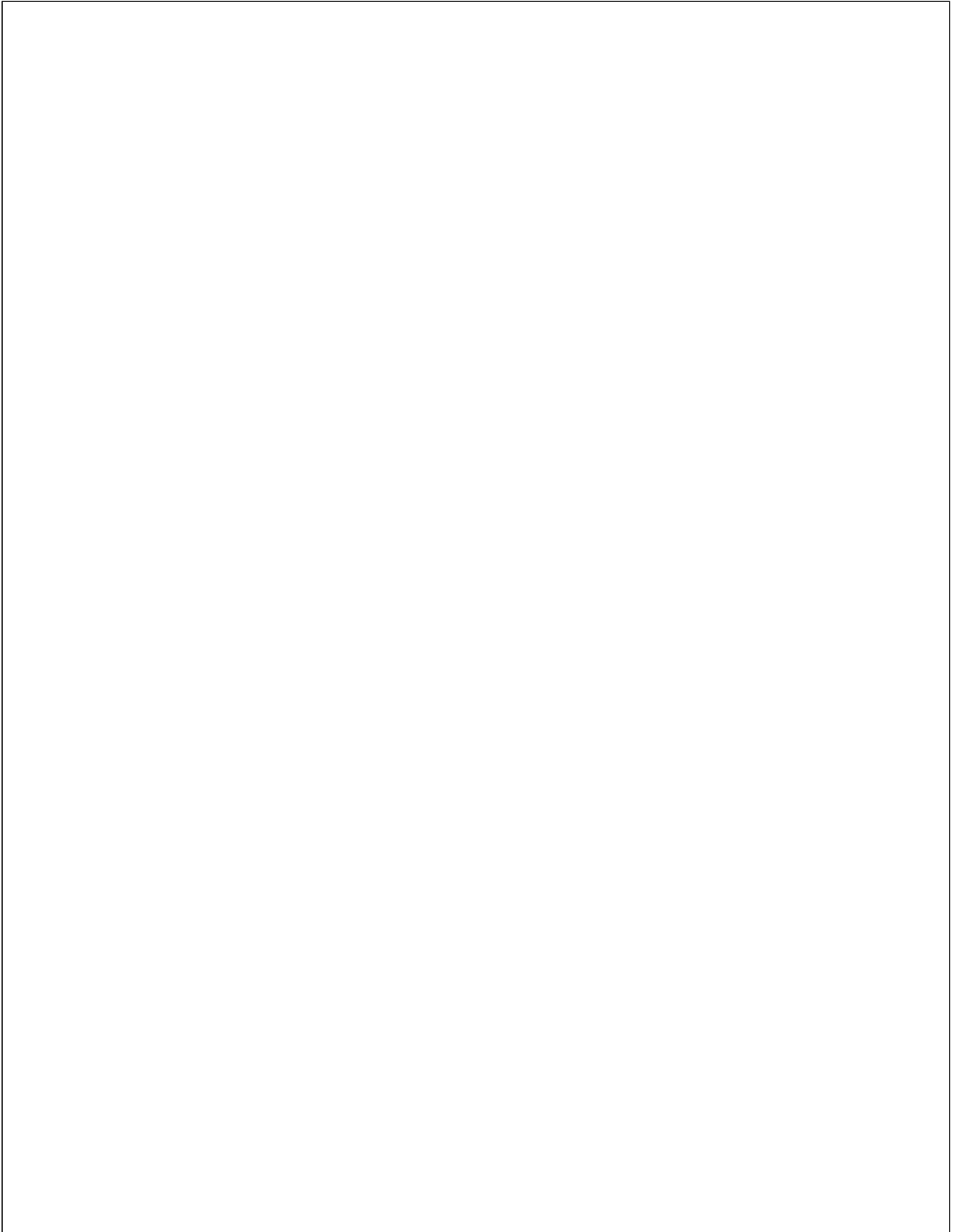**The ApacheCon 2000
March 8, 2000
Orlando, Florida**

Tutorial:
Getting Started with mod_perl
(Part II of II)

By **Stas Bekman**
Internet and Intranet programmer
http://singlesheaven.com/stas/
<sbekman@iname.com>

This document is originally written in **POD**, converted to **HTML** by **pod2html** utility and then to **PostScript** by **html2ps** utility.

(you will find a Table of Contents at the end)

1 Getting Started Fast

1.1 mod_perl in Four Slides

Each tutorial will concentrate on different aspects of running a mod_perl server and mod_perl programming. In case you don't know how to get started with it, or you think it's a difficult task, these slides will take away any worries you might have had when you came to this tutorial.

In just four slides you will be able to install and configure a mod_perl server. And, of course, to write new code and reuse the existing code under mod_perl.

The four slides (sections) are:

- Installation

- Configuration

- The ''mod_perl rules'' Apache::Registry Scripts

- The ''mod_perl rules'' Apache Perl Module

1.2 What is mod_perl?

But before we go any further, there is a chance that you don't know what mod_perl is. So let's make a little introduction to mod_perl.

Everybody knows that Perl scripts running under mod_cgi have numerous shortcomings. There are many of them, but code recompilation and Perl interpreter loading overhead at each request is the hardest one to overcome.

Among various attempts to improve on mod_cgi's shortcomings, mod_perl has proved to be one of the better ones and has been widely adopted by CGI developers. According to the http://perl.apache.org/netcraft/ about 412000 hosts use mod_perl. Doug MacEachern fathered the core code of this Apache module and licensed it under the ''Artistic License'' as Perl itself.

mod_perl does away with mod_cgi's forking by reusing the existing child processes. In this new model, the child process doesn't exit anymore when it has processed a request. The Perl interpreter is loaded only once, when the process is started. Since the interpreter is persistent throughout the process' lifetime, all code is loaded and compiled only once, the first time it is seen. This makes all subsequent requests run much faster because everything is already loaded and compiled. Response processing is now reduced to running your code. This improves response times by a factor of 10 to 100, depending on the code being executed.

Doug didn't stop here, he went and extended mod_cgi's functionality by adding a complete Perl API to the Apache core. This makes it possible to write a complete Apache module in Perl, a feat that used to require coding in C. From then on mod_perl enabled the programmer to handle all phases of request processing in Perl.

The new Perl API also allows complete server configuration in Perl. This has which made the lives of many server administrators much easier, as they could now benefit from dynamically generating the configuration, freed from hunting for bugs in huge configuration files full of similar directives for virtual hosts and the like.

To provide backwards compatibility for plain CGI scripts that used to be run under mod_cgi, while still benefiting from a preloaded perl and modules, a few special handlers were written, each allowing a different level of proximity to pure mod_perl functionality. Some take full advantage of mod_perl, while others only a partial one.

mod_perl embeds a copy of the Perl interpreter into the Apache httpd executable, providing complete access to Perl functionality within Apache. This enables a set of mod_perl-specific configuration directives, all of which start with the string Perl*. Most, but not all, of these directives are used to specify handlers for various phases of the request.

It might occur to you that sticking a large executable (Perl) into another large executable (Apache) makes a very, very large program. mod_perl certainly makes httpd significantly bigger and you will need more RAM on your production server to be able to run many mod_perl processes, but in reality the situation is different. Since mod_perl processes requests much faster, the number of the processes needed to handle the same request rate is much lower relative to the mod_cgi approach. Generally you need slightly more memory available, and the speed improvements you will see are well worth every megabyte of memory you can add.

Now let's get back to the *All-In-Four-Slides*...

1.3 Installation

Did you know that it takes about 10 minutes to build and install a mod_perl enabled Apache server on a computer with a pretty average processor and a decent amount of system memory? It goes like this:

```
% cd /usr/src
% lwp-download http://www.apache.org/dist/apache_x.x.x.tar.gz
% lwp-download http://perl.apache.org/dist/mod_perl-x.xx.tar.gz
% tar xzvf apache_x.x.x.tar.gz
% tar xzvf mod_perl-x.xx.tar.gz
% cd mod_perl-x.xx
% perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
% make && make test && make install
% cd ../apache_x.x.x
% make install
```

That's all!

- Of course you must replace *x.x.x* with the actual version numbers of the mod_perl and Apache releases that you use.

- The GNU `tar` utility knows how to uncompress a gzipped tar archive (use the `z` option).

All that's left is to add a few configuration lines to a *httpd.conf*, an Apache configuration file, start the server and enjoy mod_perl.

1.4 Configuration

Add the following to the configuration file *httpd.conf*:

```
   # for Apache::Registry mode
Alias /perl/ /home/httpd/perl/

PerlModule Apache::Registry
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  allow from all
  PerlSendHeader On
</Location>
```

This configuration causes every URI starting with */perl* to be handled by the Apache mod_perl module. It will use the handler from the Perl module `Apache::Registry`.

1.5 The "mod_perl rules" Apache::Registry Scripts

You can write plain perl/CGI scripts just as under mod_cgi:

```
mod_perl_rules1.pl
------------------
print "Content-type: text/plain\r\n\r\n";
print "mod_perl rules!\n";
```

Of course you can write them in the Apache Perl API:

```
mod_perl_rules2.pl
------------------
my $r = shift;
$r->send_http_header('text/plain');
$r->print("mod_perl rules!\n");
```

Save both files under the */home/httpd/perl* directory, make them executable and readable by server, and issue these requests using your favorite browser:

```
http://localhost/perl/mod_perl_rules1.pl
http://localhost/perl/mod_perl_rules2.pl
```

In both cases you will see on the following response:

```
  mod_perl rules!
```

1.6 The "mod_perl rules" Apache Perl Module

To create an Apache Perl module, all you have to do is to wrap the code into a `handler` subroutine and return the status to the server.

```
ModPerl/Rules.pm
----------------
use Apache::Constants;

sub handler{
  my $r = shift;
  $r->send_http_header('text/plain');
  print "mod_perl rules!\n";
  return OK;
}
```

Create a directory called *ModPerl* under one of the directories in `@INC`, and put *Rules.pm* into it. Then add the following snippet to *httpd.conf*:

```
PerlModule ModPerl::Rules
<Location /mod_perl_rules>
  SetHandler perl-script
  PerlHandler ModPerl::Rules
</Location>
```

Now you can issue a request to:

  http://localhost/perl/mod_perl_rules

and just as with our *mod_perl_rules.pl* scripts you will see:

```
  mod_perl rules!
```

as the response.

1.7 Is That All I Need To Know About mod_perl?

Definitely not!

These slides are intended to show you that you can install and start using a mod_perl server within 30 minutes of downloading the sources.

There is much more to mod_perl than this, you will need to plan your study around the projects you want to implement. Fortunately, there are many resources and lots of help freely available to you.

At the end of each titorial you will find a chapter describing the available resources and pointers to them.

;o)

2 Perl Reference

2.1 What we will learn in this chapter

- Tracing Warnings Reports

- `my()` Scoped Variable in Nested Subroutines

- When You Cannot Get Rid of The Inner Subroutine

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

- Using Global Variables and Sharing Them Between Modules/Packages

- The Scope of the Special Perl Variables

- Compiled Regular Expressions

- perldoc's Rarely Known But Very Useful Options

2.2 Tracing Warnings Reports

Sometimes it's very hard to understand what a warning is complaining about. You see the source code, but you cannot understand why some specific snippet produces that warning. The mystery often results from the fact that the code can be called from different places if it's located inside a subroutine.

Here is an example:

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

correct();
incorrect();

sub correct{
  print_value("Perl");
}

sub incorrect{
  print_value();
}

sub print_value{
  my $var = shift;
  print "My value is $var\n";
}
```

In the code above, `print_value()` prints the passed value, `correct()` passes the value to print and in `incorrect()` we forgot to pass it. When we run the script:

```
  % ./warnings.pl
```

we get the warning:

```
  Use of uninitialized value at ./warnings.pl line 16.
```

Perl complains about an undefined variable $var at the line that attempts to print its value:

```
  print "My value is $var\n";
```

But how do we know why it is undefined? The reason here obviously is that the calling function didn't pass the argument. But how do we know who was the caller? In our example there are two possible callers, in the general case there can be many of them, perhaps located in other files.

We can use the caller() function, which tells who has called us, but even that might not be enough: it's possible to have a longer sequence of called subroutines, and not just two. For example, here it is sub third() which is at fault, and putting sub caller() in sub second() would not help us very much:

```
  sub third{
    second();
  }
  sub second{
    my $var = shift;
    first($var);
  }
  sub first{
    my $var = shift;
   print "Var = $var\n"
  }
```

The solution is quite simple. What we need is a full calls stack trace to the call that triggered the warning.

The Carp module comes to our aid with its cluck() function. Let's modify the script by adding a couple of lines. The rest of the script is unchanged.

```
  warnings2.pl
  -----------
  #!/usr/bin/perl -w

  use Carp ();
  local $SIG{__WARN__} = \&Carp::cluck;

  correct();
  incorrect();

  sub correct{
    print_value("Perl");
  }

  sub incorrect{
    print_value();
  }
```

```
  sub print_value{
    my $var = shift;
    print "My value is $var\n";
  }
```

Now when we execute it, we see:

```
  Use of uninitialized value at ./warnings2.pl line 19.
    main::print_value() called at ./warnings2.pl line 14
    main::incorrect() called at ./warnings2.pl line 7
```

Take a moment to understand the calls stack trace. The deepest calls are printed first. So the second line tells us that the warning was triggered in `print_value()`; the third, that `print_value()` was called by `incorrect()` subroutine.

```
  script => incorrect() => print_value()
```

We go into `incorrect()` and indeed see that we forgot to pass the variable. Of course when you write a subroutine like `print_value` it would be a good idea to check the passed arguments before starting execution. We omitted that step to contrive an easily debugged example.

Sure, you say, I could find that problem by simple inspection of the code!

Well, you're right. But I promise you that your task would be quite complicated and time consuming if your code has some thousands of lines. In addition, under mod_perl, certain uses of the `eval` operator and ''here documents'' are known to throw off Perl's line numbering, so the messages reporting warnings and errors can have incorrect line numbers.

Getting the trace helps a lot.

2.3 my() Scoped Variable in Nested Subroutines

Before we proceed let's make the assumption that we want to develop the code under the `strict` pragma. We will use lexically scoped variables (with help of the `my()` operator) whenever it's possible.

2.3.1 The Poison

Let's look at this code:

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

  use strict;

  sub print_power_of_2 {
    my $x = shift;

    sub power_of_2 {
      return $x ** 2;
    }
```

```
   my $result = power_of_2();
   print "$x^2 = $result\n";
}

print_power_of_2(5);
print_power_of_2(6);
```

Don't let the weird subroutine names to fool you, the `print_power_of_2()` subroutine should print
the square of the passed number. Let's run the code and see whether it works:

```
% ./nested.pl

5^2 = 25
6^2 = 25
```

Ouch, something is wrong. May be there is a bug in Perl and it doesn't work correctly with number 6?
Let's try again using the 5 and 7:

```
print_power_of_2(5);
print_power_of_2(7);
```

And run it:

```
% ./nested.pl

5^2 = 25
7^2 = 25
```

Wow, does it works only for 5? How about using 3 and 5:

```
print_power_of_2(3);
print_power_of_2(5);
```

and the result is:

```
% ./nested.pl

3^2 = 9
5^2 = 9
```

Now we start to understand--only the first call to the `print_power_of_2()` function works correctly.
Which makes us think that our code has some kind of memory for results of the first execution, or it
ignores the arguments in subsequent executions.

2.3.2 The Diagnosis

Let's follow the guidelines and use the `-w` flag. Now execute the code:

```
% ./nested.pl

Variable "$x" will not stay shared at ./nested.pl line 9.
5^2 = 25
6^2 = 25
```

We have never seen such a warning message before and we don't quite understand what it means. The `diagnostics` pragma will certainly help us. Let's prepend this pragma before the `strict` pragma in our code:

```
#!/usr/bin/perl -w

use diagnostics;
use strict;
```

And execute it:

```
% ./nested.pl

Variable "$x" will not stay shared at ./nested.pl line 10 (#1)

  (W) An inner (nested) named subroutine is referencing a lexical
  variable defined in an outer subroutine.

  When the inner subroutine is called, it will probably see the value of
  the outer subroutine's variable as it was before and during the
  *first* call to the outer subroutine; in this case, after the first
  call to the outer subroutine is complete, the inner and outer
  subroutines will no longer share a common value for the variable.  In
  other words, the variable will no longer be shared.

  Furthermore, if the outer subroutine is anonymous and references a
  lexical variable outside itself, then the outer and inner subroutines
  will never share the given variable.

  This problem can usually be solved by making the inner subroutine
  anonymous, using the sub {} syntax.  When inner anonymous subs that
  reference variables in outer subroutines are called or referenced,
  they are automatically rebound to the current values of such
  variables.

5^2 = 25
6^2 = 25
```

Well, now everything is clear. We have the **inner** subroutine `power_of_2()` and the **outer** subroutine `print_power_of_2()` in our code.

