Introduction to mod_perl (part VI): Even More Perl Basics

By Stas Bekman

If you like this article, please make a donation to the Perl development grant fund.

Table of Contents

Variables Globally, Lexically Scoped And Fully Qualified

You will hear a lot about namespaces, symbol tables and lexical scoping in Perl discussions, but little of it will make any sense without a few key facts:

Symbols, Symbol Tables and Packages; Typeglobs

There are two important types of symbol: package global and lexical. We will talk about lexical symbols later, for now we will talk only about package global symbols, which we will refer to simply as *global symbols*.

The names of pieces of your code (subroutine names) and the names of your global variables are symbols. Global symbols reside in one symbol table or another. The code itself and the data do not; the symbols are the names of pointers which point (indirectly) to the memory areas which contain the code and data. (Note for C/C++ programmers: we use the term 'pointer' in a general sense of one piece of data referring to another piece of data not in a specific sense as used in C or C++.)

There is one symbol table for each package, (which is why *global symbols* are really *package global symbols*).

You are always working in one package or another.

Like in C, where the first function you write must be called main(), the first statement of your first Perl script is in package `main::` which is the default package. Unless you say otherwise by using the `package` statement, your symbols are all in package `main::`. You should be aware straight away that files and packages are *not related*. You can have any number of packages in a single file; and a single package can be in one file or spread over many files. However it is very common to have a single package in a single file. To declare a package you write:

```
package mypackagename;
```

From the following line you are in package `mypackagename` and any symbols you declare reside in that package. When you create a symbol (variable, subroutine etc.) Perl uses the name of the package in which you are currently working as a prefix to create the fully qualified name of the symbol.

When you create a symbol, Perl creates a symbol table entry for that symbol in the current package's symbol table (by default `main::`). Each symbol table entry is called a *typeglob*. Each typeglob can hold information on a scalar, an array, a hash, a subroutine (code), a filehandle, a directory handle and a format, each of which all have the same name. So you see now that there are two indirections for a global variable: the symbol, (the thing's name), points to its typeglob and the typeglob for the thing's type (scalar, array, etc.) points to the data. If we had a scalar and an array with the same name their name would point to the same typeglob, but for each type of data the typeglob points to somewhere different and so the scalar's data and the array's data are completely separate and independent, they just happen to have the same name.

Most of the time, only one part of a typeglob is used (yes, it's a bit wasteful). You will by now know that you distinguish between them by using what the authors of the Camel book call a *funny character*. So if we have a scalar called '`line`' we would refer to it in code as `$line`, and if we had an array of the same

name, that would be written, `@line`. Both would point to the same typeglob (which would be called `*line`), but because of the *funny character* (also known as *decoration*) perl won't confuse the two. Of course we might confuse ourselves, so some programmers don't ever use the same name for more than one type of variable.

Every global symbol is in some package's symbol table. To refer to a global symbol we could write the *fully qualified* name, e.g. `$main::line`. If we are in the same package as the symbol we can omit the package name, e.g. `$line` (unless you use the <strict> pragma and then you will have to predeclare the variable using the `vars` pragma). We can also omit the package name if we have imported the symbol into our current package's namespace. If we want to refer to a symbol that is in another package and which we haven't imported we must use the fully qualified name, e.g. `$otherpkg::box`.

Most of the time you do not need to use the fully qualified symbol name because most of the time you will refer to package variables from within the package. This is very like C++ class variables. You can work entirely within package `main::` and never even know you are using a package, nor that the symbols have package names. In a way, this is a pity because you may fail to learn about packages and they are extremely useful.

The exception is when you *import* the variable from another package. This creates an alias for the variable in the *current* package, so that you can access it without using the fully qualified name.

Whilst global variables are useful for sharing data and are necessary in some contexts it is usually wisest to minimise their use and use *lexical variables*, discussed next, instead.

Note that when you create a variable, the low-level business of allocating memory to store the information is handled automatically by Perl. The intepreter keeps track of the chunks of memory to which the pointers are pointing and takes care of undefining variables. When all references to a variable have ceased to exist then the perl garbage collector is free to take back the memory used ready for recycling. However perl almost never returns back memory it has already used to the operating system during the lifetime of the process.

Lexical Variables and Symbols

The symbols for lexical variables (i.e. those declared using the keyword `my`) are the only symbols which do *not* live in a symbol table. Because of this, they are not available from outside the block in which they are declared. There is no typeglob associated with a lexical variable and a lexical variable can refer only to a scalar, an array or a hash.

If you need access to the data from outside the package then you can return it from a subroutine, or you can create a global variable (i.e. one which has a package prefix) which points or refers to it and return that. The pointer or reference must be global so that you can refer to it by a fully qualified name. But just like in C try to avoid having global variables. Using OO methods generally solves this problem, by providing methods to get and set the desired value within the object that can be lexically scoped inside the package and passed by reference.

