other, and each also called a function from that other module. In that case, it's easy to use `require` instead.

Perl packages may be nested inside other package names, so we can have package names containing `::`. But if we used that package name directly as a filename it would make for unwieldy or impossible filenames on some systems. Therefore, if a module's name is, say, `Text::Soundex`, then its definition is actually found in the library file *Text/Soundex.pm*.

Perl modules always have a *.pm* file, but there may also be dynamically linked executables (often ending in *.so*) or autoloaded subroutine definitions (often ending in *.al*) associated with the module. If so, these will be entirely transparent to the user of the module. It is the responsibility of the *.pm* file to load (or arrange to autoload) any additional functionality. For example, although the POSIX module happens to do both dynamic loading and autoloading, the user can say just `use POSIX` to get it all.

### Making your module threadsafe

Since 5.6.0, Perl has had support for a new type of threads called interpreter threads (ithreads). These threads can be used explicitly and implicitly.

Ithreads work by cloning the data tree so that no data is shared between different threads. These threads can be used by using the `threads` module or by doing fork() on win32 (fake fork() support). When a thread is cloned all Perl data is cloned, however non-Perl data cannot be cloned automatically. Perl after 5.7.2 has support for the `CLONE` special subroutine. In `CLONE` you can do whatever you need to do, like for example handle the cloning of non-Perl data, if necessary. `CLONE` will be called once as a class method for every package that has it defined (or inherits it). It will be called in the context of the new thread, so all modifications are made in the new area. Currently CLONE is called with no parameters other than the invocant package name, but code should not assume that this will remain unchanged, as it is likely that in future extra parameters will be passed in to give more information about the state of cloning.

If you want to CLONE all objects you will need to keep track of them per package. This is simply done using a hash and Scalar::Util::weaken().

Perl after 5.8.7 has support for the `CLONE_SKIP` special subroutine. Like `CLONE`, `CLONE_SKIP` is called once per package; however, it is called just before cloning starts, and in the context of the parent thread. If it returns a true value, then no objects of that class will be cloned; or rather, they will be copied as unblessed, undef values. This provides a simple mechanism for making a module threadsafe; just add `sub CLONE_SKIP { 1 }` at the top of the class, and `DESTROY()` will be now only be called once per object. Of course, if the child thread needs to make use of the objects, then a more sophisticated approach is needed.

Like `CLONE`, `CLONE_SKIP` is currently called with no parameters other than the invocant package name, although that may change. Similarly, to allow for future expansion, the return value should be a single `0` or `1` value.

## SEE ALSO

See *perlmodlib* for general style issues related to building Perl modules and classes, as well as descriptions of the standard library and CPAN, *Exporter* for how Perl's standard import/export mechanism works, *perltoot* and *perltooc* for an in-depth tutorial on creating classes, *perlobj* for a hard-core reference document on objects, *perlsub* for an explanation of functions and scoping, and *perlxstut* and *perlguts* for more information on writing extension modules.