When the inner `power_of_2()` subroutine is called for the first time, it sees the value of the outer `print_power_of_2()` subroutine's `$x` variable. On subsequent calls the `$x` variable won't be updated, no matter what the value of it in the outer subroutine. There are two copies of the `$x` variable, no longer a single one shared by the two routines.

2.3.3 The Remedy

The `diagnostics` pragma suggests that the problem can be solved by making the inner subroutine anonymous.

An anonymous subroutine can act as a *closure* with respect to lexically scoped variables. Basically this means that if you define a subroutine in a particular **lexical** context at a particular moment, then it will run in that same context later, even if called from outside that context. The upshot of this is that when the subroutine **runs**, you get the same copies of the lexically scoped variables which were visible when the subroutine was **defined**. So you can pass arguments to a function when you define it, as well as when you invoke it.

Let's rewrite the code to use this technique:

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

use strict;

sub print_power_of_2 {
  my $x = shift;

  my $func_ref = sub {
    return $x ** 2;
  };

  my $result = &$func_ref();
  print "$x^2 = $result\n";
}

print_power_of_2(5);
print_power_of_2(6);
```

Now `$func_ref` contains a reference to an anonymous function, which we later use when we need to get the power of two. (In Perl, a function is the same thing as a subroutine.) Since it is anonymous, the function will automatically be rebound to the new value of the outer scoped variable $x, and the results will now be as expected.

Let's verify:

```
% ./anonymous.pl

5^2 = 25
6^2 = 36
```

Indeed, *anonymous.pl* worked as we expected.

2.4 When You Cannot Get Rid of The Inner Subroutine

First you might wonder, why in the world will someone need to define an inner subroutine? Well, for example to reduce some of Perl's script startup overhead you might decide to write a daemon that will compile the scripts and modules only once, and cache the pre-compiled code in memory. When some script is to be executed, you just tell the daemon the name of the script to run and it will do the rest and do it much faster.

Seems like an easy task, and it is. The only problem is once the script is compiled, how do you execute it? Or let's put it the other way: after it was executed for the first time and it stays compiled in the daemon memory, how do you call it again? If you could get all developers to code the scripts so each has a subroutine called `run()` that will actually execute the code in the script then you have half of the problem solved.

But how does the daemon know to refer to some specific script if they all run in the `main::` name space? One solution might be to ask the developers to declare a package in each and every script, and for the package name to be derived from the script name. However, since there is chance that there will be more than one script with the same name but residing in different directories, then in order to prevent name-space collisions the directory has to be a part of the package name too. And don't forget that script may be moved from one directory to another, so you will have to make sure that the package name is corrected every time the script gets moved.

But why enforce these strange rules on developers, when we can arrange for our daemon to do this work? For every script that daemon is about to execute for the first time, it should be wrapped inside the package whose name is constructed from the mangled path to the script and a subroutine called `run()`. For example if the daemon is about to execute the script */tmp/hello.pl*:

```
hello.pl
--------
#!/usr/bin/perl
print "Hello\n";
```

Prior to running it, the daemon will change the code to be:

```
wrapped_hello.pl
----------------
package cache::tmp::hello_2epl;

sub run{
  #!/usr/bin/perl
  print "Hello\n";
}
```

The package name is constructed from the prefix `cache::`, each directory separation slash is replaced with `::`, and non alphanumeric characters are encoded so that for example `.` (a dot) becomes `_2e` (an underscore followed by the ASCII code for a dot in hex representation).

```
 % perl -e 'printf "%x",ord(".")'
```

prints: 2e. The underscore is the same you see in URL encoding where % character is used instead (%2E), but since % has a special meaning in Perl (prefix of hash variable) it couldn't be used.

Now when the daemon is requested to execute the script */tmp/hello.pl*, all it has to do is to build the package name as before based on the location of the script and call its `run()` subroutine:

```
  use cache::tmp::hello_2epl;
  cache::tmp::hello_2epl::run();
```

We have just written a partial prototype of the daemon we desired. The only method now remaining undefined is how to pass the path to the script to the daemon. This detail is left to the reader as an exercise.

If you are familiar with the `Apache::Registry` module, you know that it works in almost the same way. It uses a different package prefix and the generic function is called `handler()` and not `run()`. The scripts to run are passed through the HTTP protocol's headers.

Now you understand that there are cases where your normal subroutines can become inner, since if your script was a simple:

```
  simple.pl
  ---------
  #!/usr/bin/perl
  sub hello { print "Hello" }
  hello();
```

Wrapped into a `run()` subroutine it becomes:

```
  simple.pl
  ---------
  package cache::simple_2epl;

  sub run{
    #!/usr/bin/perl
    sub hello { print "Hello" }
    hello();
  }
```

Therefore, `hello()` is an inner subroutine and if you have used `my()` scoped variables defined and altered outside and used inside `hello()`, it won't work as you expect starting from the second call, as was explained in the previous section.

2.4.1 Remedies for Inner Subroutines

First of all there is nothing to worry about, as long as you don't forget to turn the warnings On. If you do happen to have the ''my() Scoped Variable in Nested Subroutines'' problem, Perl will always alert you.

Given that you have a script that has this problem, what are the ways to solve it? There are many of them and we will discuss some of them here.

We will use the following code to show the different solutions.

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

use strict;

for (1..3){
  print "run: [time $_]\n";
  run();
}

sub run {

  my $counter = 0;

  increment_counter();
  increment_counter();

  sub increment_counter{
    $counter++;
    print "Counter is equal to $counter !\n";
  }

} # end of sub run
```

This code executes the run() subroutine three times, which in turn initializes the $counter variable to 0, every time it executed and then calls the inner subroutine increment_counter() twice. Sub increment_counter() prints $counter's value after incrementing it. One might expect to see the following output:

```
run: [time 1]
Counter is equal to 1 !
Counter is equal to 2 !
run: [time 2]
Counter is equal to 1 !
Counter is equal to 2 !
run: [time 3]
Counter is equal to 1 !
Counter is equal to 2 !
```

But as we have already learned from the previous sections, this is not what we are going to see. Indeed, when we run the script we see:

```
% ./multirun.pl
```

```
  Variable "$counter" will not stay shared at ./nested.pl line 18.
  run: [time 1]
  Counter is equal to 1 !
  Counter is equal to 2 !
  run: [time 2]
  Counter is equal to 3 !
  Counter is equal to 4 !
  run: [time 3]
  Counter is equal to 5 !
  Counter is equal to 6 !
```

Obviously, the $counter variable is not reinitialized on each execution of run(). It retains its value from the previous execution, and sub increment_counter() increments that.

One of the workarounds is to use globally declared variables, with the vars pragma.

```
  multirun1.pl
  -----------
  #!/usr/bin/perl -w

  use strict;
  use vars qw($counter);

  for (1..3){
    print "run: [time $_]\n";
    run();
  }

  sub run {

    $counter = 0;

    increment_counter();
    increment_counter();

    sub increment_counter{
      $counter++;
      print "Counter is equal to $counter !\n";
    }

  } # end of sub run
```

If you run this and the other solutions offered below, the expected output will be generated:

```
  % ./multirun1.pl

  run: [time 1]
  Counter is equal to 1 !
  Counter is equal to 2 !
  run: [time 2]
  Counter is equal to 1 !
  Counter is equal to 2 !
  run: [time 3]
  Counter is equal to 1 !
  Counter is equal to 2 !
```

By the way, the warning we saw before has gone, and so has the problem, since there is no my() (lexi-cally defined) variable used in the nested subroutine.

Another approach is to use fully qualified variables. This is better, since less memory will be used, but it adds a typing overhead:

```
multirun2.pl
-----------
#!/usr/bin/perl -w

use strict;

for (1..3){
  print "run: [time $_]\n";
  run();
}

sub run {

  $main::counter = 0;

  increment_counter();
  increment_counter();

  sub increment_counter{
    $main::counter++;
    print "Counter is equal to $main::counter !\n";
  }

} # end of sub run
```

You can also pass the variable to the subroutine by value and make the subroutine return it after it was updated. This adds time and memory overheads, so it may not be good idea if the variable can be very large, or if speed of execution is an issue.

Don't rely on the fact that the variable is small during the development of the application, it can grow quite big in situations you don't expect. For example, a very simple HTML form text entry field can return a few megabytes of data if one of your users is bored and wants to test how good is your code. It's not uncommon to see users Copy-and-Paste 10Mb core dump files into a form's text fields and then submit it for your script to process.

```
multirun3.pl
-----------
#!/usr/bin/perl -w

use strict;

for (1..3){
  print "run: [time $_]\n";
  run();
}

sub run {
```

```
   my $counter = 0;

   $counter = increment_counter($counter);
   $counter = increment_counter($counter);

   sub increment_counter{
     my $counter = shift || 0 ;

     $counter++;
     print "Counter is equal to $counter !\n";

     return $counter;
   }

 } # end of sub run
```

Finally, you can use references to do the job. The version of increment_counter() below accepts a reference to the $counter variable and increments its value after first dereferencing it. When you use a reference, the variable you use inside the function is physically the same bit of memory as the one outside the function. This technique is often used to enable a called function to modify variables in a calling function.

```
 multirun4.pl
 -----------
 #!/usr/bin/perl -w

 use strict;

 for (1..3){
   print "run: [time $_]\n";
   run();
 }

 sub run {

   my $counter = 0;

   increment_counter(\$counter);
   increment_counter(\$counter);

   sub increment_counter{
     my $r_counter = shift || 0;

     $$r_counter++;
     print "Counter is equal to $$r_counter !\n";
   }

 } # end of sub run
```

Here is yet another and more obscure reference usage. We modify the value of $counter inside the subroutine by using the fact that variables in @_ are aliases for the actual scalar parameters. Thus if you called a function with two arguments, those would be stored in $_[0] and $_[1]. In particular, if an element $_[0] is updated, the corresponding argument is updated (or an error occurs if it is not updatable).

```
  multirun5.pl
  -----------
  #!/usr/bin/perl -w

  use strict;

  for (1..3){
    print "run: [time $_]\n";
    run();
  }

  sub run {

    my $counter = 0;

    increment_counter($counter);
    increment_counter($counter);

    sub increment_counter{
      $_[0]++;
      print "Counter is equal to $_[0] !\n";
    }

  } # end of sub run
```

Now you have at least five workarounds to choose from.

For more information please refer to perlref and perlsub manpages.

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

2.5.1 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.

2.5.2 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() functions. Before attempting to load a file or a module, 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 compiled it.

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 executes the BEGIN block above exactly.

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 doesn't work correctly under mod_perl.

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
```

Just like with `use lib` but no hard coded path required.

As I've mentioned earlier, `FindBin` will not work in the mod_perl environment, 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.

2.5.3 Modules, Libraries and Files

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

- **The Library or the File**

  A file which contains perl subroutines and other code.

  It generally doesn't include a package declaration.

  Its last statement returns true.

  It can be named in any way desired, but generally its extension is *.pl* or *.ph*.

  Examples:

  ```
  config.pl
  ----------
  $dir = "/home/httpd/cgi-bin";
  $cgi = "/cgi-bin";
  1;

  mysubs.pl
  ----------
  sub print_header{
    print "Content-type: text/plain\r\n\r\n";
  }
  1;
  ```

- **the Module**

  A file which contains perl subroutines and other code.

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

  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;
```

2.5.4 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() 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.

2.5.5 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 `/` and appending *.pm* at the end. So `My::Module` becomes *My/Module.pm*.

`use()` is exactly equivalent to:

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

   use MyModule qw(foo bar);
   BEGIN {require MyModule; import MyModule ("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 manpage generally describes what modules are exported by default.

The second pair exports all the tags passed as arguments. No default tags are exported unless explicitly told to.

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 `unimport Module LIST` instead of `import()`.

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

2.6 Using Global Variables and Sharing Them Between Modules/Packages

2.6.1 Making Variables Global

When you first wrote `$x` in your code you created a global variable. It is visible everywhere in the file you have used it. If you defined it inside a package, it is visible inside the package. But it will work only if you do not use `strict` pragma and you **HAVE** to use this pragma if you want to run your scripts under mod_perl. Read the `strict` pragma manpage to find out why.

2.6.2 Making Variables Global With strict Pragma On

First you use :

```
  use strict;
```

Then you use:

```
 use vars qw($scalar %hash @array);
```

Starting from this moment the variables are global only in the package where you defined them. If you want to share global variables between packages, here is what you can do.

2.6.3 Using Exporter.pm to Share Global Variables

Assume that you want to share the `CGI.pm` object (I will use `$q`) between your modules. For example, you create it in `script.pl`, but you want it to be visible in `My::HTML`. First, you make `$q` global.

```
  script.pl:
  ----------------
  use vars qw($q);
  use CGI;
  use lib qw(.);
  use My::HTML qw($q); # My/HTML.pm is in the same dir as script.pl
  $q = new CGI;

  My::HTML::printmyheader();
  ----------------
```

Note that we have imported $q from My::HTML. And My::HTML does the export of $q:

```
My/HTML.pm
----------------
package My::HTML;
use strict;

BEGIN {
  use Exporter ();

  @My::HTML::ISA         = qw(Exporter);
  @My::HTML::EXPORT      = qw();
  @My::HTML::EXPORT_OK   = qw($q);

}

use vars qw($q);

sub printmyheader{
  # Whatever you want to do with $q... e.g.
  print $q->header();
}
1;
------------------
```

So the $q is shared between the My::HTML package and script.pl. It will work vice versa as well, if you create the object in My::HTML but use it in script.pl. You have true sharing, since if you change $q in script.pl, it will be changed in My::HTML as well.

What if you need to share $q between more than two packages? For example you want My::Doc to share $q as well.

You leave My::HTML untouched, and modify *script.pl* to include:

```
 use My::Doc qw($q);
```

Then you write My::Doc exactly like My::HTML - except of course that the content is different :).

One possible pitfall is when you want to use My::Doc in both My::HTML and *script.pl*. Only if you add

```
  use My::Doc qw($q);
```

into My::HTML will $q be shared. Otherwise My::Doc will not share $q any more. To make things clear here is the code:

```
  script.pl:
  ----------------
  use vars qw($q);
  use CGI;
  use lib qw(.);
  use My::HTML qw($q); # My/HTML.pm is in the same dir as script.pl
  use My::Doc  qw($q); # Ditto
  $q = new CGI;

  My::HTML::printmyheader();
  ----------------

  My/HTML.pm
  ----------------
  package My::HTML;
  use strict;

  BEGIN {
    use Exporter ();

    @My::HTML::ISA          = qw(Exporter);
    @My::HTML::EXPORT       = qw();
    @My::HTML::EXPORT_OK    = qw($q);


  }

  use vars     qw($q);
  use My::Doc  qw($q);

  sub printmyheader{
    # Whatever you want to do with $q... e.g.
    print $q->header();

    My::Doc::printtitle('Guide');
  }
  1;
  ------------------

  My/Doc.pm
  ----------------
  package My::Doc;
  use strict;

  BEGIN {
    use Exporter ();

    @My::Doc::ISA          = qw(Exporter);
    @My::Doc::EXPORT       = qw();
    @My::Doc::EXPORT_OK    = qw($q);


  }

  use vars qw($q);

  sub printtitle{
    my $title = shift || 'None';
```

```
   print $q->h1($title);
}
1;
-------------------
```

2.6.4 Using the Perl Aliasing Feature to Share Global Variables

As the title says you can import a variable into a script or module without using `Exporter.pm`. I have found it useful to keep all the configuration variables in one module `My::Config`. But then I have to export all the variables in order to use them in other modules, which is bad for two reasons: polluting other packages' name spaces with extra tags which increase the memory requirements; and adding the overhead of keeping track of what variables should be exported from the configuration module and what imported, for some particular package. I solve this problem by keeping all the variables in one hash `%c` and exporting that. Here is an example of `My::Config`:

```
package My::Config;
use strict;
use vars qw(%c);
%c = (
  # All the configs go here
  scalar_var => 5,

  array_var  => [
                  foo,
                  bar,
                 ],

  hash_var   => {
                  foo => 'Foo',
                  bar => 'BARRR',
                 },
);
1;
```

Now in packages that want to use the configuration variables I have either to use the fully qualified names like `$My::Config::test`, which I dislike or import them as described in the previous section. But hey, since we have only one variable to handle, we can make things even simpler and save the loading of the `Exporter.pm` package. We will use the Perl aliasing feature for exporting and saving the keystrokes:

```
package My::HTML;
use strict;
use lib qw(.);
  # Global Configuration now aliased to global %c
use My::Config (); # My/Config.pm in the same dir as script.pl
use vars qw(%c);
*c = \%My::Config::c;

  # Now you can access the variables from the My::Config
print $c{scalar_val};
print $c{array_val}[0];
print $c{hash_val}{foo};
```

Of course $c is global everywhere you use it as described above, and if you change it somewhere it will affect any other packages you have aliased $My::Config::c to.

Note that aliases work either with global or local() vars - you cannot write:

```
my *c = \%My::Config::c;
```

Which is an error. But you can write:

```
local *c = \%My::Config::c;
```

For more information about aliasing, refer to the Camel book, second edition, pages 51-52.