The phrase ‘‘lexical variable’’ is a bit of a misnomer, we are really talking about ‘‘lexical symbols’’. The data can be referenced by a global symbol too, and in such cases when the lexical symbol goes out of scope the data will still be accessible through the global symbol. This is perfectly legitimate and cannot be

compared to the terrible mistake of taking a pointer to an automatic C variable and returning it from a function--when the pointer is dereferenced there will be a segmentation fault. (Note for C/C++ programmers: having a function return a pointer to an auto variable is a disaster in C or C++; the perl equivalent, returning a reference to a lexical variable created in a function is normal and useful.)

- `my()` vs. `use vars`:

  With use vars(), you are making an entry in the symbol table, and you are telling the compiler that you are going to be referencing that entry without an explicit package name.

  With my(), NO ENTRY IS PUT IN THE SYMBOL TABLE. The compiler figures out `at compile time` which `my()` variables (i.e. lexical variables) are the same as each other, and once you hit execute time you cannot go looking those variables up in the symbol table.

- `my()` vs. `local()`:

  `local()` creates a temporal-limited package-based scalar, array, hash, or glob -- when the scope of definition is exited at runtime, the previous value (if any) is restored. References to such a variable are *also* global... only the value changes. (Aside: that is what causes variable suicide. :)

  `my()` creates a lexically-limited non-package-based scalar, array, or hash -- when the scope of definition is exited at compile-time, the variable ceases to be accessible. Any references to such a variable at runtime turn into unique anonymous variables on each scope exit.

use(), require(), do(), %INC and @INC Explained

The @INC array

@INC is a special Perl variable which is the equivalent of the shell's PATH variable. Whereas PATH contains a list of directories to search for executables, @INC contains a list of directories from which Perl modules and libraries can be loaded.

When you use(), require() or do() a filename or a module, Perl gets a list of directories from the @INC variable and searches them for the file it was requested to load. If the file that you want to load is not located in one of the listed directories, you have to tell Perl where to find the file. You can either provide a path relative to one of the directories in @INC, or you can provide the full path to the file.

The %INC hash

%INC is another special Perl variable that is used to cache the names of the files and the modules that were successfully loaded and compiled by use(), require() or do() statements. Before attempting to load a file or a module with use() or require(), Perl checks whether it's already in the %INC hash. If it's there, the loading and therefore the compilation are not performed at all. Otherwise the file is loaded into memory and an attempt is made to compile it. do() does unconditional loading--no lookup in the %INC hash is made.

If the file is successfully loaded and compiled, a new key-value pair is added to %INC. The key is the name of the file or module as it was passed to the one of the three functions we have just mentioned, and if it was found in any of the @INC directories except "." the value is the full path to it in the file system.

The following examples will make it easier to understand the logic.

First, let's see what are the contents of @INC on my system:

```
% perl -e 'print join "\n", @INC'
/usr/lib/perl5/5.00503/i386-linux
/usr/lib/perl5/5.00503
/usr/lib/perl5/site_perl/5.005/i386-linux
/usr/lib/perl5/site_perl/5.005
.
```

Notice the . (current directory) is the last directory in the list.

Now let's load the module strict.pm and see the contents of %INC:

```
% perl -e 'use strict; print map {"$_ => $INC{$_}\n"} keys %INC'

strict.pm => /usr/lib/perl5/5.00503/strict.pm
```

Since strict.pm was found in */usr/lib/perl5/5.00503/* directory and */usr/lib/perl5/5.00503/* is a part of @INC, %INC includes the full path as the value for the key strict.pm.

Now let's create the simplest module in `/tmp/test.pm`:

```
test.pm
-------
1;
```

It does nothing, but returns a true value when loaded. Now let's load it in different ways:

```
% cd /tmp
% perl -e 'use test; print map {"$_ => $INC{$_}\n"} keys %INC'

test.pm => test.pm
```

Since the file was found relative to `.` (the current directory), the relative path is inserted as the value. If we alter `@INC`, by adding */tmp* to the end:

```
% cd /tmp
% perl -e 'BEGIN{push @INC, "/tmp"} use test; \
print map {"$_ => $INC{$_}\n"} keys %INC'

test.pm => test.pm
```

Here we still get the relative path, since the module was found first relative to `"."`. The directory */tmp* was placed after `.` in the list. If we execute the same code from a different directory, the `"."` directory won't match,

```
% cd /
% perl -e 'BEGIN{push @INC, "/tmp"} use test; \
print map {"$_ => $INC{$_}\n"} keys %INC'

test.pm => /tmp/test.pm
```

so we get the full path. We can also prepend the path with unshift(), so it will be used for matching before `"."` and therefore we will get the full path as well:

```
% cd /tmp
% perl -e 'BEGIN{unshift @INC, "/tmp"} use test; \
print map {"$_ => $INC{$_}\n"} keys %INC'

test.pm => /tmp/test.pm
```

The code:

```
BEGIN{unshift @INC, "/tmp"}
```

can be replaced with the more elegant:

```
use lib "/tmp";
```

Which is almost equivalent to our `BEGIN` block and is the recommended approach.