2.7 The Scope of the Special Perl Variables

Special Perl variables like $| (buffering), $^T (time), $^W (warnings), $/ (input record separator), $\ (output record separator) and many more are all global variables. This means that you cannot scope them with my(). Only local() is permitted to do that. Since the child server doesn't usually exit, if in one of your scripts you modify a global variable it will be changed for the rest of the process' life and will affect all the scripts executed by the same process.

We will demonstrate the case on the input record separator variable. If you undefine this variable, a diamond operator will suck in the whole file at once if you have enough memory. Remembering this you should never write code like the example below.

```
$/ = undef;
open IN, "file" ....
  # slurp it all into a variable
$all_the_file = <IN>;
```

The proper way is to have a local() keyword before the special variable is changed, like this:

```
local $/ = undef;
open IN, "file" ....
  # slurp it all inside a variable
$all_the_file = <IN>;
```

But there is a catch. local() will propagate the changed value to any of the code below it. The modified value will be in effect until the script terminates, unless it is changed again somewhere else in the script.

A cleaner approach is to enclose the whole of the code that is affected by the modified variable in a block, like this:

```
{
  local $/ = undef;
  open IN, "file" ....
    # slurp it all inside a variable
  $all_the_file = <IN>;
}
```

That way when Perl leaves the block it restores the original value of the `$/` variable, and you don't need to worry elsewhere in your program about its value being changed here.

2.8 Compiled Regular Expressions

When using a regular expression that contains an interpolated Perl variable, if it is known that the variable (or variables) will not change during the execution of the program, a standard optimization technique is to add the `/o` modifier to the regexp pattern. This directs the compiler to build the internal table once, for the entire lifetime of the script, rather than every time the pattern is executed. Consider:

```
my $pat = '^foo$'; # likely to be input from an HTML form field
foreach( @list ) {
  print if /$pat/o;
}
```

This is usually a big win in loops over lists, or when using `grep()` or `map()` operators.

In long-lived mod_perl scripts, however, the variable can change according to the invocation and this can pose a problem. The first invocation of a fresh httpd child will compile the regex and perform the search correctly. However, all subsequent uses by that child will continue to match the original pattern, regardless of the current contents of the Perl variables the pattern is supposed to depend on. Your script will appear to be broken.

There are two solutions to this problem:

The first is to use `eval q//`, to force the code to be evaluated each time. Just make sure that the eval block covers the entire loop of processing, and not just the pattern match itself.

The above code fragment would be rewritten as:

```
my $pat = '^foo$';
eval q{
  foreach( @list ) {
    print if /$pat/o;
  }
}
```

Just saying:

```
foreach( @list ) {
  eval q{ print if /$pat/o; };
}
```

is going to be a horribly expensive proposition.

You can use this approach if you require more than one pattern match operator in a given section of code. If the section contains only one operator (be it an `m//` or `s///`), you can rely on the property of the null pattern, that reuses the last pattern seen. This leads to the second solution, which also eliminates the use of eval.

The above code fragment becomes:

```
my $pat = '^foo$';
"something" =~ /$pat/; # dummy match (MUST NOT FAIL!)
foreach( @list ) {
  print if //;
}
```

The only gotcha is that the dummy match that boots the regular expression engine must absolutely, positively succeed, otherwise the pattern will not be cached, and the `//` will match everything. If you can't count on fixed text to ensure the match succeeds, you have two possibilities.

If you can guarantee that the pattern variable contains no meta-characters (things like *, +, ^, $...), you can use the dummy match:

```
"$pat" =~ /\Q$pat\E/; # guaranteed if no meta-characters present
```

If there is a possibility that the pattern can contain meta-characters, you should search for the pattern or the non-searchable \377 character as follows:

```
"\377" =~ /$pat|^\377$/; # guaranteed if meta-characters present
```

Another approach:

It depends on the complexity of the regexp to which you apply this technique. One common usage where a compiled regexp is usually more efficient is to ''*match any one of a group of patterns*'' over and over again.

Maybe with a helper routine, it's easier to remember. Here is one slightly modified from Jeffery Friedl's example in his book ''*Mastering Regex*''.

```
######################################################
# Build_MatchMany_Function
# -- Input:  list of patterns
# -- Output: A code ref which matches its $_[0]
#            against ANY of the patterns given in the
#            "Input", efficiently.
#
sub Build_MatchMany_Function {
  my @R = @_;
  my $expr = join '||', map { "\$_[0] =~ m/\$R[$_]/o" } ( 0..$#R );
  my $matchsub = eval "sub { $expr }";
  die "Failed in building regex @R: $@" if $@;
  $matchsub;
}
```

Example usage:

```
@some_browsers = qw(Mozilla Lynx MSIE AmigaVoyager lwp libwww);
$Known_Browser=Build_MatchMany_Function(@some_browsers);
```

```
while (<ACCESS_LOG>) {
  # ...
  $browser = get_browser_field($_);
  if ( ! &$Known_Browser($browser) ) {
    print STDERR "Unknown Browser: $browser\n";
  }
  # ...
}
```

2.9 perldoc's Rarely Known But Very Useful Options

First of all, I want to stress that you cannot become a Perl hacker without knowing how to read Perl documentation and search through it. Books are good, but an easily accessible and searchable Perl reference is at your fingertips and is a great time saver.

While you can use online Perl documentation at the Web, the `perldoc` utility provides you with access to the documentation installed on your system. To find out what Perl manpages are available execute:

```
% perldoc perl
```

To find what functions perl has, execute:

```
% perldoc perlfunc
```

To learn the syntax and to find examples of a specific function, you would execute (e.g. for `open()`):

```
% perldoc -f open
```

Note: In perl5.00503 and earlier, there is a bug in this and the `-q` options of `perldoc`. It won't call `pod2man`, but will display the section in POD format instead. Despite this bug it's still readable and very useful.

To search through the Perl FAQ (*perlfaq* manpage) sections you would (e.g for the `open` keyword) execute:

```
% perldoc -q open
```

This will show you all the matching Q&A sections, still in POD format.

To read the *perldoc* manpage you execute:

```
% perldoc perldoc
```

;o)

3 CGI to mod_perl Porting. mod_perl Coding guidelines.

3.1 What we will learn in this chapter

- Before you start to code

- Exposing Apache::Registry secrets

- Sometimes it Works, Sometimes it Doesn't

- `@INC` and mod_perl

- Reloading Modules and Required Files

- Name collisions with Modules and libs

- __END__ and __DATA__ tokens

- Output from system calls

- Using `format()` and `write()`

- Terminating requests and processes, the `exit()` and `child_terminate()` functions

- `die()` and mod_perl

- I/O is different

- STDIN, STDOUT and STDERR streams

- Global Variables Persistance

- Generating correct HTTP Headers

- NPH (Non Parsed Headers) scripts

- BEGIN blocks

- END blocks

- Command line Switches (-w, -T, etc)

- The strict pragma

- Passing ENV variables to CGI

- Apache and syslog

- Filehandlers and locks leakages

- The Script Is Too Dirty, But It Does The Job And I Cannot Afford To Rewrite It.

- Apache::PerlRun--a closer look

3.2 Before you start to code

It can be a good idea to tighten up some of your Perl programming practices, since mod_perl doesn't tolerate sloppy programming.

This chapter relies on a certain level of Perl knowledge. Please read through the *Perl Reference* chapter and make sure you know the material covered there. This will allow me to concentrate on pure mod_perl issues and make them more prominent to the experinced Perl programmer, which would otherwise be lost in the sea of Perl background notes.

Additional resources:

- **Perl Module Mechanics**

  This page describes the mechanics of creating, compiling, releasing, and maintaining Perl modules. http://world.std.com/~swmcd/steven/perl/module_mechanics.html

  The information is very relevant to a mod_perl developer.

- **The Eagle Book**

  ''Writing Apache Modules with Perl and C'' is a ''must have'' book!

  See the details at http://www.modperl.com .

- **"Programming Perl" Book**

- **"Perl Cookbook" Book**

3.3 Exposing Apache::Registry secrets

Let's start with some simple code and see what can go wrong with it, detect bugs and debug them, discuss possible pitfalls and how to avoid them.

I will use a simple CGI script, that initializes a `$counter` to 0, and prints its value to the screen while incrementing it.

```
counter.pl:
----------
#!/usr/bin/perl -w
use strict;

print "Content-type: text/plain\r\n\r\n";

my $counter = 0;
```

```
for (1..5) {
  increment_counter();
}

sub increment_counter{
  $counter++;
  print "Counter is equal to $counter !\r\n";
}
```

You would expect to see the output:

```
Counter is equal to 1 !
Counter is equal to 2 !
Counter is equal to 3 !
Counter is equal to 4 !
Counter is equal to 5 !
```

And that's what you see when you execute this script the first time. But let's reload it a few times... See, suddenly after a few reloads the counter doesn't start its count from 1 any more. We continue to reload and see that it keeps on growing, but not steadily starting almost randomly at 10, 10, 10, 15, 20... Weird...

```
Counter is equal to 6 !
Counter is equal to 7 !
Counter is equal to 8 !
Counter is equal to 9 !
Counter is equal to 10 !
```

We saw two anomalies in this very simple script: Unexpected increment of our counter over 5 and inconsistent growth over reloads. Let's investigate this script.

3.3.1 The First Mystery

First let's peek into the `error_log` file. Since we have enabled the warnings what we see is:

```
Variable "$counter" will not stay shared
at /home/httpd/perl/conference/counter.pl line 13.
```

The *Variable "$counter" will not stay shared* warning is generated when the script contains a named nested subroutine (a not anonymous subroutine defined inside another subroutine) that refers to a lexically scoped variable defined outside this nested subroutine.

Add 'use diagnostics;' to see the long version of the warning.

Do you see a nested named subroutine in my script? I don't! What's going on? Maybe it's a bug? But wait, maybe the perl interpreter sees the script in a different way, maybe the code goes through some changes before it actually gets executed? The easiest way to check what's actually happening is to run the script with a debugger.

But since we must debug it when it's being executed by the webserver, a normal debugger wouldn't help, because the debugger has to be invoked from within the webserver. Luckily Doug MacEachern wrote the `Apache::DB` module and we will use it to debug my script. While `Apache::DB` allows you to debug the code interactively, we will do it non-interactively.

Modify the `httpd.conf` file in the following way:

```
PerlSetEnv PERLDB_OPTS "NonStop=1 LineInfo=/tmp/db.out AutoTrace=1 frame=2"
PerlModule Apache::DB
<Location /perl>
  PerlFixupHandler Apache::DB
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  PerlSendHeader On
</Location>
```

Restart the server and issue a request to *counter.pl* as before. On the surface nothing has changed--we still see the correct output as before, but two things happened in the background:

First, the file */tmp/db.out* was written, with a complete trace of the code that was executed.

Second, *error_log* now contains the real code that was actually executed. This is produced as a side effect of reporting the *Variable "$counter" will not stay shared at...* warning that we saw earlier.

Here is the code that was actually executed:

```
package Apache::ROOT::perl::conference::counter_2epl;
use Apache qw(exit);
sub handler {
  BEGIN {
    $^W = 1;
  };
  $^W = 1;

  use strict;

  print "Content-type: text/plain\r\n\r\n";

  my $counter = 0;

  for (1..5) {
    increment_counter();
  }

  sub increment_counter{
    $counter++;
    print "Counter is equal to $counter !\r\n";
  }
}
```

The original code wasn't idented. I've idented it for you to stress that the code was wrapped inside the `handler()` subroutine.

What do we learn from this?

First, that every cgi script is cached under a package whose name is formed from the `Apache::ROOT::` prefix and the relative part of the script's URL (`perl::conference::counter_2epl`) by replacing all occurrences of `/` with `::`. That's how mod_perl knows what script should be fetched from the cache--each script is just a package with a single subroutine named `handler`.

Second, you see now why the `diagnostics` pragma talked about an inner (nested) subroutine--`increment_counter` is actually a nested subroutine.

With mod_perl, each subroutine in every `Apache::Registry` script is nested inside the `handler` subroutine.

It's important to understand that the *inner subroutine* effect happens only with code that `Apache::Registry` wraps with a declaration of the `handler` subroutine. If you put your code into a library or module, which the main script `require()`'s or `use()`'s, this effect doesn't occur.

For example if we put the subroutine `increment_counter()` into `mylib.pl`, save it in the same directory as the main script and `require()` it, there will be no problem at all. (Don't forget the `1;` at the end of the library or the `require()` might fail.)

```
mylib.pl:
---------
sub increment_counter{
  $counter++;
  print "Counter is equal to $counter !\r\n";
}
1;

counter.pl:
----------
#!/usr/bin/perl -w

use strict;
require "./mylib.pl";

print "Content-type: text/plain\r\n\r\n";

my $counter = 0;

for (1..5) {
  increment_counter();
}
```

Personally, unless the script is very short, I tend to write all the code in external libraries, and to have only a few lines in the main script. Generally the main script simply calls the main function of my library. Usually I call it `init()`. I don't worry about nested subroutine effects anymore (unless I create them myself :).

The section *Remedies working for Inner Subroutine* from the *Perl Reference* chapter discusses some of the possible workarounds for this problem.

You shouldn't be intimidated by this issue at all, since Perl is your friend. Just keep the warnings mode **On** and Perl will gladly tell you whenever you have this effect, by saying:

```
Variable "$counter" will not stay shared at ...[snipped]
```

Just don't forget to check your *error_log* file, before going into production!

By the way, the above example was pretty boring. In my first days of using mod_perl, I wrote a simple user registration program. I'll give a very simple representation of this program.

```
use CGI;
$q = new CGI;
my $name = $q->param('name');
print_respond();

sub print_respond{
  print "Content-type: text/plain\r\n\r\n";
  print "Thank you, $name!";
}
```

My boss and I checked the program at the development server and it worked OK. So we decided to put it in production. Everything was OK, but my boss decided to keep on checking by submitting variations of his profile. Imagine the surprise when after submitting his name (let's say ''The Boss'' :), he saw the response ''Thank you, Stas Bekman!''.

What happened is that I tried the production system as well. I was new to mod_perl stuff, and was so excited with the speed improvement that I didn't notice the nested subroutine problem. It hit me. At first I thought that maybe Apache had started to confuse connections, returning responses from other people's requests. I was wrong of course.

Why didn't we notice this when we were trying the software on our development server? Keep reading and you will understand why.

3.3.2 The Second Mystery

Let's return to our original example and proceed with the second mystery we noticed. Why did we see inconsistent results over numerous reloads?

That's very simple. Every time a server gets a request to process, it hands it over one of the children, generally in a round robin fashion. So if you have 10 httpd children alive, the first 10 reloads might seem to be correct because the effect we've just talked about starts to appear from the second re-invocation. Subsequent reloads then return unexpected results.

Moreover, requests can appear at random and children don't always run the same scripts. At any given moment one of the children could have served the same script more times than any other, and another may never have run it. That's why we saw the strange behavior.

Now you see why we didn't notice the problem with the user registration system in the example. First, we didn't look at the `error_log`. (As a matter of fact we did, but there were so many warnings in there that we couldn't tell what were the important ones and what were not). Second, we had too many server children running to notice the problem.

A workaround is to run the server as a single process. You achieve this by invoking the server with the `-X` parameter (`httpd -X`). Since there are no other servers (children) running, you will see the problem on the second reload.

But before that, let the `error_log` help you detect most of the possible errors--most of the warnings can become errors, so you should make sure to check every warning that is detected by perl, and probably you should write the code in such a way that no warnings appear in the `error_log`. If your `error_log` file is filled up with hundreds of lines on every script invocation, you will have difficulty noticing and locating real problems.

Of course none of the warnings will be reported if the warning mechanism is not turned **On**. Refer to the ''*Warnings Explained*'' section from the *Perl Reference* chapter to learn about warnings in general and to the ''*Warnings*'' section in this chapter to learn how to turn them on and off under mod_perl.

3.4 Sometimes it Works, Sometimes it Doesn't

When you start running your scripts under mod_perl, you might find yourself in a situation where a script seems to work, but sometimes it screws up. And the more it runs without a restart, the more it screws up. Often the problem is easily detectable and solvable. You have to test your script under a server running in single process mode (`httpd -X`).

Generally the problem you have is of using global variables. Because global variables don't change from one script invocation to another unless you change them, you can find your scripts do strange things.

Let's look at three real world examples:

3.4.1 An Easy Break-in

The first example is amazing--Web Services. Imagine that you enter some site where you have an account, perhaps a free email account. Now you want to see other users' mail.

You type in a username you want to peek at and a dummy password and try to enter the account. On some services this will work!!!

You say, why in the world does this happen? The answer is simple: **Global Variables**. You have entered the account of someone who happened to be served by the same server child as you. Because of sloppy programming, a global variable was not reset at the beginning of the program and voila, you can easily peek into others' email! Here is an example of sloppy code:

```
use vars ($authenticated);
my $q = new CGI;
my $username = $q->param('username');
my $passwd   = $q->param('passwd');
```

```
  authenticate($username,$passwd);
    # failed, break out
  unless ($authenticated){
    print "Wrong passwd";
    exit;
  }
    # user is OK, fetch user's data
  show_user($username);

  sub authenticate{
    my ($username,$passwd) = @_;
        # some checking
    $authenticated = 1 if SOME_USER_PASSWD_CHECK_IS_OK;
  }
```

Do you see the catch? With the code above, I can type in any valid username and any dummy passwd and enter that user's account, if someone has successfully entered his account before me using the same child process! Since `$authenticated` is global--if it becomes 1 once, it'll stay 1 for the remainder of the child's life!!! The solution is trivial--reset `$authenticated` to 0 at the beginning of the program.

A cleaner solution of course is not to rely on global variables, but rely on the return value from the function.

```
  my $q = new CGI;
  my $username = $q->param('username');
  my $passwd   = $q->param('passwd');
  my $authenticated = authenticate($username,$passwd);
    # failed, break out
  unless ($authenticated){
    print "Wrong passwd";
    exit;
  }
    # user is OK, fetch user's data
  show_user($username);

  sub authenticate{
    my ($username,$passwd) = @_;
        # some checking
    return (SOME_USER_PASSWD_CHECK_IS_OK) ? 1 : 0;
  }
```

Of course this example is trivial--but believe me it happens!

3.4.2 Thinking mod_cgi

Just another little one liner that can spoil your day, assuming you forgot to reset the `$allowed` variable. It works perfectly OK in plain mod_cgi:

```
  $allowed = 1 if $username eq 'admin';
```

But using mod_perl, and if your system administrator with superuser access rights has previously used the system, anybody who is lucky enough to be served later by the same child which served your administrator will happen to gain the same rights.

The obvious fix is:

```
$allowed = $username eq 'admin' ? 1 : 0;
```

3.4.3 Regular Expression Memory

Another good example is usage of the **/o** regular expression modifier, which compiles a regular expression once, on its first execution, and never compiles it again. This problem can be difficult to detect, as after restarting the server each request you make will be served by a different child process, and thus the regex pattern for that child will be compiled afresh. Only when you make a request that happens to be served by a child which has already cached the regex will you see the problem. Generally you miss that. When you press reload, you see that it works (with a new, fresh child). Eventually it doesn't, because you get a child that has already cached the regex and won't recompile because of the **/o** modifier.

An example of such a case would be:

```
my $pat = $q->param("keyword");
foreach( @list ) {
  print if /$pat/o;
}
```

To make sure you don't miss these bugs always test your CGI in single process mode (httpd -X).

To solve this particular **/o** modifier problem refer to the *Compiled Regular Expressions* section of the *Perl Reference* chapter.

3.5 @INC and mod_perl

The basic Perl `@INC` behaviour is explained in the section *use(), require(), do(), %INC and @INC Explained* of the *Perl Reference* chapter.

When running under mod_perl, once the server is up `@INC` is frozen and cannot be updated. The only opportunity to **temporarily** modify `@INC` is while the script or the module are loaded and compiled for the first time. After that its value is reset to the original one. The only way to change `@INC` permanently is to modify it at Apache startup.

Two ways to alter `@INC` at server startup:

- In the configuration file. For example add:

    ```
    PerlSetEnv PERL5LIB /home/httpd/perl
    ```

  or

    ```
    PerlSetEnv PERL5LIB /home/httpd/perl:/home/httpd/mymodules
    ```

- In the startup file directly alter the `@INC`. For example

```
    startup.pl
    ----------
    use lib qw(/home/httpd/perl /home/httpd/mymodules);
```

and load the startup file from the configuration file by:

```
    PerlRequire /path/to/startup.pl
```

3.6 Reloading Modules and Required Files

You might want to read the ''*use(), require(), do(), %INC and @INC Explained*'' of the *Perl Reference* chapter before you proceed.

When you develop plain CGI scripts, you can just change the code, and rerun the CGI from your browser. Since the script isn't cached in memory, the next time you call it the server starts up a new perl process, which recompiles it from scratch. The effects of any modifications you've applied are immediately present.

The situation is different with `Apache::Registry`, since the whole idea is to get maximum performance from the server. By default, the server won't spend time checking whether any included library modules have been changed. It assumes that they weren't, thus saving a few milliseconds to `stat()` the source file (multiplied by however many modules/libraries you `use()` and/or `require()` in your script.)

The only check that is done is to see whether your main script has been changed. So if you have only scripts which do not `use()` or `require()` other perl modules or packages, there is nothing to worry about. If, however, you are developing a script that includes other modules, the files you `use()` or `require()` aren't checked for modification and you need to do something about that.

So how do we get our modperl-enabled server to recognize changes in library modules? Well, there are a couple of techniques:

3.6.1 Restarting the server

The simplest approach is to restart the server each time you apply some change to your code.

After restarting the server about 100 times, you will tire of it and you will look for other solutions.

3.6.2 Using Apache::StatINC for the Development Process

Help comes from the `Apache::StatINC` module. When Perl pulls a file via `require()`, it stores the full pathname as a value in the global hash `%INC` with the file name as the key. `Apache::StatINC` looks through `%INC` and it immediately reloads any files it finds in there if it sees that they have been updated on disk.

To enable this module just add two lines to `httpd.conf`.

```
PerlModule Apache::StatINC
PerlInitHandler Apache::StatINC
```

To be sure it really works, turn on debug mode on your development box by adding `PerlSetVar StatINCDebug On` to your config file. You end up with something like this:

```
PerlModule Apache::StatINC
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  PerlSendHeader On
  PerlInitHandler Apache::StatINC
  PerlSetVar StatINCDebug On
</Location>
```

Be aware that only the modules located in `@INC` are reloaded on change, and you can change `@INC` only before the server has been started (in the startup file).

Nothing you do in your scripts and modules which are pulled in with `require()` after server startup will have any effect on `@INC`.

When you write:

```
use lib qw(foo/bar);
```

`@INC` is changed only for the time the code is being parsed and compiled. When that's done, `@INC` is reset to its original value.

To make sure that you have set `@INC` correctly, configure */perl-status* location, fetch http://www.nowhere.com/perl-status?inc and look at the bottom of the page, where the contents of `@INC` will be shown.

Notice the following trap:

While ''.'' is in `@INC`, perl knows to `require()` files with pathnames given relative to the current (script) directory. After the script has been parsed, the server doesn't remember the path!

So you can end up with a broken entry in `%INC` like this:

```
$INC{bar.pl} eq "bar.pl"
```

If you want Apache::StatINC to reload your script--modify `@INC` at server startup, or use a full path in the `require()` call.

3.6.3 Reloading handlers

If you want to reload a perlhandler on each invocation, the following trick will do it:

```
PerlHandler "sub { do 'MyTest.pm'; MyTest::handler(shift) }"
```

`do()` reloads `MyTest.pm` on every request.

3.7 Name collisions with Modules and libs

This sections requires an indepth understanding of *use(), require(), do(), %INC and @INC*. Please refer to the *Perl Reference* chapter to learn more about it.

To make things clear before we go into details: each child process has its own `%INC` hash which is used to store information about its compiled modules. The keys of the hash are the names of the modules and files passed as arguments to `require()` and `use()`. The values are the full or relative paths to these modules and files.

Suppose we have `my-lib.pl` and `MyModule.pm` both located at `/home/httpd/perl/my/`.

- `/home/httpd/perl/my/` is in `@INC` at server startup.

  ```
  require "my-lib.pl";
  use MyModule.pm;
  print $INC{"my-lib.pl"},"\n";
  print $INC{"MyModule.pm"},"\n";
  ```

  prints:

  ```
  /home/httpd/perl/my/my-lib.pl
  /home/httpd/perl/my/MyModule.pm
  ```

  Adding `use lib`:

  ```
  use lib qw(.);
  require "my-lib.pl";
  use MyModule.pm;
  print $INC{"my-lib.pl"},"\n";
  print $INC{"MyModule.pm"},"\n";
  ```

  prints:

  ```
  my-lib.pl
  MyModule.pm
  ```

- `/home/httpd/perl/my/` isn't in `@INC` at server startup.

  ```
  require "my-lib.pl";
  use MyModule.pm;
  print $INC{"my-lib.pl"},"\n";
  print $INC{"MyModule.pm"},"\n";
  ```

wouldn't work, since perl cannot find the modules.

Adding `use lib`:

```
use lib qw(.);
require "my-lib.pl";
use MyModule.pm;
print $INC{"my-lib.pl"},"\n";
print $INC{"MyModule.pm"},"\n";
```

prints:

```
my-lib.pl
MyModule.pm
```

Let's look at three scripts with faults related to name space. For the following discussion we will consider just one individual child process.

Scenario 1

First, You can't have two identical module names running under the same server! Only the first one found in a `use()` or `require()` statement will be compiled into the package, the request for the other module will be skipped, since the server will think that it's already compiled. This is a direct result of using <%INC>, which has keys equal to the names of the modules. Two identical names will refer to the same key in the hash.

So if you have two different `Foo` modules in two different directories and two scripts `script1.pl` and `script2.pl`, placed like this:

```
./perl/tool1/Foo.pm
./perl/tool1/tool1.pl
./perl/tool2/Foo.pm
./perl/tool2/tool2.pl
```

Where a sample code could be:

```
./perl/tool1/tool1.pl
--------------------
use Foo;
print "Content-type: text/plain\r\n\r\n";
print "I'm Script number One\n";
foo();
--------------------

./perl/tool1/Foo.pm
--------------------
sub foo{
  print "<B>I'm Tool Number One!</B>\n";
}
1;
--------------------
```

```
   ./perl/tool2/tool2.pl
   --------------------
   use Foo;
   print "Content-type: text/plain\r\n\r\n";
   print "I'm Script number Two\n";
   foo();
   --------------------

   ./perl/tool2/Foo.pm
   --------------------
   sub foo{
     print "<B>I'm Tool Number Two!</B>\n";
   }
   1;
   --------------------
```

Both scripts call `use Foo;`. Only the first one called will know about `Foo`. When you call the second script it will not know about `Foo` at all--it's like you've forgotten to write `use Foo;`. Run the server in the single server mode (httpd -X) to detect this kind of bug immediately.

You will see the following in the error_log file:

```
   Undefined subroutine
   &Apache::ROOT::perl::tool2::tool2_2epl::foo called at
   /home/httpd/perl/tool2/tool2.pl line 4.
```

Scenario 2

If the files do not declare a package, the above is true for files you `require()` as well:

Suppose the content of the scripts and `config.pl` files is exactly like in the example above, and you have a directory structure like this:

```
   ./perl/tool1/config.pl
   ./perl/tool1/tool1.pl
   ./perl/tool2/config.pl
   ./perl/tool2/tool2.pl
```

and both scripts contain

```
   use lib qw(.);
   require "config.pl";
```

The second scenario is not different from the first, there is almost no difference between `use()` and `require()` if you don't have to import some symbols into a calling script. Only the first script served will actually do the `require()`, for the same reason as the example above. `%INC` already includes the key *"config.pl"*!

Scenario 3

It is interesting that the following scenario will fail too!

```
./perl/tool/config.pl
./perl/tool/tool1.pl
./perl/tool/tool2.pl
```

where `tool1.pl` and `tool2.pl` both `require()` the **same** `config.pl`.

There are three solutions for this:

Solution 1

The first two faulty scenarios can be solved by placing your library modules in a subdirectory struc-
ture so that they have different path prefixes. The file system layout will be something like:

```
./perl/tool1/Tool1/Foo.pm
./perl/tool1/tool1.pl
./perl/tool2/Tool2/Foo.pm
./perl/tool2/tool2.pl
```

And modify the scripts:

```
use Tool1::Foo;
use Tool2::Foo;
```

For `require()` (scenario number 2) use the following:

```
./perl/tool1/tool1-lib/config.pl
./perl/tool1/tool1.pl
./perl/tool2/tool2-lib/config.pl
./perl/tool2/tool2.pl
```

And each script contains respectively:

```
use lib qw(.);
require "tool1-lib/config.pl";

use lib qw(.);
require "tool2-lib/config.pl";
```

This solution isn't good, since while it might work for you now, if you add another script that wants
to use the same module or `config.pl` file, it would fail as we saw in the third scenario.

Let's see some better solutions.

Solution 2

Another option is to use a full path to the script, so it will be used as a key in `%INC`;

```
require "/full/path/to/the/config.pl";
```

This solution solves the problem of the first two scenarios. I was surprised that it worked for the third scenario as well!

With this solution you loose some portability. If you move the tool around in the file system you will have to change the base directory or write some additional script that will automatically update the hardcoded path after it was moved. Of course you will have to remember to invoke it.

Solution 3

Make sure you read all of this solution.

Declare a package in the required files! It should be unique to the rest of the package names you use. `%INC` will then use the unique package name for the key. It's a good idea to use at least two-level package names for your private modules, e.g. `MyProject::Carp` and not `Carp`, since the latter will collide with an existing standard package. Even if as of the time of your coding it doesn't yet exist, a package might enter the next perl distribution as a standard module and your code will be broken. Foresee problems like this and save yourself future trouble.

What are the implications of package declaration?

Without package declarations, it is very convenient to `use()` or `require()` files because all the variables and subroutines are part of the `main::` package. Any of them can be used as if they are part of the main script. With package declarations things are more awkward. You have to use the `Package::function()` method to call a subroutine from `Package` and to access a global variable `$foo` inside the same package you have to write `$Package::foo`.

Lexically defined variables, those declared with `my()` inside `Package` will be inaccessible from outside the package.

You can leave your scripts unchanged if you import the names of the global variables and subroutines into the namespace of package **main::** like this:

```
use Module qw(:mysubs sub_b $var1 :myvars);
```

You can export both subroutines and global variables. Note however that this method has the disadvantage of consuming more memory for the current process.

See `perldoc Exporter` for information about exporting other variables and symbols.

This completely covers the third scenario. When you use different module names in package declarations, as explained above, you cover the first two as well.

See also the `perlmodlib` and `perlmod` manpages.

From the above discussion it should be clear that you cannot run development and production versions of the tools using the same apache server! You have to run a separate server for each. They can be on the same machine, but the servers will use different ports.

3.8 __END__ and __DATA__ tokens

`Apache::Registry` scripts cannot contain __END__ or __DATA__ tokens.

Why? Because `Apache::Registry` scripts are being wrapped into a subroutine called `handler`, like
the script at URI `/perl/test.pl`:

```
print "Content-type: text/plain\r\n\r\n";
print "Hi";
```

When the script is being executed under `Apache::Registry` handler, it actually becomes:

```
package Apache::ROOT::perl::test_2epl;
use Apache qw(exit);
sub handler {
  print "Content-type: text/plain\r\n\r\n";
  print "Hi";
}
```

So if you happen to put an __END__ tag, like:

```
print "Content-type: text/plain\r\n\r\n";
print "Hi";
__END__
Some text that wouldn't be normally executed
```

it will be turned into:

```
package Apache::ROOT::perl::test_2epl;
use Apache qw(exit);
sub handler {
  print "Content-type: text/plain\r\n\r\n";
  print "Hi";
  __END__
  Some text that wouldn't be normally executed
}
```

and you try to execute this script, you will receive the following warning:

```
Missing right bracket at .... line 4, at end of line
```

Perl cuts everything after the __END__ tag. The same applies to the __DATA__ tag.

Also, rememeber that whatever applies to `Apache::Registry` scripts, in most cases applies to
`Apache::PerlRun` scripts.

3.9 Output from system calls

The output of `system()`, `exec()`, and `open(PIPE,"|program")` calls will not be sent to the
browser unless your Perl was configured with `sfio`.

You can use backticks as a possible workaround:

```
print `command here`;
```

But you're throwing performance out the window either way. Best not to fork at all if you can avoid it.

3.10 Using format() and write()

The interface to filehandles which are linked to variables with Perl's `tie()` function is not yet complete. The `format()` and `write()` functions are missing. If you configure Perl with `sfio`, `write()` and `format()` should work just fine.

3.11 Terminating requests and processes, the exit() and child_terminate() functions

Perl's `exit()` built-in function cannot be used in mod_perl scripts. Calling it causes the mod_perl process to exit (which defeats the object of using mod_perl). The `Apache::exit()` function should be used instead.

You might start your scripts by overriding the `exit()` subroutine (if you use `Apache::exit()` directly, you will have a problem testing the script from the shell, unless you put `use Apache ();` into your code.) I use the following code:

```
BEGIN {
    # Auto-detect if we are running under mod_perl or CGI.
  $USE_MOD_PERL = ( (exists $ENV{'GATEWAY_INTERFACE'}
                and $ENV{'GATEWAY_INTERFACE'} =~ /CGI-Perl/)
                   or exists $ENV{'MOD_PERL'} ) ? 1 : 0;
}
use subs qw(exit);

# Select the correct exit function
########
sub exit{
  $USE_MOD_PERL ? Apache::exit(0) : CORE::exit(0);
}
```

Now the correct `exit()` will be always chosen, whether you run the script under mod_perl, ordinary CGI or from the shell.