These approaches to modifying `@INC` can be labor intensive, since if you want to move the script around in the file-system you have to modify the path. This can be painful, for example, when you move your scripts from development to a production server.

There is a module called `FindBin` which solves this problem in the plain Perl world, but unfortunately it won't work under mod_perl, since it's a module and as any module it's loaded only once. So the first script using it will have all the settings correct, but the rest of the scripts will not if located in a different directory from the first.

For the sake of completeness, I'll present this module anyway.

If you use this module, you don't need to write a hard coded path. The following snippet does all the work for you (the file is */tmp/load.pl*):

```
load.pl
-------
#!/usr/bin/perl

use FindBin ();
use lib "$FindBin::Bin";
use test;
print "test.pm => $INC{'test.pm'}\n";
```

In the above example `$FindBin::Bin` is equal to */tmp*. If we move the script somewhere else... e.g. */tmp/x* in the code above `$FindBin::Bin` equals */home/x*.

```
% /tmp/load.pl

test.pm => /tmp/test.pm
```

This is just like `use lib` except that no hard coded path is required.

You can use this workaround to make it work under mod_perl.

```
do 'FindBin.pm';
unshift @INC, "$FindBin::Bin";
require test;
#maybe test::import( ... ) here if need to import stuff
```

This has a slight overhead because it will load from disk and recompile the `FindBin` module on each request. So it may not be worth it.

Modules, Libraries and Program Files

Before we proceed, let's define what we mean by *module*, *library* and *program file*.

- **Libraries**
  These are files which contain Perl subroutines and other code.

When these are used to break up a large program into manageable chunks they don't generally include a package declaration; when they are used as subroutine libraries they often do have a package declaration.

Their last statement returns true, a simple `1;` statement ensures that.

They can be named in any way desired, but generally their extension is *.pl*.

Examples:

```
config.pl
----------
# No package so defaults to main::
$dir = "/home/httpd/cgi-bin";
$cgi = "/cgi-bin";
1;

mysubs.pl
----------
# No package so defaults to main::
sub print_header{
  print "Content-type: text/plain\r\n\r\n";
}
1;

web.pl
------------
package web ;
# Call like this: web::print_with_class('loud',"Don't shout!");
sub print_with_class{
  my( $class, $text ) = @_ ;
  print qq{<span class="$class">$text</span>};
}
1;
```

- **Modules**
  A file which contains perl subroutines and other code.

  It generally declares a package name at the beginning of it.

  Modules are generally used either as function libraries (which *.pl* files are still but less commonly used for), or as object libraries where a module is used to define a class and its methods.

  Its last statement returns true.

  The naming convention requires it to have a *.pm* extension.

  Example:

```
MyModule.pm
-----------
package My::Module;
$My::Module::VERSION = 0.01;
```

```
sub new{ return bless {}, shift;}
END { print "Quitting\n"}
1;
```

- **Program Files**
  Many Perl programs exist as a single file. Under Linux and other Unix-like operating systems the file often has no suffix since the operating system can determine that it is a perl script from the first line (shebang line) or if it's Apache that executes the code, there is a variety of ways to tell how and when the file should be executed. Under Windows a suffix is normally used, for example `.pl` or `.plx`.

  The program file will normally `require()` any libraries and `use()` any modules it requires for execution.

  It will contain Perl code but won't usually have any package names.

  Its last statement may return anything or nothing.

`require()`

`require()` reads a file containing Perl code and compiles it. Before attempting to load the file it looks up the argument in `%INC` to see whether it has already been loaded. If it has, `require()` just returns without doing a thing. Otherwise an attempt will be made to load and compile the file.

`require()` has to find the file it has to load. If the argument is a full path to the file, it just tries to read it. For example:

```
require "/home/httpd/perl/mylibs.pl";
```

If the path is relative, `require()` will attempt to search for the file in all the directories listed in `@INC`. For example:

```
require "mylibs.pl";
```

If there is more than one occurrence of the file with the same name in the directories listed in `@INC` the first occurrence will be used.

The file must return *TRUE* as the last statement to indicate successful execution of any initialization code. Since you never know what changes the file will go through in the future, you cannot be sure that the last statement will always return *TRUE*. That's why the suggestion is to put ‘‘`1;`’’ at the end of file.