Note that if you run the script under `Apache::Registry`, **The Apache function `exit()` overrides the Perl core built-in function**. While you see `exit()` listed in `@EXPORT_OK` of the Apache package, `Apache::Registry` does something you don't see and imports this function for you. This means that if your script is running under `Apache::Registry` handler you don't have to worry about `exit()`. The same applies to `Apache::PerlRun`.

If you use `CORE::exit()` in scripts running under modperl, the child will exit, but neither a proper exit nor logging will happen on the way. `CORE::exit()` cuts off the server's legs.

Note that `Apache::exit(-2)` or `Apache::exit(Apache::Constants::DONE)` will cause the server to exit gracefully, completing the logging functions and protocol requirements etc.

If you need to shut down the child cleanly after the request was completed, use the `$r->child_terminate` method. You can call it anywhere in the code, and not just at the ''end''. This sets the value of the `MaxRequestsPerChild` configuration variable to 1 and clears the `keepalive` flag. After the request is serviced, the current connection is broken, because of the `keepalive` flag, and the parent tells the child to cleanly quit, because `MaxRequestsPerChild` is smaller than the number of requests served.

You can accomplish this in two ways--in the `Apache::Registry` script:

```
Apache->request->child_terminate;
```

or in httpd.conf:

```
PerlFixupHandler "sub { shift->child_terminate }"
```

You would want to use the latter example only if you wanted the child to terminate every time the registered handler is called. Probably this is not what you want.

Here is an example of assigning of the postprocessing handler:

```
my $r = shift;
$r->post_connection(\&exit_child);
sub exit_child{
    # some logic here if needed
  $r->child_terminate;
}
```

The above is the code that is used by the `Apache::SizeLimit` module which terminates processes that grow bigger than a value you choose.

`Apache::GTopLimit` (based on libgtop) is a similar module. It does the same thing, plus you can configure it to terminate processes when their shared memory shrinks below some specified size.

As mentioned before, it is unnecessary to postpone the execution of `child_terminate()`. You can call it anywhere in the code, it won't terminate the child's execution until the request has been served. Don't confuse it with `exit()`.

3.12 die() and mod_perl

When you write:

```
  open FILE, "foo" or die "Cannot open foo file for reading: $!";
```

in a perl script and execute it--the script would `die()` if it will be unable to open the file, by aborting the script execution, printing the death reason and quitting the Perl interpreter.

You hardly will find a properly written Perl script that doesn't have at least one `die()` statement in it, if it has to cope with system calls and alike.

CGI script running under mod_cgi exits on its completion. The Perl interperter exits as well. So it doesn't really matter whether the interpreter quits because the script died by natural death (when the last statement was executed) or aborted by `die()` statement.

In mod_perl we don't want the interpreter to quit. We know already that when the script completes its chores the interpeter won't quit. There is no reason why it should quit when the script is stopped because of `die()`. As a result calling `die()` wouldn't quit the process.

And this is how it works--when the `die()` gets triggered, it's mod_perl's `$SIG{__DIE__}` handler that logs the error message and calls Apache::exit() instead of real `die()`. Thus the script stops, but the process doesn't quit.

This is an example of a trapping code, not the real code:

```
  $SIG{__DIE__} = sub { print STDERR @_; Apache::exit(); }
```

3.13 I/O is different

If you are using Perl 5.004 or better, most CGI scripts can run under mod_perl untouched. If you're using 5.003, Perl's built-in `read()` and `print()` functions do not work as they do under CGI. If you're using `CGI.pm`, use $query->print instead of plain ol' `print()`.

3.14 STDIN, STDOUT and STDERR streams

In mod_perl both `STDIN` and `STDOUT` are tied to the socket the request came from. `STDERR` is tied to the *error_log* file.

To print to STDOUT you can either use a regular `print()` (which is automagically tied to the the socket) or the `$r->print` method.

3.15 Global Variables Persistance

Since the child process generally doesn't exit before it has serviced several requests, global variables persist inside the same process from request to request. This means that you must never rely on the value of the global variable if it wasn't initialized at the beginning of the request processing.

You should avoid using global variables unless it's impossible without them, because it will make the code development harder and you will have to make very sure that all the variables are initialized before being used. Use `my()` scoped variables everywhere you can.

You should be especially careful with *Perl Special Variables* (See the *Perl Reference* chapter) which cannot be lexically scoped. You have to use `local()` instead.

3.16 Generating correct HTTP Headers

A HTTP response header consists of at least two fields. HTTP response and `Content-type`:

```
HTTP/1.0 200 OK
Content-Type: text/plain
```

after adding one more new line, you can start printing the content. A more complete response includes the date timestamp and server type, like in this response:

```
HTTP/1.0 200 OK
Date: Tue, 28 Dec 1999 18:47:58 GMT
Server: Apache/1.3.10-dev (Unix) mod_perl/1.21_01-dev
Content-Type: text/plain
```

To notify that the server was configured with KeepAlive Off, you need to tell the client that the connection was closed, with:

```
Connection: close
```

There can be other headers as well, like caching control and other specified by HTTP protocol. You can code the response header with a single `print()`:

```
  print qq{HTTP/1.1 200 OK
Date: Tue, 28 Dec 1999 18:49:41 GMT
Server: Apache/1.3.10-dev (Unix) mod_perl/1.21_01-dev
Connection: close
Content-type: text/plain

  };
```

Notice the double new line at the end. But you have to prepare a timestamp string (`Apache::Util::ht_time()` does just this) and to know what server you are running under. You needed to send only the response MIME type (`Content-type`) under mod_cgi, why should you do that manually under mod_perl.

Actually do want to set some headers manually, but not everytime. So mod_perl gives you the default set of headers, just like in the example above. And if you want to override or add more headers you can do that as well. Let's see how to do that.

When writing your own handlers and scripts with the Perl API the proper way to send the HTTP header is with `send_http_header()` method. If you need to add or override methods you can use the `headers_out()` method:

```
$r->headers_out("Server" => "Apache Next Generation 10.0");
$r->headers_out("Date" => "Tue, 28 Dec 1999 18:49:41 GMT");
```

When you have prepared all the headers you send them with:

```
$r->send_http_header;
```

Some headers have special aliases:

```
$r->content_type('text/plain');
```

is the same as:

```
 headers_out("Content-type" => "text/plain");
```

A typical handler looks like this:

```
$r->content_type('text/plain');
$r->send_http_header;
return OK if $r->header_only;
```

If the client issues a HTTP HEAD request rather than the usual GET, to be compliant with the HTTP proto-
col we should not send the document body, but the the HTTP header only. When Apache receives a
HEAD request, it sets *header_only()* to true. If we see that this has happened, we return from the handler
immediately with an OK status code.

Generally, you don't need the explicit content type setting, since Apache does it for you, by looking up the
MIME type of the request by matching the extension of the URI in the MIME tables (from the *mime.types*
file). So if the request URI is */welcome.html*, the text/html content-type will be picked. However for
CGI scripts or URIs that cannot be mapped by a known extension, you should set the appropriate type by
using content_type() method.

The situation is a little bit different with Apache::Registry and similar handlers. If you take a basic
CGI script like this:

```
print "Content-type: text/plain\r\n\r\n";
print "Hello world";
```

it wouldn't work, because the HTTP header will not be sent. By default, mod_perl does not send any
headers itself. You may wish to change this by adding

```
PerlSendHeader On
```

in the <Location> part of your configuration. Now, the response line and common headers will be sent
as they are by mod_cgi. Just as with mod_cgi, PerlSendHeader will not send the MIME type and a
terminating double newline. Your script must send that itself, e.g.:

```
print "Content-type: text/html\r\n\r\n";
```

According to HTTP specs, you should send ''\cM\cJ'', ''\015\012'' or ''\0x0D\0x0A'' string. The ''\r\n'' is the way to do that on UNIX and MS-DOS/Windows machines. However, on a Mac ''\r\n'' eq ''\012\015'', exactly the other way around.

Note, that in most UNIX CGI scripts, developers use a simpler ''\n\n'' and not ''\r\n\r\n''. There are occasions where sending ''\n'' without ''\r'' can cause problems, make it a habit to send ''\r\n'' every time.

The `PerlSendHeader On` directive tells mod_perl to intercept anything that looks like a header line (such as `Content-Type: text/plain`) and automatically turn it into a correctly formatted HTTP/1.0 header, the same way it happens with CGI scripts running under mod_cgi. This allows you to keep your CGI scripts unmodified.

You can use $ENV{PERL_SEND_HEADER} to find out whether `PerlSendHeader` is **On** or **Off**. You use it in your module like this:

```
 if($ENV{PERL_SEND_HEADER}) {
     print "Content-type: text/html\r\n\r\n";
 }
 else {
     my $r = Apache->request;
     $r->content_type('text/html');
     $r->send_http_header;
 }
```

If you use `CGI.pm`'s `header()` function to generate HTTP headers, you do not need to activate this directive because `CGI.pm` detects *mod_perl* and calls `send_http_header()` for you. However, it does not hurt to use this directive anyway.

There is no free lunch--you get the mod_cgi behavior at the expense of the small but finite overhead of parsing the text that is sent. Note that mod_perl makes the assumption that individual headers are not split across print statements.

The `Apache::print()` routine has to gather up the headers that your script outputs, in order to pass them to `$r->send_http_header`. This happens in `src/modules/perl/Apache.xs` (print) and `Apache/Apache.pm` (send_cgi_header). There is a shortcut in there, namely the assumption that each print statement contains one or more complete headers. If for example you generate a `Set-Cookie` header by multiple `print()` statements, like this:

```
   print "Content-type: text/html\n";
   print "Set-Cookie: iscookietext\; ";
   print "expires=Wednesday, 09-Nov-1999 00:00:00 GMT\; ";
   print "path=\/\; ";
   print "domain=\.mmyserver.com\; ";
   print "\r\n\r\n";
   print "hello";
```

your generated `Set-Cookie` header is split over a number of print statements and gets lost. The above example wouldn't work! Try this instead:

```
   print "Content-type: text/html\n";
   my $cookie = "Set-Cookie: iscookietext\; ";
   $cookie .= "expires=Wednesday, 09-Nov-1999 00:00:00 GMT\; ";
   $cookie .= "path=\/\; ";
   $cookie .= "domain=\.mmyserver.com\; ";
   print $cookie;
   print "\r\n\r\n";
   print "hello";
```

Sometimes when you call a script you see an ugly `"Content-Type: text/html"` displayed at the top of the page, and of course the HTML the rest of the HTML code won't be rendered correctly by the browser. As you have seen above, this generally happens when your code has already sent the header so you see it rendered into a browser's page. This might happen when you call the `CGI.pm $q->header` method or mod_perl's `$r->send_http_header`.

If you have a complicated application where the header might be generated from many different places, depending on the calling logic, you might want to write a special subroutine that sends a header, and keeps track of whether the header has been already sent. Of course you can use a global variable to flag that the header has already been sent:

```
  use strict;
  use vars qw{$header_printed};
  $header_printed = 0;

  print_header("text/plain");
  print "It worked!\n";
  print_header("text/plain");

  sub print_header {
    my $type = shift || "text/html";
    unless  ($header_printed) {
      $header_printed = 1;
      my $r = Apache->request;
      $r->content_type($type);
      $r->send_http_header;
    }
  }
```

`$header_printed` variable that flags whether the header was sent or not gets initialized to false (0) at the beginning of each code invocation. Note that the second invocation of `print_header()` within the same code, will do nothing, since `$header_printed` will become true after `print_header()` will be executed for the first time.

A little bit memory more friendly solution is to use a fully qualified variable instead:

```
  use strict;
  $main::header_printed = 0;

  print_header("text/plain");
  print "It worked!\n";
  print_header("text/plain");

  sub print_header {
    my $type = shift || "text/html";
```

```
   unless  ($main::header_printed) {
     $main::header_printed = 1;
     my $r = Apache->request;
     $r->content_type($type);
     $r->send_http_header;
   }
 }
```

We just removed the global variable predeclaration, allowing you to use $header_printed under "use strict" and replaced $header_printed with $main::header_printed;

Someone may become tempted to use a more elegant Perl solution--the closure effect which seems to be a natural to be used here. Unfortunately it will not work. If the process was starting fresh for each script or handler, like with plain mod_cgi scripts, it would work just fine:

```
use strict;

print_header("text/plain");
print "It worked!\n";
print_header("text/plain");

{
  my $header_printed = 0;
  sub print_header {
    my $type = shift || "text/html";
    unless  ($header_printed) {
      $header_printed = 1;
      my $r = Apache->request;
      $r->content_type($type);
      $r->send_http_header;
    }
  }
}
```

In this code $header_printed is declared as lexically scoped (with my()) outside the subroutine print_header() and modified inside of it. Curly braces define the block which limits the scope of the lexically variable.

This means that once print_header() sets it to 1, it will stay 1 as long as the code is running. So all consequent calls to this subroutine will just return without doing a thing. Which serves our goal, but unfortunately it will work only for the first time the script will be invoked within a process. When the script will be accessed for a second time and will be served by the same process--the header will not be printed anymore, since print_header() will remember that the value of $header_printed equals to 1, it wouldn't be reinitialized, since the soubroutine wouldn't be recompiled.

Let's make our smart method more elaborate with PerlSendHeader directive settings, so it always does the right thing. It's especially important if you write an application that you are going to distribute, hopefully as Open Source.

You can continue to improve this subroutine even further to handle additional headers, such as cookies.

3.17 NPH (Non Parsed Headers) scripts

To run a Non Parsed Header CGI script under mod_perl, simply add to your code:

```
local $| = 1;
```

And if you normally set `PerlSendHeader On`, add this to your server's configuration file:

```
<Files */nph-*>
  PerlSendHeader Off
</Files>
```

3.18 BEGIN blocks

Perl executes `BEGIN` blocks as soon as possible, at the time of compiling the code. The same is true under mod_perl. However, since mod_perl normally only compiles scripts and modules once, either in the parent server or once per-child, `BEGIN` blocks in that code will only be run once. As the `perlmod` manpage explains, once a `BEGIN` block has run, it is immediately undefined. In the mod_perl environment, this means that `BEGIN` blocks will not be run during the response to an incoming request unless that request happens to be the one that causes the compilation of the code.

`BEGIN` blocks in modules and files pulled in via `require()` or `use()` will be executed:

- Only once, if pulled in by the parent process.

- Once per-child process if not pulled in by the parent process.

- An additional time, once per child process if the module is pulled in off a disk again via `Apache::StatINC`.

- An additional time, in the parent process on each restart if `PerlFreshRestart` is `On`.

- Unpredictable if you fiddle with `%INC` yourself.

`BEGIN` blocks in `Apache::Registry` scripts will be executed, as above plus:

- Only once, if pulled in by the parent process via

  `Apache::RegistryLoader`.

- Once per-child process if not pulled in by the parent process.

- An additional time, once per child process, each time the script file changes on disk.

- An additional time, in the parent process on each restart if pulled in by the parent process via `Apache::RegistryLoader` and `PerlFreshRestart` is `On`.

3.19 END blocks

As the `perlmod` manpage explains, an `END` subroutine is executed as late as possible, that is, when the interpreter exits. In the mod_perl environment, the interpreter does not exit until the server shuts down. However, mod_perl does make a special case for `Apache::Registry` scripts.

Normally, `END` blocks are executed by Perl during its **perl_run()** function. This is called once each time the Perl program is executed, i.e. under mod_cgi, once per invocation of the CGI script. However, mod_perl only calls **perl_run()** once, during server startup. Any `END` blocks encountered during main server startup, i.e. those pulled in by the `PerlRequire` or by any `PerlModule`, are suspended.

Apache versions 1.3b3 and later run the `END` blocks at `child_exit()`.

Except during the cleanup phase, any `END` blocks encountered during compilation of `Apache::Registry` scripts, including subsequent invocations when the script is cached in memory, are called after the script has completed.

All other `END` blocks encountered during other `Perl*Handler` call-backs, e.g. `PerlChildInitHandler`, will be suspended while the process is running and called during `child_exit()` when the process is shutting down. Module authors might wish to use `$r->register_cleanup()` as an alternative to `END` blocks if this behavior is not desirable. `$r->register_cleanup()` is called at the CleanUp processing phase of each request and thus can be used to emulate plain perl's `END{}` block behavior.

3.20 Command line Switches (-w, -T, etc)

Normally when you run perl from the command line, you have the shell invoke it with `#!/bin/perl` (sometimes referred to as a shebang line). In scripts running under mod_cgi, you may use perl execution switch arguments as described in the `perlrun` manpage, such as `-w`, `-T` or `-d`. Since scripts running under mod_perl don't need the shebang line, all switches except `-w` are ignored by mod_perl. This feature was added for a backward compatibility with CGI scripts.

Most command line switches have a special variable equivalent. Consult the `perlvar` manpage for more details.

3.20.1 Warnings

There are three ways to enable warnings:

- **Globally to all Processes**

  Setting:

  ```
  PerlWarn On
  ```

in `httpd.conf` will turn warnings **On** in any script.

You can then fine tune your code, turning warnings **Off** and **On** by setting the `$^W` variable in your scripts.

- **Locally to a script**

  ```
  #!/usr/bin/perl -w
  ```

  will turn warnings **On** for the scope of the script. You can turn them **Off** and **On** in the script by setting the `$^W` variable as noted above.

- **Locally to a block**

  This code turns warnings mode **On** for the scope of the block.

  ```
  {
    local $^W = 1;
    # some code
  }
  ```

  This turns it **Off**:

  ```
  {
    local $^W = 0;
    # some code
  }
  ```

  Note, that if you forget the `local` operator this code will affect the child processing the current request, and all the subsequent requests processed by that child. Thus

  ```
  $^W = 0;
  ```

  will turn the warnings *Off*, no matter what.

  If you want to turn warnings *On* for the scope of the whole file, as in the previous item, you can do this by adding:

  ```
  local $^W = 1;
  ```

  at the beginning of the file. Since a file is effectively a block, file scope behaves like a block's curly braces { } and `local $^W` at the start of the file will be effective for the whole file.

While having warning mode turned **On** is a must for a development server, you should turn it globally **Off** in a production server, since if every served request generates only one warning, and your server serves millions of requests per day, your log file will eat up all of your disk space and your system will die.

3.20.2 Taint Mode

Perl's `-T` switch enables *Taint* mode. If you aren't forcing all your scripts to run under **Taint** mode you are looking for trouble from malicious users. (See the *perlsec* manpage for more information)

Since the `-T` switch doesn't have an equivalent perl variable, mod_perl provides the `PerlTaintCheck` directive to turn on taint checks. In `httpd.conf`, enable this mode with:

```
PerlTaintCheck On
```

Now any code compiled inside httpd will be taint checked.

If you use the `-T` switch, Perl will warn you that you should use the `PerlTaintCheck` configuration directive and will otherwise ignore it.

3.20.3 Other switches

Finally, if you still need to to set additional perl startup flags such as `-d` and `-D`, you can use an environment variable `PERL5OPT`.

3.21 The strict pragma

It's _absolutely_ mandatory (at least for development) to start all your scripts with:

```
use strict;
```

If needed, you can always turn off the 'strict' pragma or a part of it inside the block, e.g:

```
{
  no strict 'refs';
  ... some code
}
```

It's more important to have `strict` pragma enabled under mod_perl than anywhere else. While it's not required by the language, its use cannot be too strongly recommended. It will save you a great deal of time. And, of course, clean scripts will still run under mod_cgi (plain CGI)!

3.22 Passing ENV variables to CGI

To pass an environment variable from a configuration file, add to it:

```
PerlSetEnv key val
PerlPassEnv key
```

e.g.:

```
  PerlSetEnv PERLDB_OPTS "NonStop=1 LineInfo=/tmp/db.out AutoTrace=1"
```

will set $ENV{PERLDB_OPTS}, and it will be accessible in every child.

%ENV is only set up for CGI emulation. If you are using the API, you should use $r->subpro-cess_env, $r->notes or $r->pnotes for passing data around between handlers. %ENV is slow because it must update the underlying C environment table. This also exposes the data on systems which allow users to see the environment with ps.

In any case, %ENV and the tables used by those methods are all cleared after the request is served so that $ENV{SESSION_ID} will not be swapped or reused by different http requests.

3.23 Apache and syslog

When native syslog support is enabled, the stderr stream will be redirected to /dev/null!

It has nothing to do with mod_perl (plain Apache does the same). Doug wrote the Apache::LogSTDERR module to work around this.

3.24 Filehandlers and locks leakages

When you write a script running under mod_cgi, you can get away with sloppy programming, like opening a file and letting the interpreter close it for you when the script had finished its run:

```
  open IN, "in.txt" or die "Cannot open in.txt for reading : $!\n";
```

For mod_perl, before the end of the script you **must** close() any files you opened!

```
  close IN;
```

If you forget to close(), you might get file descriptor leakage and (if you flock()ed on this file descriptor) unlock problems.

Even if you do close the files, but for some reason the interpreter was stopped before the close() call, the leakage is still there. For example when a user presses the Stop button. After a long run without restarting Apache your machine might run out of file descriptors, and worse, files might be left locked and unusable.

What can you do? Use IO::File (and the other IO::* modules). This allows you to assign the file handler to variable which can be my() (lexically) scoped. When this variable goes out of scope the file or other file system entity will be properly closed (and unlocked if it was locked). Lexically scoped variables will always go out of scope at the end of the script's invocation even if it was aborted in the middle. If the variable was defined inside some internal block, it will go out of scope at the end of the block. For example:

```
  {
    my $fh = new IO::File("filename") or die $!;
    # read from $fh
  } # ...$fh is closed automatically at end of block, without leaks.
```

As I have just mentioned, you don't have to create a special block for this purpose. A script in a file is effectively written in a block with the same scope as the file, so you can simply write:

```
  my $fh = new IO::File("filename") or die $!;
    # read from $fh
    # ...$fh is closed automatically at end of script, without leaks.
```

Using a { BLOCK }) makes sure is that the file is closed the moment that the end of the block is reached.

An even faster and lighter technique is to use `Symbol.pm`:

```
  my $fh = Symbol::gensym();
  open $fh, "filename" or die $!;
```

Use these approaches to ensure you have no leakages, but don't be too lazy to write `close()` statements. Make it a habit.

3.25 The Script Is Too Dirty, But It Does The Job And I Cannot Afford To Rewrite It.

You still can win from using mod_perl.

One approach is to replace the `Apache::Registry` handler with `Apache::PerlRun` and define a new location. The script can reside in the same directory on the disk.

```
  # srm.conf
  Alias /cgi-perl/ /home/httpd/cgi/

  # httpd.conf
  <Location /cgi-perl>
    #AllowOverride None
    SetHandler perl-script
    PerlHandler Apache::PerlRun
    Options ExecCGI
    allow from all
    PerlSendHeader On
  </Location>
```

Another ''bad'', but workable method is to set `MaxRequestsPerChild` to 1, which will force each child to exit after serving only one request. You will get the preloaded modules, etc., but the script will be compiled for each request, then thrown away. This isn't good for ''high-traffic'' sites, as the parent server will need to fork a new child each time one is killed. You can fiddle with `MaxStartServers` and `MinSpareServers`, so that the parent pre-spawns more servers than actually required and the killed one will immediately be replaced with a fresh one. Probably that's not what you want.

3.26 Apache::PerlRun--a closer look

`Apache::PerlRun` gives you the benefit of preloaded Perl and its modules. This module's handler emulates the CGI environment, allowing programmers to write scripts that run under CGI or mod_perl without any change. Unlike `Apache::Registry`, the `Apache::PerlRun` handler does not cache the script inside a subroutine. Scripts will be ''compiled'' on each request. After the script has run, its name space is flushed of all variables and subroutines. Still, you don't have the overhead of loading the Perl interpreter and the compilation time of the standard modules. If your script is very light, but uses lots of standard modules, you will see no difference between `Apache::PerlRun` and `Apache::Registry`!.

Be aware though, that if you use packages that use internal variables that have circular references, they will be not flushed!!! `Apache::PerlRun` only flushes your script's name space, which does not include any other required packages' name spaces. If there's a reference to a `my()` scoped variable that's keeping it from being destroyed after leaving the eval scope (of `Apache::PerlRun`), that cleanup might not be taken care of until the server is shutdown and `perl_destruct()` is run, which always happens after running command line scripts. Consider this example:

```
package Foo;
sub new { bless {} }
sub DESTROY {
  warn "Foo->DESTROY\n";
}

eval <<'EOF';
package my_script;
my $self = Foo->new;
#$self->{circle} = $self;
EOF

print $@ if $@;
print "Done with script\n";
```

First you'll see:

```
Foo->DESTROY
Done with script
```

Then, uncomment the line where `$self` makes a circular reference, and you'll see:

```
Done with script
Foo->DESTROY
```

In this case, under mod_perl you wouldn't see `Foo->DESTROY` until the server shutdown, or until your module properly took care of things.

;o)

4 Controlling and Monitoring the Server

4.1 What we will learn in this chapter

- Restarting techniques

- Implications of sending TERM, HUP, and USR1 to the server

- Using apachectl to control the server

- Safe Code Updates on a Live Production Server

- SUID start-up scripts

- Preparing for Machine Reboot

- Monitoring the Server. A watchdog.

- Running server in a single mode

4.2 Restarting techniques

All of these techniques require that you know the server PID (Process ID). The easiest way to find the PID is to look it up in the `httpd.pid` file. With my configuration it exists as `/usr/local/var/httpd_perl/run/httpd.pid`. It's easy to discover where to look at, by checking out the httpd.conf file. Open the file and locate the entry `PidFile`:

```
PidFile /usr/local/var/httpd_perl/run/httpd.pid
```

Another way is to use the `ps` and `grep` utilities:

```
% ps auxc | grep httpd_perl
```

or maybe:

```
% ps -ef | grep httpd_perl
```

This will produce a list of all `httpd_perl` (the parent and the children) processes. You are looking for the parent process. If you run your server as root - you will easily locate it, since it belongs to root. If you run the server as user, most likely all the processes will belong to that user (unless defined differently in the `httpd.conf`), but it's still easy to know 'who is the parent' -- the one of the smallest size...

You will notice many `httpd_perl` executables running on your system, but you should not send signals to any of them except the parent, whose pid is in the `PidFile`. That is to say you shouldn't ever need to send signals to any process except the parent. There are three signals that you can send the parent: **TERM**, **HUP**, and **USR1**.

4.3 Implications of sending TERM, HUP, and USR1 to the server

We will concentrate here on the implications of sending these signals to a mod_perl enabled server. For documentation on the implications of sending these signals to a plain Apache server see http://www.apache.org/docs/stopping.html .

TERM Signal: stop now

> Sending the **TERM** signal to the parent causes it to immediately attempt to kill off all of its children. This process may take several seconds to complete, following which the parent itself exits. Any requests in progress are terminated, and no further requests are served.

> That's the moment that the accumulated `END` blocks will be executed! Note that if you use `Apache::Registry` or `Apache::PerlRun`, then `END` blocks are being executed upon each request (at the end).

HUP Signal: restart now

> Sending the **HUP** signal to the parent causes it to kill off its children like in **TERM** (Any requests in progress are terminated) but the parent doesn't exit. It re-reads its configuration files, and re-opens any log files. Then it spawns a new set of children and continues serving hits.

> The server will reread its configuration files, flush all the compiled and preloaded modules, and rerun any startup files. It's equivalent to stopping, then restarting a server.

> Note: If your configuration file has errors in it when you issue a restart then your parent will not restart but exit with an error. See below for a method of avoiding this.

USR1 Signal: graceful restart

> The **USR1** signal causes the parent process to advise the children to exit after their current request (or to exit immediately if they're not serving anything). The parent re-reads its configuration files and re-opens its log files. As each child dies off the parent replaces it with a child from the new generation of the configuration, which begins serving new requests immediately.

> The only difference between **USR1** and **HUP** is that **USR1** allows children to complete any in-progress request prior to killing them off.

> By default, if a server is restarted (ala `kill -USR1 `cat logs/httpd.pid`` or with **HUP** signal), Perl scripts and modules are not reloaded. To reload **PerlRequire**'s, **PerlModule**'s, other `use()`'d modules and flush the `Apache::Registry` cache, enable with this command:

> ```
>  PerlFreshRestart On              (in httpd.conf)
> ```

It's worth mentioning that restart or termination can sometimes take quite a lot of time. Check out the `PERL_DESTRUCT_LEVEL=-1` option during the mod_perl `perl Makefile.PL` stage, which speeds this up and leads to more robust operation in the face of problems, like running out of memory. It is only usable if no significant cleanup has to be done by perl `END` blocks and `DESTROY` methods when the child terminates, of course. What constitutes significant cleanup? Any change of state outside of the current process that would not be handled by the operating system itself. So committing database transactions is significant but closing an ordinary file isn't.

Some folks prefer to specify signals using numerical values, rather than symbolics. If you are looking for these, check out your `kill(3)` man page. My page points to `/usr/include/sys/signal.h`, the relevant entries are:

```
#define SIGHUP    1    /* hangup, generated when terminal disconnects */
#define SIGTERM   15   /* software termination signal */
#define SIGUSR1   30   /* user defined signal 1 */
```

4.4 Using apachectl to control the server

Apache's distribution provides a nice script to control the server. It's called **apachectl** and it's installed into the same location with httpd. In our scenario - it's `/usr/local/sbin/httpd_perl/apachectl`.

Start httpd:

```
% /usr/local/sbin/httpd_perl/apachectl start
```

Stop httpd:

```
% /usr/local/sbin/httpd_perl/apachectl stop
```

Restart httpd if running by sending a **SIGHUP** or start if not running:

```
% /usr/local/sbin/httpd_perl/apachectl restart
```

Do a graceful restart by sending a **SIGUSR1** or start if not running:

```
% /usr/local/sbin/httpd_perl/apachectl graceful
```

Do a configuration syntax test:

```
% /usr/local/sbin/httpd_perl/apachectl configtest
```

Replace `httpd_perl` with `httpd_docs` in the above calls to control the **httpd_docs** server.

There are other options for **apachectl**, use `help` option to see them all.

It's important to understand that this script is based on the PID file which is `PIDFILE=/usr/local/var/httpd_perl/run/httpd.pid`. If you delete the file by hand - **apachectl** will fail to run.

Also, notice that **apachectl** is suitable to use from within your Unix system's startup files so that your web server is automatically restarted upon system reboot. Either copy the **apachectl** file to the appropriate location (`/etc/rc.d/rc3.d/S99apache` works on my RedHat Linux system) or create a symlink with that name pointing to the the canonical location. (If you do this, make certain that the script is writable only by root -- the startup scripts have root privileges during init processing, and you don't want to be opening any security holes.)

4.5 Safe Code Updates on a Live Production Server

You have prepared a new version of code, uploaded it into a production server, restarted it and it doesn't work. What could be worse than that? You also cannot go back, because you have overwritten the good working code.

It's quite easy to prevent it! Just don't overwrite the previous good files!!!

Personally I do all updates on the live server with a following sequence. Assume that the root directory lies in `/home/httpd/perl/rel`. When I'm about to update the files I create a new directory `/home/httpd/perl/beta`, copy the old files from `/home/httpd/perl/rel` and update it with new files I'm about to replace. The I do last sanity checks (file permissions (read+executable), run `perl -c` on the new modules to make sure there no errors in them). When I think I'm ready I do:

```
% cd /home/httpd/perl
% mv rel old && mv beta rel && stop && sleep 3 && restart && err
```

Let's explain what I'm doing. First I use alises to make things faster:

```
% alias | grep apachectl
graceful        /usr/local/apache/bin/apachectl graceful
rehup   /usr/local/apache/sbin/apachectl restart
restart /usr/local/apache/bin/apachectl restart
start   /usr/local/apache/bin/apachectl start
stop    /usr/local/apache/bin/apachectl stop

% alias err
tail -f /usr/local/apache/logs/error_log
```

So I write all the commands in one line, separated with semicolon and only then press `Enter` key. That ensures that if I suddenly get a connection lost (sadly but that happens sometimes) I wouldn't leave the server down if only the `stop` command squeezed in.

I backup the old working directory in `old`, and move the new one instead. I stop the server, give it a few seconds to shutdown (it might take even longer) and then do `restart` followed by immediate view of the tail of the `error_log` file in order to see that everything is OK. `apachectl` generates the status messages too early (e.g. on `stop` it says server has been stopped, while it's not yet, so don't rely on it, rely on `error_log` file instead). Also you have noticed that I use `restart` and not just `start`. I do this for the same reason of Apache's long stopping times (it depends on what you do with it of course!), so if you use `start` and Apache didn't release the port it listens to, the start would fail and `error_log` would tell that port is in use, e.g.:

```
   Address already in use: make_sock: could not bind to port 8080
```

But if you use `restart`, it will patiently wait for the server to quit and then will cleanly start it.

Now what happens if the new modules are broken? First of all, I see immediately the indication of the problems reported at `error_log` file, which I `tail -f` immediately after a restart command. That's easy, we just put everything as it was before:

```
  % mv rel bad && mv old rel && stop && sleep 3 && restart && err
```

And 99.9% that everything would be alright, and you have had only about 10 secs of downtime, which is pretty good!

4.6 SUID start-up scripts

For those who wants to use **SUID** startup script, here is an example for you. This script is **SUID** to **root**, and should be executable only by members of some special group at your site. Note the 10th line, which ''fixes an obscure error when starting apache/mod_perl'' by setting the real to the effective UID. As others have pointed out, it is the mismatch between the real and the effective UIDs that causes Perl to croak on the -e switch.

Note that you must be using a version of Perl that recognizes and emulates the suid bits in order for this to work. The script will do different things depending on whether it is named `start_http`, `stop_http` or `restart_http`. You can use symbolic links for this purpose.