Although you should use the real filename for most files, if the file is a module, you may use the following convention instead:

```
require My::Module;
```

This is equal to:

```
require "My/Module.pm";
```

If `require()` fails to load the file, either because it couldn't find the file in question or the code failed to compile, or it didn't return *TRUE*, then the program would die(). To prevent this the `require()` statement can be enclosed into an `eval()` exception-handling block, as in this example:

```
require.pl
----------
#!/usr/bin/perl -w

eval { require "/file/that/does/not/exists"};
if ($@) {
  print "Failed to load, because : $@"
}
print "\nHello\n";
```

When we execute the program:

```
% ./require.pl

Failed to load, because : Can't locate /file/that/does/not/exists in
@INC (@INC contains: /usr/lib/perl5/5.00503/i386-linux
/usr/lib/perl5/5.00503 /usr/lib/perl5/site_perl/5.005/i386-linux
/usr/lib/perl5/site_perl/5.005 .) at require.pl line 3.

Hello
```

We see that the program didn't die(), because *Hello* was printed. This *trick* is useful when you want to check whether a user has some module installed, but if she hasn't it's not critical, perhaps the program can run without this module with reduced functionality.

If we remove the `eval()` part and try again:

```
require.pl
----------
#!/usr/bin/perl -w

require "/file/that/does/not/exists";
print "\nHello\n";

% ./require1.pl

Can't locate /file/that/does/not/exists in @INC (@INC contains:
/usr/lib/perl5/5.00503/i386-linux /usr/lib/perl5/5.00503
/usr/lib/perl5/site_perl/5.005/i386-linux
/usr/lib/perl5/site_perl/5.005 .) at require1.pl line 3.
```

The program just die()s in the last example, which is what you want in most cases.

For more information refer to the perlfunc manpage.

use()

use(), just like require(), loads and compiles files containing Perl code, but it works with modules only. The only way to pass a module to load is by its module name and not its filename. If the module is located in *MyCode.pm*, the correct way to `use()` it is:

```
use MyCode
```

and not:

```
use "MyCode.pm"
```

`use()` translates the passed argument into a file name replacing `::` with the operating system's path separator (normally `/`) and appending *.pm* at the end. So `My::Module` becomes *My/Module.pm*.

`use()` is exactly equivalent to:

```
BEGIN { require Module; Module->import(LIST); }
```

Internally it calls `require()` to do the loading and compilation chores. When `require()` finishes its job, `import()` is called unless `()` is the second argument. The following pairs are equivalent:

```
use MyModule;
BEGIN {require MyModule; MyModule->import; }

use MyModule qw(foo bar);
BEGIN {require MyModule; MyModule->import("foo","bar"); }

use MyModule ();
BEGIN {require MyModule; }
```

The first pair exports the default tags. This happens if the module sets `@EXPORT` to a list of tags to be exported by default. The module's manpage normally describes what tags are exported by default.

The second pair exports only the tags passed as arguments.

The third pair describes the case where the caller does not want any symbols to be imported.

`import()` is not a builtin function, it's just an ordinary static method call into the ''`MyModule`'' package to tell the module to import the list of features back into the current package. See the Exporter manpage for more information.

When you write your own modules, always remember that it's better to use `@EXPORT_OK` instead of `@EXPORT`, since the former doesn't export symbols unless it was asked to. Exports pollute the namespace of the module user. Also avoid short or common symbol names to reduce the risk of name clashes.

When functions and variables aren't exported you can still access them using their full names, like `$My::Module::bar` or `$My::Module::foo()`. By convention you can use a leading underscore on names to informally indicate that they are *internal* and not for public use.

There's a corresponding ''`no`'' command that un-imports symbols imported by `use`, i.e., it calls `Module->unimport(LIST)` instead of `import()`.

do()

While do() behaves almost identically to require(), it reloads the file unconditionally. It doesn't check %INC to see whether the file was already loaded.

If do() cannot read the file, it returns undef and sets $! to report the error. If do() can read the file but cannot compile it, it returns undef and puts an error message in $@. If the file is successfully compiled, do() returns the value of the last expression evaluated.

References

- An article by Mark-Jason Dominus about how Perl handles variables and namespaces, and the difference between `use vars()` and `my()` - http://www.plover.com/~mjd/perl/FAQs/Namespaces.html .

- For an indepth explanation of Perl data types see the Chapters 3 and 6 in the book *''Advanced Perl Programming''* by Sriram Srinivasan.

  And of course the *''Programming Perl''* by L.Wall, T. Christiansen and J.Orwant (also known as the *''Camel''* book, named after the camel picture on the cover of the book). Look at the Chapters 10, 11 and 21.

- The *Exporter*, *perlvar*, *perlmod* and *perlmodlib* man pages.