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

# These constants will need to be adjusted.
$PID_FILE = '/home/www/logs/httpd.pid';
$HTTPD = '/home/www/httpd -d /home/www';

# These prevent taint warnings while running suid
$ENV{PATH}='/bin:/usr/bin';
$ENV{IFS}='';

# This sets the real to the effective ID, and prevents
# an obscure error when starting apache/mod_perl
$< = $>;
$( = $) = 0; # set the group to root too

# Do different things depending on our name
($name) = $0 =~ m|([^/]+)$|;

if ($name eq 'start_http') {
    system $HTTPD and die "Unable to start HTTP";
    print "HTTP started.\n";
    exit 0;
}

# extract the process id and confirm that it is numeric
$pid = `cat $PID_FILE`;
$pid =~ /(\d+)/ or die "PID $pid not numeric";
```

```
$pid = $1;

if ($name eq 'stop_http') {
    kill 'TERM',$pid or die "Unable to signal HTTP";
    print "HTTP stopped.\n";
    exit 0;
}

if ($name eq 'restart_http') {
    kill 'HUP',$pid or die "Unable to signal HTTP";
    print "HTTP restarted.\n";
    exit 0;
}

die "Script must be named start_http, stop_http, or restart_http.\n";
```

4.7 Preparing for Machine Reboot

When you run your own development box, it's OK to start the webserver by hand when you need it. On the production system, there is chance that the machine the server is running on will have to be rebooted. Once the reboot is completed, who is going to rememeber to start the server? It's an easy to forget task, and what happens if you aren't around when the machine was rebooted?

After the server installation is complete, it's important not to forget that you need to put a script, to perform the server startup and shutdown, into a standard system location, like /etc/rc.d/init.d or equivalent (varies from OS to OS). This is the directory where all other daemons are being started and shutted down from.

Generally the simplest solution is to copy there the apachectl script, that you will find in the same directory with httpd executable after Apache installation. If you have more than one Apache server, you have to put a script for each one, of course renaming them on the way.

For example on Linux RedHat machine with two server setup, I've the following setup:

```
/etc/rc.d/init.d/httpd_docs
/etc/rc.d/init.d/httpd_perl
/etc/rc.d/rc3.d/S86httpd_docs -> ../init.d/httpd_docs
/etc/rc.d/rc3.d/S87httpd_perl -> ../init.d/httpd_perl
/etc/rc.d/rc6.d/K86httpd_docs -> ../init.d/httpd_docs
/etc/rc.d/rc6.d/K87httpd_perl -> ../init.d/httpd_perl
```

In <init.d> directory reside the scripts themselves. In the rest of directories reside the symbolic links to these scripts, prepended with numbers to preserve a particular order of execution.

When a machine is booted and its runlevel set as 3 (multiuser+network), Linux goes into /etc/rc.d/rc3.d/ and executes the scripts the symbolic links point to with the start argument, so when it sees the *S87httpd_perl*, it executes:

```
/etc/rc.d/init.d/httpd_perl start
```

When the machine is being shutted down, the scripts pointed from *etc/rc.d/rc6.d/* directory are being executed, this time the scripts are called with `stop` argument, like:

```
/etc/rc.d/init.d/httpd_perl stop
```

Most of the systems are coming with GUI utilites to automate the symbolic links creation. For example Linux RH includes a `control-panel` utility, which among other utilities includes a `RunLevel Manager` that will help you to properly create the symbolic links. Of course before you use it, you should put the `apachectl` or similar scripts into a *init.d* or equivalent directory.

4.8 Monitoring the Server. A watchdog.

With mod_perl many things can happen to your server. The worst one is the possibility that the server will die when you will be not around. As with any other critical service you need to run some kind of watchdog.

One simple solution is to use a slightly modified **apachectl** script which I called apache.watchdog and to put it into the crontab to be called every 30 minutes or even every minute - if it's so critical to make sure the server will be up all the time.

The crontab entry:

```
0,30 * * * * /path/to/the/apache.watchdog >/dev/null 2>&1
```

The script:

```sh
#!/bin/sh

# this script is a watchdog to see whether the server is online
# It tries to restart the server if it's
# down and sends an email alert to admin

# admin's email
EMAIL=webmaster@somewhere.far
#EMAIL=root@localhost

# the path to your PID file
PIDFILE=/usr/local/var/httpd_perl/run/httpd.pid

# the path to your httpd binary, including options if necessary
HTTPD=/usr/local/sbin/httpd_perl/httpd_perl

# check for pidfile
if [ -f $PIDFILE ] ; then
  PID=`cat $PIDFILE`

  if kill -0 $PID; then
    STATUS="httpd (pid $PID) running"
    RUNNING=1
  else
    STATUS="httpd (pid $PID?) not running"
    RUNNING=0
```

```
   fi
else
  STATUS="httpd (no pid file) not running"
  RUNNING=0
fi

if [ $RUNNING -eq 0 ]; then
  echo "$0 $ARG: httpd not running, trying to start"
  if $HTTPD ; then
    echo "$0 $ARG: httpd started"
    mail $EMAIL -s "$0 $ARG: httpd started" </dev/null >& /dev/null
  else
    echo "$0 $ARG: httpd could not be started"
    mail $EMAIL -s "$0 $ARG: httpd could not be started" </dev/null >& /dev/null
  fi
fi
```

Another approach, probably even more practical, is to use the cool `LWP` perl package , to test the server by trying to fetch some document (script) served by the server. Why is it more practical? Because, while server can be up as a process, it can be stuck and not working, So failing to get the document will trigger restart, and ''probably'' the problem will go away. (Just replace `start` with `restart` in the `$restart_command` below.

Again we put this script into a crontab to call it every 30 minutes. Personally I call it every minute, to fetch some very light script. Why so often? If your server starts to spin and trash your disk's space with multiply error messages, in a 5 minutes you might run out of free space, which might bring your system to its knees. And most chances that no other child will be able to serve requests, since the system will be too busy, writing to an `error_log` file. Think big -- if you are running a heavy service, which is very fast, since you are running under mod_perl, adding one more request every minute, will be not felt by the server at all.

So we end up with crontab entry:

```
  * * * * * /path/to/the/watchdog.pl >/dev/null 2>&1
```

And the watchdog itself:

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

  use strict;
  use diagnostics;
  use URI::URL;
  use LWP::MediaTypes qw(media_suffix);

  my $VERSION = '0.01';
  use vars qw($ua $proxy);
  $proxy = '';

  require LWP::UserAgent;
  use HTTP::Status;

  ###### Config ########
  my $test_script_url = 'http://www.stas.com:81/perl/test.pl';
  my $monitor_email   = 'root@localhost';
```

```perl
  my $restart_command = '/usr/local/sbin/httpd_perl/apachectl restart';
  my $mail_program    = '/usr/lib/sendmail -t -n';
  ######################

  $ua  = new LWP::UserAgent;
  $ua->agent("$0/Stas " . $ua->agent);
  # Uncomment the proxy if you don't use it!
  #   $proxy="http://www-proxy.com";;
  $ua->proxy('http', $proxy) if $proxy;

  # If returns '1' it's we are alive
  exit 1 if checkurl($test_script_url);

  # We have got the problem - the server seems to be down. Try to
  # restart it.
  my $status = system $restart_command;
  #   print "Status $status\n";

  my $message = ($status == 0)
              ? "Server was down and successfully restarted!"
              : "Server is down. Can't restart.";

  my $subject = ($status == 0)
              ? "Attention! Webserver restarted"
              : "Attention! Webserver is down. can't restart";

  # email the monitoring person
  my $to = $monitor_email;
  my $from = $monitor_email;
  send_mail($from,$to,$subject,$message);

  # input:  URL to check
  # output: 1 if success, o for fail
  ######################
  sub checkurl{
    my ($url) = @_;

    # Fetch document
    my $res = $ua->request(HTTP::Request->new(GET => $url));

    # Check the result status
    return 1 if is_success($res->code);

    # failed
    return 0;
  } #  end of sub checkurl

  # sends email about the problem
  ######################
  sub send_mail{
    my($from,$to,$subject,$messagebody) = @_;

    open MAIL, "|$mail_program"
        or die "Can't open a pipe to a $mail_program :$!\n";

    print MAIL <<__END_OF_MAIL__;
  To: $to
```

```
  From: $from
  Subject: $subject

  $messagebody

  __END_OF_MAIL__

    close MAIL;
}
```

4.9 Running server in a single mode

Often while developing new code, you will want to run the server in single process mode. Running in single process mode inhibits the server from ''daemonizing'', allowing you to run it more easily under debugger control.

```
  % /usr/local/sbin/httpd_perl/httpd_perl -X
```

When you execute the above the server will run in the fg (foreground) of the shell you have called it from. So to kill you just kill it with **Ctrl-C**.

Note that in -X mode the server will run very slowly while fetching images. If you use Netscape while your server is running in single-process mode, HTTP's KeepAlive feature gets in the way. Netscape tries to open multiple connections and keep them open. Because there is only one server process listening, each connection has to time-out before the next succeeds. Turn off KeepAlive in httpd.conf to avoid this effect while developing or you can press **STOP** after a few seconds (assuming you use the image size params, so the Netscape will be able to render the rest of the page).

In addition you should know that when running with -X you will not see any control messages that the parent server normally writes to the error_log. (Like ''server started, server stopped and etc''.) Since httpd -X causes the server to handle all requests itself, without forking any children, there is no controlling parent to write status messages.

;o)

5 mod_perl and Relational Databases

5.1 What we will learn in this chapter

- Why Relational (SQL) Databases

- Apache::DBI - Initiate a persistent database connection

- Introduction

- Configuration

- Preopening DBI connections

- Debugging Apache::DBI

- Troubleshooting

5.2 Why Relational (SQL) Databases

Nowadays millions of people surf the Internet. There are millions of Terabytes of data lying around. To manipulate the data new smart techniques and technologies were invented. One of the major inventions was the relational database, which allows us to search and modify huge stores of data in very little time. We use **SQL** (Structured Query Language) to manipulate the contents of these databases.

When people started to use the web, they found that they needed to write web interfaces to their databases. CGI is the most widely used technology for building such interfaces. The main limitation of a CGI script driving a database is that its database connection is not persistent - on every request the CGI script has to initiate a connection to the database, and when the request is completed the connection is closed. `Apache::DBI` was written to remove this limitation. When you use it, you have a database connection which persists for the process' entire life. So when your mod_perl script needs to use a database, `Apache::DBI` provides a valid connection immediately and your script starts work right away without having to initiate a database connection first.

This is possible only with CGI running under a mod_perl enabled server, since in this model the child process does not quit when the request has been served.

It's almost as straightforward as is it sounds, there are just a few things to know about and we will cover them in this section.

5.3 Apache::DBI - Initiate a persistent database connection

This module initiates a persistent database connection. It is possible only with mod_perl.

5.3.1 Introduction

The DBI module can make use of the Apache::DBI module. When it loads, the DBI module tests if the environment variable $ENV{GATEWAY_INTERFACE} starts with CGI-Perl, and if the Apache::DBI module has already been loaded. If so, the DBI module will forward every connect() request to the Apache::DBI module. Apache::DBI uses the ping() method to look for a database handle from a previous connect() request, and tests if this handle is still valid. If these two conditions are fulfilled it just returns the database handle.

If there is no appropriate database handle or if the ping() method fails, Apache::DBI establishes a new connection and stores the handle for later re-use. When the script is run again by a child that is still connected, Apache::DBI just checks the cache of open connections by matching *host*,*username* and *password* parameters against it. A matching connection is returned if available or a new one is initiated and then returned.

There is no need to delete the disconnect() statements from your code. They won't do anything because the Apache::DBI module overloads the disconnect() method with an empty one.

When this module should be used and when shouldn't?

You will want to use this module if you are opening several database connections to the server. Apache::DBI will make them persistent per child, so if you have ten children and each opens two different connections (with different connect() arguments) you will have in total twenty opened and persistent connections. After the initial connect() you will save the connection time for every connect() request from your DBI module. This can be a huge benefit for a server with a high volume of database traffic.

You must NOT use this module if you are opening a special connection for each of your users. Each connection will stay persistent and in a short time the number of connections will be so big that your machine will scream in agony and die.

If you want to use Apache::DBI but you have both situations on one machine, at the time of writing the only solution is to run two Apache/mod_perl servers, one which uses Apache::DBI and one which does not.

5.3.2 Configuration

After installing this module, the configuration is simple - add the following directive to httpd.conf

```
  PerlModule Apache::DBI
```

Note that it is important to load this module before any other Apache*DBI module and DBI module itself!

You can skip preloading DBI, since Apache::DBI does that. But there is no harm in leaving it in, as long as it is loaded after Apache::DBI.

5.3.3 Preopening DBI connections

If you want to make sure that a connection will already be opened when your script is first executed after a server restart, then you should use the `connect_on_init()` method in the startup file to preload every connection you are going to use. For example:

```
Apache::DBI->connect_on_init
("DBI:mysql:myDB::myserver",
 "username",
 "passwd",
 {
  PrintError => 1, # warn() on errors
  RaiseError => 0, # don't die on error
  AutoCommit => 1, # commit executes immediately
 }
);
```

As noted above, use this method only if you only want all of apache to be able to connect to the database server as one user (or as a very few users).

Be warned though, that if you call `connect_on_init()` and your database is down, Apache children will be delayed at server startup, trying to connect. They won't begin serving requests until either they are connected, or the connection attempt fails. Depending on your DBD driver, this can take several minutes!

5.3.4 Debugging Apache::DBI

If you are not sure this module is working as advertised, you should enable Debug mode in the startup script by:

```
$Apache::DBI::DEBUG = 1;
```

Starting with `ApacheDBI-0.84`, setting `$Apache::DBI::DEBUG = 1` will produce only minimal output. For a full trace you set `$Apache::DBI::DEBUG = 2`.

Another approach is to add to `httpd.conf` (which does the same):

```
PerlModule Apache::DebugDBI
```

After setting the DEBUG level you will see entries in the `error_log` both when `Apache::DBI` initializes a connection and when it returns one from its cache. Use the following command to view the log in real time (your `error_log` might be located at a different path, it is set in the Apache configuration files):

```
tail -f /usr/local/apache/logs/error_log
```

I use `alias` (in `tcsh`) so I do not have to remember the path:

```
alias err "tail -f /usr/local/apache/logs/error_log"
```

5.3.5 Troubleshooting

5.3.5.1 The Morning Bug

The SQL server keeps a connection to the client open for a limited period of time. Many developers were bitten by so called **Morning bug**, when every morning the first users to use the site received a `No Data Returned` message, but after that everything worked fine. The error is caused by `Apache::DBI` returning a handle of the invalid connection (the server closed it because of a timeout), and the script was dying on that error. The infamous `ping()` method was introduced to solve this problem, but still people were being bitten by this problem. Another solution was found - to increase the timeout parameter when starting the SQL server. Currently I startup `MySQL` server with a script `safe_mysql`, so I have modified it to use this option:

```
nohup $ledir/mysqld [snipped other options] -O wait_timeout=172800
```

1. **0**

   seconds is equal to 48 hours. This change solves the problem.

Note that as from version `0.82`, `Apache::DBI` implements `ping()` inside the `eval` block. This means that if the handle has timed out it should be reconnected automatically, and avoid the morning bug.

5.3.5.2 Opening connections with different parameters

When it received a connection request, before it will decide to use an existing cached connection, `Apache::DBI` insists that the new connection be opened in exactly the same way as the cached connection. If I have one script that sets `LongReadLen` and one that does not, `Apache::DBI` will make two different connections. So instead of having a maximum of 40 open connections, I can end up with 80.

However, you are free to modify the handle immediately after you get it from the cache. So always initiate connections using the same parameters and set `LongReadLen` (or whatever) afterwards.

5.3.5.3 Debugging code which deploys DBI

To log a trace of `DBI` statement execution, you must set the `DBI_TRACE` environment variable. The `PerlSetEnv DBI_TRACE` directive must appear before you load `Apache::DBI` and `DBI`.

For example if you use `Apache::DBI`, modify your `httpd.conf` with:

```
PerlSetEnv DBI_TRACE "3=/tmp/dbitrace.log"
PerlModule Apache::DBI
```

Replace 3 with the TRACE level you want. The traces from each request will be appended to `/tmp/dbitrace.log`. Note that the logs might interleave if requests are processed concurrently.

Within your code you can control trace generation with the `trace()` method:

```
  DBI->trace($trace_level)
  DBI->trace($trace_level, $trace_filename)
```

0 disables the trace. 2 generates detailed call trace information including parameters and return values.

;o)

6 mod_perl and dbm files

6.1 What we will learn in this chapter

- Where and Why to use dbm files

- mod_perl and dbm

- Locking dbm handlers

6.2 Where and Why to use dbm files

If you need a light database, with an easy API, using simple key-value pairs to store and manipulate the records, this is a solution that should be amongst the first you consider. The maximum practical size of a dbm database depends on your hardware and the desired response times of course, but as a rough guide consider 5000 to 10000 records to be reasonable.

Some of the earliest databases implemented on Unix were dbm files, and many are still in use today. As of this writing the Berkeley DB is the most powerful dbm implementation.

With dbm, the whole database is rarely read into a memory. Combine this feature with the use of smart storage techniques, and dbm files can be manipulated much faster than their flat file brothers. Flat file databases can become very slow on insert, update and delete operations, especially when the number of records exceeds a couple of thousand. The situation is worse if you need to run a sort algorithm on a flat file.

Several different indexing algorithms can be used with dbm:

- The `HASH` algorithm gives a `O(1)` complexity of search and update, fast insert and delete, but a slow sort. (You have to do it yourself.)

- The `BTREE` algorithm allows arbitrary key/value pairs to be stored in a sorted, balanced binary tree, which allows us to get a sorted sequence of data pairs in `O(1)`, but at the expense of much slower insert, update, delete operations than is the case with `HASH`.

- The `RECNO` algorithm is more complicated, and enables both fixed-length and variable-length flat text files to be manipulated using the same key/value pair interface as in `HASH` and `BTREE`. In this case the key will consist of a record (line) number.

Most often you will want to use the `HASH` method, but your choice depends very much on your application.

dbm databases are not limited to storing key/value pairs. They can store more complicated data structures with the help of the `MLDBM` module. This module can dump and restore the whole symbol table of your script, including arrays, hashes and other complicated data structures.

6.3 mod_perl and dbm

Where does mod_perl fit into the picture?

If you are using a read only dbm file you can have it work faster if you keep it open (tied) all the time, so when your CGI script wants to access the database it is already tied and ready to be used. It will work with dynamic (read/write) databases as well but you need to use locking and data flushing to avoid data corruption.

Although mod_perl and dbm can give huge performance gains to your CGIs scripts, you should be very careful. You need to consider locking, and the consequences of `die()` and unexpected process deaths.

If your locking mechanism cannot handle dropped locks, a stale lock can deactivate your whole site. You can enter a deadlock situation if two processes simultaneously try to acquire locks on two separate databases. Each has locked only one of the databases, and cannot continue without locking the second. Yet this will never be freed because it is locked by the other process. If your processes all ask for their DB files in the same order, this situation cannot occur.

If you modify the DB you should be make very sure that you flush the data and synchronize it, especially when the process serving your CGI unexpectedly dies. In general your application should be tested very thoroughly before you put it into production to handle important data.

6.4 Locking dbm handlers

Let's make the lock status a global variable, so it will persist from request to request. If we request a lock - READ (shared) or WRITE (exclusive), we obtain the current lock status first.

If we are making a *READ* lock request, it is granted as soon as the file becomes unlocked or if it is already *READ* locked. The lock status becomes *READ* on success.

If we make a *WRITE* lock request, it is granted as soon as the file becomes unlocked. The lock status becomes *WRITE* on success.

The treatment of the *WRITE* lock request is most important.

If the DB is *READ* locked, a process that makes a WRITE request will poll until there are no reading or writing processes left. Lots of processes can successfully read the file, since they do not block each other. This means that a process that wants to write to the file (so first it needs to obtain an exclusive lock) may never get a chance to squeeze in. The following diagram represents a possible scenario where everybody can read but no one can write:

```
  [-p1-]                      [--p1--]
      [--p2--]
   [---------p3---------]
                 [------p4-----]
      [--p5--]   [----p5----]
```

The result is a starving process, which will timeout the request, and it will fail to update the DB. This is a good reason not to cache the dbm handle with dynamic dbm files. It will work perfectly with static DBM files without any need to lock files at all.

Ken Williams solved the above problem with his `Tie::DB_Lock` module, which I will present in the next section.

6.4.1 Tie::DB_Lock

`Tie::DB_Lock` ties hashes to databases using shared and exclusive locks. This module, by Ken Williams, solves the problems raised in the previous section.

The main difference from what I have described above is that `Tie::DB_Lock` copies a dbm file on read. Reading processes do not have to keep the file locked while they read it, and writing processes can still access the file while others are reading. This works best when you have lots of long-duration reading, and a few short bursts of writing.

The drawback of this module is the heavy IO performed when every reader makes a fresh copy of the DB. With big dbm files this can be quite a disadvantage and can slow the server down considerably.

An alternative would be to have one copy of the dbm image shared by all the reading processes. This can cut the number of files that are copied, and puts the responsibility of copying the read-only file on the writer, not the reader. It would need some care to make sure it does not disturb readers when putting a new read-only copy into place.

6.4.2 Locking techniques that work with dbm files

6.4.2.1 Flawed methods which must not be used

Caution: The suggested locking methods in the Camel book and DB_File man page (at least before the version 1.72) are flawed. If you use them in an environment where more than one process can modify the dbm file, it can get corrupted!!! The following is an explanation of why this happens.

You may not use a tied file's filehandle for locking, since you get the filehandle after the file has been already tied. It's too late to lock. The problem is that the database file is locked **after** it is opened. When the database is opened, the first 4k (in my dbm library) are read and then cached in memory. Therefore, a process can open the database file, cache the first 4k, and then block while another process writes to the file. If the second process modifies the first 4k of the file, when the original process gets the lock is now has an inconsistent view of the database. If it writes using this view it may easily corrupt the database on disk.

This problem can be difficult to trace because it does not cause corruption every time a process has to wait for a lock. One can do quite a bit of writing to a database file without actually changing the first 4k. But once you suspect this problem you can easily reproduce it by making your program modify the records in the first 4k of the DB.

6.4.2.2 Lock on tie (only supported by a few operating systems)

On some Operating Systems like FreeBSD, it's possible to lock on tie:

```
tie my %t, 'DB_File', $TOK_FILE, O_RDWR | O_EXLOCK, 0664;
```

and only release the lock by untieing the file. Notice the `O_EXLOCK` flag, which is not available on all Operating Systems.

6.4.2.3 DB_File::Lock

Here is `DB_File::Lock` which does the locking by using an external lockfile. This allows you to gain the lock before the file is tied. Note that it's not yet on CPAN and so is listed here in its entirety. Note also that this code still needs some testing, so **be careful** if you use it on a production machine.

```perl
package DB_File::Lock;
require 5.004;

use strict;

BEGIN {
    # RCS/CVS compliant:  must be all one line, for MakeMaker
  $DB_File::Lock::VERSION = do { my @r = (q$Revision: 1.5 $ =~ /\d+/g); sprintf "%d."."%02d" x $#r, @r };

}

use DB_File ();
use Fcntl qw(:flock O_RDWR O_CREAT);
use Carp qw(croak carp verbose);
use Symbol ();

@DB_File::Lock::ISA    = qw( DB_File );
%DB_File::Lock::lockfhs = ();

use constant DEBUG => 0;

  # file creation permissions mode
use constant PERM_MODE => 0660;

  # file locking modes
%DB_File::Lock::locks =
  (
   read  => LOCK_SH,
   write => LOCK_EX,
  );

# SYNOPSIS:
# tie my %mydb, 'DB_File::Lock', $filepath,
#     ['read' || 'write', 'HASH' || 'BTREE']
# while (my($k,$v) = each %mydb) {
#   print "$k => $v\n";
# }
# untie %mydb;
#########
sub TIEHASH {
  my $class     = shift;
  my $file      = shift;
  my $lock_mode = lc shift || 'read';
  my $db_type   = shift || 'HASH';

  die "Dunno about lock mode: [$lock_mode].\n
      Valid modes are 'read' or 'write'.\n"
    unless $lock_mode eq 'read' or $lock_mode eq 'write';
```

```
      # Critical section starts here if in write mode!

        # create an external lock
      my $lockfh = Symbol::gensym();
      open $lockfh, ">$file.lock" or die "Cannot open $file.lock for writing: $!\n";
      unless (flock $lockfh, $DB_File::Lock::locks{$lock_mode}) {
        croak "cannot flock: $lock_mode => $DB_File::Lock::locks{$lock_mode}: $!\n";
      }

      my $self = $class->SUPER::TIEHASH
        ($file,
         O_RDWR|O_CREAT,
         PERM_MODE,
         ($db_type eq 'BTREE' ? $DB_File::DB_BTREE : $DB_File::DB_HASH )
        );

        # remove the package name in case re-blessing occurs
      (my $id = "$self") =~ s/^[^=]+=//;

        # cache the lock fh
      $DB_File::Lock::lockfhs{$id} = $lockfh;

      return $self;

    } # end of sub new


    # DESTROY is automatically called when a tied variable
    # goes out of scope, on explicit untie() or when the program is
    # interrupted, e.g. with a die() call.
    #
    # It unties the db by forwarding it to the parent class,
    # unlocks the file and removes it from the cache of locks.
    ###########
    sub DESTROY{
      my $self = shift;

      $self->SUPER::DESTROY(@_);

        # now it safe to unlock the file, (close() unlocks as well). Since
        # the object has gone we remove its lock filehandler entry
        # from the cache.
      (my $id = "$self") =~ s/^[^=]+=//; # see 'sub TIEHASH'
      close delete $DB_File::Lock::lockfhs{$id};

        # Critical section ends here if in write mode!

      print "Destroying ".__PACKAGE__."\n" if DEBUG;

    }

    ####
    END {
      print "Calling the END from ".__PACKAGE__."\n" if DEBUG;

    }

    1;
```

And you use it like this:

```
    use DB_File::Lock ();
```

A simple tie, READ lock and untie

```
use DB_File::Lock ();
my $dbfile = "/tmp/test";
tie my %mydb, 'DB_File::Lock', $dbfile, 'read';
print $mydb{foo} if exists $mydb{foo};
untie %mydb;
```

You can even skip the `untie()` call. When `$mydb` goes out of scope everything will be done automatically. However it is better use the explicit call, to make sure the critical sections between lock and unlock are as short as possible. This is especially important when requesting an exclusive (write) lock.

The following example shows how it might be convenient to skip the explicit `untie()`. In this example, we don't need to save the intermediate result, we just return and the cleanup is done automatically.

```
use DB_File::Lock ();
my $dbfile = "/tmp/test";
print user_exists("stas") ? "Yes" : "No";
sub user_exists{
  my $username = shift || '';

  warn("No username passed\n"), return 0 unless $username;

  tie my %mydb, 'DB_File::Lock', $dbfile, 'read';

  # if we match the username return 1, else 0
  return $mydb{$username} ? 1 : 0;

} # end of sub user_exists
```

Now let's write all the upper case characters and their respective ASCII values to a dbm file. Then read the file and print them the contents of the DB, unsorted.

```
use DB_File::Lock ();
my $dbfile = "/tmp/test";

  # write
tie my %mydb, 'DB_File::Lock', $dbfile,'write';
for (0..26) {
  $mydb{chr 65+$_} = $_;
}
untie %mydb;

  # now, read them and printout (unsorted)
tie %mydb, 'DB_File::Lock', $dbfile;
while (my($k,$v) = each %mydb) {
  print "$k => $v\n";
}
untie %mydb;
```

If your CGI was interrupted in the middle, `DESTROY` block will take care of unlocking the dbm file and flush any changes. So your DB will be safe against possible corruption because of unclean program termination.

;o)

7 Getting Help and Further Learning

7.1 What we will learn in this chapter

- Getting help

- Get help with mod_perl

- Get help with Perl

- Get help with Perl/CGI

- Get help with Apache

- Get help with DBI

- Get help with Squid

7.2 Getting help

If after reading this guide and other documents listed in this section, you feel that your question is not yet answered, please ask the apache/mod_perl mailing list to help you. But first try to browse the mailing list archive. Most of the time you will find the answer for your question by searching the mailing archive, since there is a big chance someone else has already encountered the same problem and found a solution for it. If you ignore this advice, do not be surprised if your question will be left unanswered - it bores people to answer the same question more than once. It does not mean that you should avoid asking questions. Just do not abuse the available help and **RTFM** before you call for **HELP**. (You have certainly heard the infamous fable of the shepherd boy and the wolves)

7.3 Get help with mod_perl

- **mod_perl home**

  http://perl.apache.org

- **mod_perl Garden project**

  http://modperl.sourcegarden.org

- **mod_perl Books**

  - **'Apache Modules' Book**

    http://www.modperl.com is the home site of The Apache Modules Book, a book about creating Web server modules using the Apache API, written by Lincoln Stein and Doug MacEachern.

    Now you can purchase the book at your local bookstore or from the online dealer. O'Reilly lists this book as:

```
Writing Apache Modules with Perl and C
By Lincoln Stein & Doug MacEachern
1st Edition March 1999
1-56592-567-X, Order Number: 567X
746 pages, $34.95
```

○ **'Enabling web services with mod_perl' Book**

http://www.modperlbook.com is the home site of the new mod_perl book, that Eric Cholet and Stas Bekman are co-authoring together. We expect the book to be published in fall 2000.

Ideas, suggestions and comments are welcome. You may send them to info@modperlbook.com .

- **mod_perl Guide**

  by Stas Bekman at http://perl.apache.org/guide

- **mod_perl FAQ**

  by Frank Cringle at http://perl.apache.org/faq/ .

- **mod_perl performance tuning guide**

  by Vivek Khera at http://perl.apache.org/tuning/ .

- **mod_perl plugin reference guide**

  by Doug MacEachern at http://perl.apache.org/src/mod_perl.html .

- **Quick guide for moving from CGI to mod_perl**

  at http://perl.apache.org/dist/cgi_to_mod_perl.html .

- **mod_perl_traps, common traps and solutions for mod_perl users**

  at http://perl.apache.org/dist/mod_perl_traps.html .

- **mod_perl Quick Reference Card**

  http://www.refcards.com (Apache and other refcards are available from this link)

- **mod_perl Resources Page**

  http://www.perlreference.com/mod_perl/

- **mod_perl mailing list**

  The Apache/Perl mailing list (modperl@apache.org) **is available for mod_perl users and develop-
  ers to share ideas, solve problems and discuss things related to mod_perl and the Apache::*
  modules.** To subscribe to this list, send mail to majordomo@apache.org with empty `Subject` and

with `Body`:

```
subscribe modperl
```

A **searchable** mod_perl mailing list archive available at http://forum.swarthmore.edu/epigone/modperl . We owe it to Ken Williams.

More archives available:

- http://www.geocrawler.com/lists/3/web/182/0/

- http://www.bitmechanic.com/mail-archives/modperl/

- http://www.mail-archive.com/modperl%40apache.org/

- http://www.davin.ottawa.on.ca/archive/modperl/

- http://www.progressive-comp.com/Lists/?l=apache-modperl&r=1&w=2#apache-modperl

- http://www.egroups.com/group/modperl/

7.4 Get help with Perl

- **The Perl FAQ**

  http://www.perl.com/CPAN/doc/FAQs/FAQ/PerlFAQ.html

- **The Perl home**

  http://www.perl.com/

- **The Perl Journal**

  http://www.tpj.com/

- **Perl Module Mechanics**

  http://world.std.com/~swmcd/steven/perl/module_mechanics.html - This page describes the mechanics of creating, compiling, releasing and maintaining Perl modules.

7.5 Get help with Perl/CGI

- **Perl/CGI FAQ**

  at http://www.perl.com/CPAN/doc/FAQs/cgi/perl-cgi-faq.html

- **Answers to some bothering Perl and Perl/CGI questions**

  http://www.singlesheaven.com/stas/TULARC/webmaster/myfaq.html

- **Idiot's Guide to CGI programming**

  http://www.perl.com/CPAN/doc/FAQs/cgi/idiots-guide.html

- **WWW Security FAQ**

  http://www.w3.org/Security/Faq/www-security-faq.html

- **CGI/Perl Taint Mode FAQ**

  http://www.gunther.web66.com/FAQS/taintmode.html (by Gunther Birznieks)

7.6 Get help with Apache

- **Apache Project's Home**

  http://www.apache.org

- **Apache Quick Reference Card**

  http://www.refcards.com (Apache and other refcards are available from this link)

- **The Apache FAQ**

  http://www.apache.org/docs/misc/FAQ.html

- **Apache Server Documentation**

  http://www.apache.org/docs/

- **Apache Handlers**

  http://www.apache.org/docs/handler.html

- **mod_rewrite Guide**

  http://www.engelschall.com/pw/apache/rewriteguide/

7.7 Get help with DBI

- **Perl DBI examples**

http://www.saturn5.com/~jwb/dbi-examples.html (by Jeffrey William Baker).

- **DBI Homepage**

  http://www.symbolstone.org/technology/perl/DBI/

- **DBI mailing list information**

  http://www.fugue.com/dbi/

- **DBI mailing list archives**

  http://outside.organic.com/mail-archives/dbi-users/ http://www.xray.mpe.mpg.de/mailing-lists/dbi/

- **Persistent connections with mod_perl**

  http://perl.apache.org/src/mod_perl.html#PERSISTENT_DATABASE_CONNECTIONS

7.8 Get help with Squid - Internet Object Cache

- Home page - http://squid.nlanr.net/

- FAQ - http://squid.nlanr.net/Squid/FAQ/FAQ.html

- Users Guide - http://squid.nlanr.net/Squid/Users-Guide/

- Mailing lists - http://squid.nlanr.net/Squid/mailing-lists.html

;o)

Table of Contents: