# ApacheCon Europe
# October 24, 2000
# London, UK

## Tutorial: Getting Started with mod_perl

by **Stas Bekman**
http://stason.org/
<stas@stason.org>
JazzValley.com, CTO

Master Copy URL: http://perl.apache.org/guide/

This document is originally written in **POD**, converted to **HTML**, **PostScript** and **PDF** by `Pod::HtmlPsPdf` Perl module.

You can download all the HTML files, POD sources and build scripts from http://www.perl.com/CPAN-local/authors/id/S/ST/STAS/ or at the same directory from one of your favorite CPAN mirrors.
The latest CVS snapshot is available from http://www.stason.org/guide-snapshots/.
The PDF version of the Guide is available from http://perl.apache.org/guide/.

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

**Your corrections of the technical and grammatical errors are very welcome. You are encouraged to help me improve the guide the tutorial is based on.**

**If you have something to contribute please send it directly to me. Thank you!**

**If it's a small fix, just remember to tell me the chapter and the name of the section you are talking about.**

**If it includes big text modifications please download the POD sources of the Guide from CPAN http://www.perl.com/CPAN-local/authors/id/S/ST/STAS/, edit them directly and send the patched file to me. There is no need to send me patches, send the POD files themselves.**

**Thank you!**

# 1   Agenda

# 1.1  Agenda

- I will start the presentation with a very basic introduction into mod_perl, 10 lines installation instructions, a simple configuration and a few code examples. These should help you get your feet wet if you are really new to mod_perl.

- Afterwards I'll talk about the machine setups most popular servers use. I'll explain the incentives for having the light Apache and the heavy mod_perl servers serving different kinds of requests. We will see the two major setups, one using squid as a front-end machine, and the other plain Apache server with mod_proxy.

- Finally we will see some mod_perl peculiarities you should know about, will talk about the modules which allows you to run your CGI scripts unaltered.

- The are two more sections left for the post-conference reading. The first one is a Perl reference. It's talking about Perl stuff which is very important to know when coding for mod_perl. And the other one includes additional information about mod_perl and related products resources. You should use it to find your way to find the answer to the questions that you might need to get answered, on your way to becoming a mod_perl guru or when you need some general help.

;o)

# 2   mod_perl in Four Slides

## 2.1  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

## 2.2  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/ as of March 2000 about 612425 hosts use mod_perl. Doug MacEachern fathered the core code of this Apache module and licensed it under the Apache Software License.

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*...

## 2.3  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.

# 2.4  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`.

# 2.5  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:
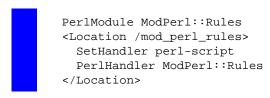
```
mod_perl rules!
```

## 2.6  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
----------------
package ModPerl::Rules;
use Apache::Constants;

sub handler{
  my $r = shift;
  $r->send_http_header('text/plain');
  print "mod_perl rules!\n";
  return OK;
}
1;
```

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/mod_perl_rules
```

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

```
mod_perl rules!
```

as the response.

## 2.7  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 this titorial you will find a chapter describing the available resources and pointers to them.

;o)

# 3   Server Setup Strategies

## 3.1  What we will learn in this chapter

- mod_perl Deployment Overview

- Standalone mod_perl Enabled Apache Server

- One Plain Apache and One mod_perl-enabled Apache Servers

- Adding a Proxy Server in http Accelerator Mode

- Implementations of Proxy Servers

## 3.2  mod_perl Deployment Overview

There are several different ways to build, configure and deploy your mod_perl enabled server. Some of them are:

1. Having one binary and one configuration file (one big binary for mod_perl).

2. Having two binaries and two configuration files (one big binary for mod_perl and one small binary for static objects like images.)

3. Any of the above plus a reverse proxy server in http accelerator mode.

If you are a newbie, I would recommend that you start with the first option and work on getting your feet wet with apache and mod_perl. Later, you can decide whether to move to the second one which allows better tuning at the expense of more complicated administration, or to the third option which gives you even more power.

1. The first option will kill your production site if you serve a lot of static data from large (4 to 15MB) webserver processes. On the other hand, while testing you will have no other server interaction to mask or add to your errors.

2. This option allows you to tune the two servers individually, for maximum performance.

   However, you need to choose between running the two servers on multiple ports, multiple IPs, etc., and you have the burden of administering more than one server. You have to deal with proxying or fancy site design to keep the two servers in synchronization.

3. The third option (proxy in http accelerator mode), once correctly configured and tuned, improves the performance of any of the above three options by caching and buffering page results.

## 3.3  Standalone mod_perl Enabled Apache Server

The first approach is to implement a straightforward mod_perl server. Just take your plain apache server and add mod_perl, like you add any other apache module. You continue to run it at the port it was running before. You probably want to try this before you proceed to more sophisticated and complex techniques.

The advantages:

- Simplicity. You just follow the installation instructions, configure it, restart the server and you are done.

- No network changes. You do not have to worry about using additional ports as we will see later.

- Speed. You get a very fast server, you see an enormous speedup from the first moment you start to use it.

The disadvantages:

- The process size of a mod_perl-enabled Apache server is huge (maybe 4Mb at startup and growing to 10Mb and more, depending on how you use it) compared to the typical plain Apache. Of course if memory sharing is in place, RAM requirements will be smaller.

  You probably have a few tens of child processes. The additional memory requirements add up in direct relation to the number of child processes. Your memory demands are growing by an order of magnitude, but this is the price you pay for the additional performance boost of mod_perl. With memory prices so cheap nowadays, the additional cost is low -- especially when you consider the dramatic performance boost mod_perl gives to your services with every 100Mb of RAM you add.

  While you will be happy to have these monster processes serving your scripts with monster speed, you should be very worried about having them serve static objects such as images and html files. Each static request served by a mod_perl-enabled server means another large process running, competing for system resources such as memory and CPU cycles. The real overhead depends on static objects request rate. Remember that if your mod_perl code produces HTML code which includes images, each one will turn into another static object request. Having another plain webserver to serve the static objects solves this unpleasant obstacle. Having a proxy server as a front end, caching the static objects and freeing the mod_perl processes from this burden is another solution. We will discuss both below.

- Another drawback of this approach is that when serving output to a client with a slow connection, the huge mod_perl-enabled server process (with all of its system resources) will be tied up until the response is completely written to the client. While it might take a few milliseconds for your script to complete the request, there is a chance it will be still busy for some number of seconds or even minutes if the request is from a slow connection client. As in the previous drawback, a proxy solution can solve this problem. More on proxies later.

  Proxying dynamic content is not going to help much if all the clients are on a fast local net (for example, if you are administering an Intranet.) On the contrary, it can decrease performance. Still, remember that some of your Intranet users might work from home through slow modem links.

If you are new to mod_perl, this is probably the best way to get yourself started.

And of course, if your site is serving only mod_perl scripts (close to zero static objects, like images), this might be the perfect choice for you!

# 3.4  One Plain Apache and One mod_perl-enabled Apache Servers

As I have mentioned before, when running scripts under mod_perl, you will notice that the httpd processes consume a huge amount of virtual memory, from 5Mb to 15Mb and even more. That is the price you pay for the enormous speed improvements under mod_perl. (Again -- shared memory keeps the real memory that is being used much smaller :)

Using these large processes to serve static objects like images and html documents is overkill. A better approach is to run two servers: a very light, plain apache server to serve static objects and a heavier mod_perl-enabled apache server to serve requests for dynamic (generated) objects (aka CGI).

From here on, I will refer to these two servers as **httpd_docs** (vanilla apache) and **httpd_perl** (mod_perl enabled apache).

The advantages:

- The heavy mod_perl processes serve only dynamic requests, which allows the deployment of fewer of these large servers.

- `MaxClients`, `MaxRequestsPerChild` and related parameters can now be optimally tuned for both `httpd_docs` and `httpd_perl` servers, something we could not do before. This allows us to fine tune the memory usage and get a better server performance.

  Now we can run many lightweight `httpd_docs` servers and just a few heavy `httpd_perl` servers.

An **important** note: When a user browses static pages and the base URL in the **Location** window points to the static server, for example `http://www.nowhere.com/index.html` -- all relative URLs (e.g. `<A HREF="/main/download.html">`) are being served by the light plain apache server. But this is not the case with dynamically generated pages. For example when the base URL in the **Location** window points to the dynamic server -- (e.g. `http://www.nowhere.com:8080/perl/index.pl`) all relative URLs in the dynamically generated HTML will be served by the heavy mod_perl processes. You must use fully qualified URLs and not relative ones! `http://www.nowhere.com/icons/arrow.gif` is a full URL, while `/icons/arrow.gif` is a relative one. Using `<BASE HREF="http://www.nowhere.com/">` in the generated HTML is another way to handle this problem. Also the `httpd_perl` server could rewrite the requests back to `httpd_docs` (much slower) and you still need the attention of the heavy servers. This is not an issue if you hide the internal port implementations, so the client sees only one server running on port `80`.

The disadvantages:

- An administration overhead.

○ The need for two different sets of configuration, log and other files. We need a special directory layout to manage these. While some directories can be shared between the two servers (like the `include` directory, containing the apache include files -- assuming that both are built from the same source distribution), most of them should be separated and the configuration files updated to reflect the changes.

○ The need for two sets of controlling scripts (startup/shutdown) and watchdogs.

○ If you are processing log files, now you probably will have to merge the two separate log files into one before processing them.

• Just as in the one server approach, we still have the problem of a mod_perl process spending its precious time serving slow clients, when the processing portion of the request was completed a long time ago. Deploying a proxy solves this, and will be covered in the next section.

As with the single server approach, this is not a major disadvantage if you are on a fast network (i.e. Intranet). It is likely that you do not want a buffering server in this case.

# 3.5  Adding a Proxy Server in http Accelerator Mode

At the beginning there were 2 servers: one plain apache server, which was *very light*, and configured to serve static objects, the other mod_perl enabled (*very heavy*) and configured to serve mod_perl scripts. We named them `httpd_docs` and `httpd_perl` respectively.

The two servers coexist at the same IP address by listening to different ports: `httpd_docs` listens to port 80 (e.g. http://www.nowhere.com/images/test.gif) and `httpd_perl` listens to port 8080 (e.g. http://www.nowhere.com:8080/perl/test.pl). Note that I did not write http://www.nowhere.com:80 for the first example, since port 80 is the default port for the http service. Later on, I will be changing the configuration of the `httpd_docs` server to make it listen to port 81.

Now I am going to convince you that you **want** to use a proxy server (in the http accelerator mode). The advantages are:

• Allow serving of static objects from the proxy's cache (objects that previously were entirely served by the `httpd_docs` server).

• You get less I/O activity reading static objects from the disk (proxy serves the most ''popular'' objects from RAM - of course you benefit more if you allow the proxy server to consume more RAM). Since you do not wait for the I/O to be completed you are able to serve static objects much faster.

• The proxy server acts as a sort of output buffer for the dynamic content. The mod_perl server sends the entire response to the proxy and is then free to deal with other requests. The proxy server is responsible for sending the response to the browser. So if the transfer is over a slow link, the mod_perl server is not waiting around for the data to move.

Using numbers is always more convincing :) Let's take a user connected to your site with 56 kbps (bps == bits/sec) modem. 1 byte == 8 bits. It means that the speed of the user's link is 56/8 = 7 kbytes/sec. I assume an average generated HTML page to be of 42Kb (Kb == kilobytes) and an average script that generates this output in 0.5 second. How long will the server wait before the user gets the whole output response? A simple calculation reveals pretty scary numbers -- it will have to wait for another 12 secs (42kb/0.5*7Kb), when it could serve another 11 (12/1-1) dynamic requests in this time.

This very simple example shows us that we need only one twelfth the number of children running, which means that we will need only one twelfth of the memory (not quite true because some parts of the code are shared).

But you know that nowadays scripts often return pages which are blown up with javascript code and similar, which can make them of 100kb size and the download time will be of the order of... (This calculation is left to you as an exercise :)

Many users like to open many browser windows and do many things at once (download files and browse graphically *heavy* sites). So the speed of 7K/sec we were assuming before, may often be 5-10 times slower.

- We are going to hide the details of the server's implementation. Users will never see ports in the URLs (more on that topic later). You can have a few boxes serving the requests, and only one serving as a front end, which spreads the jobs between the servers in a way that you can control. You can actually shut down a server, without the user even noticing, because the front end server will dispatch the jobs to other servers.

- For security reasons, using any httpd accelerator (or a proxy in httpd accelerator mode) is essential because you do not let your internal server get directly attacked by arbitrary packets from whomever. The httpd accelerator and internal server communicate in expected HTTP requests. This allows for only your public ''bastion'' accelerating www server to get hosed in a successful attack, while leaving your internal data safe.

The disadvantages are:

- Of course there are drawbacks. Luckily, these are not functionality drawbacks, but they are more administration hassle. You have another daemon to worry about, and while proxies are generally stable, you have to make sure to prepare proper startup and shutdown scripts, which are run at boot and reboot as appropriate. Also, you might want to set up the crontab to run a watchdog script.

- Proxy servers can be configured to be light or heavy, the admin must decide what gives the highest performance for his application. A proxy server like Squid is light in the concept of having only one process serving all requests. But it can appear pretty heavy when it loads objects into memory for faster service.

Have I succeeded in convincing you that you want a proxy server?

If you are on a local area network (LAN), then the big benefit of the proxy buffering the output and feeding a slow client is gone. You are probably better off sticking with a straight mod_perl server in this case.

# 3.6  Implementations of Proxy Servers

As of this writing, two proxy implementations are known to be widely used with mod_perl - **squid** proxy server and **mod_proxy** which is a part of the apache server. Let's compare them.

## 3.6.1  The Squid Server

The Advantages:

- Caching of static objects. These are served much faster, assuming that your cache size is big enough to keep the most frequently requested objects in the cache.

- Buffering of dynamic content, by taking the burden of returning the content generated by mod_perl servers to slow clients, thus freeing mod_perl servers from waiting for the slow clients to download the data. Freed servers immediately switch to serve other requests, thus your number of required servers goes down dramatically.

- Non-linear URL space / server setup. You can use Squid to play some tricks with the URL space and/or domain based virtual server support.

The Disadvantages:

- Proxying dynamic content is not going to help much if all the clients are on a fast local net. Also, a message on the squid mailing list implied that squid only buffers in 16k chunks so it would not allow a mod_perl to complete immediately if the output is larger.

- Speed. Squid is not very fast today when compared with the plain file based web servers available. Only if you are using a lot of dynamic features such as mod_perl or similar is there a reason to use Squid, and then only if the application and the server are designed with caching in mind.

- Memory usage. Squid uses quite a bit of memory.

- HTTP protocol level. Squid is pretty much a `HTTP/1.0` server, which seriously limits the deployment of `HTTP/1.1` features.

- HTTP headers, dates and freshness. The squid server might give out stale pages, confusing downstream/client caches.(You update some documents on the site, but squid will still serve the old ones.)

- Stability. Compared to plain web servers, Squid is not the most stable.

The pros and cons presented above lead to the idea that you might want to use squid for its dynamic content buffering features, but only if your server serves mostly dynamic requests. So in this situation, when performance is the goal, it is better to have a plain apache server serving static objects, and squid

proxying the mod_perl enabled server only.

## 3.6.2  Apache's mod_proxy

I do not think the difference in speed between apache's **mod_proxy** and **squid** is relevant for most sites, since the real value of what they do is buffering for slow client connections. However, squid runs as a single process and probably consumes fewer system resources.

The trade-off is that mod_rewrite is easy to use if you want to spread parts of the site across different back end servers, while mod_proxy knows how to fix up redirects containing the back-end server's idea of the location. With squid you can run a redirector process to proxy to more than one back end, but there is a problem in fixing redirects in a way that keeps the client's view of both server names and port numbers in all cases.

The difficult case is where:

- **You have DNS aliases that map to the same IP address and**

- **You want the redirect to port 80 and**

- **The server is on a different port and**

- **You want to keep the specific name the browser has already sent, so that it does not change in the client's Location window.**

The Advantages:

- No additional server is needed. We keep the one plain plus one mod_perl enabled apache servers. All you need is to enable `mod_proxy` in the `httpd_docs` server and add a few lines to `httpd.conf` file.

- The `ProxyPass` and `ProxyPassReverse` directives allow you to hide the internal redirects, so if `http://nowhere.com/modperl/` is actually `http://localhost:81/modperl/`, it will be absolutely transparent to the user. `ProxyPass` redirects the request to the mod_perl server, and when it gets the response, `ProxyPassReverse` rewrites the URL back to the original one, e.g:

  ```
  ProxyPass        /modperl/ http://localhost:81/modperl/
  ProxyPassReverse /modperl/ http://localhost:81/modperl/
  ```

- It does mod_perl output buffering like squid does.

- It even does caching. You have to produce correct `Content-Length`, `Last-Modified` and `Expires` http headers for it to work. If some of your dynamic content does not change frequently, you can dramatically increase performance by caching it with `ProxyPass`.

- `ProxyPass` happens before the authentication phase, so you do not have to worry about authenticating twice.

- Apache is able to accelerate secure HTTP requests completely, while also doing accelerated HTTP. With Squid you have to use an external redirection program for that.

- The latest (apache 1.3.6 and later) Apache proxy accelerated mode is reported to be very stable.

;o)

# 4   Porting from CGI Scripts and mod_perl Coding Guidelines.

## 4.1  What we will learn in this chapter

- 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

- More package name related issues

- \_\_END\_\_ and \_\_DATA\_\_ tokens

- Output from system calls

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

- `die()` and mod_perl

- Apache::print() and CORE::print()

- Global Variables Persistance

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

## 4.2  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 browser 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.

## 4.2.1  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 named - as opposed to anonymous - subroutine defined inside another subroutine) that refers to a lexically scoped variable defined outside this nested subroutine. This effect is explained in

the Perl Reference section at the end of this handout.

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 won'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 this 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:

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

Secondly, *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 code in the *error.log* wasn't indented. I've indented it for you to stress that the code was wrapped inside the `handler()` subroutine.

What do we learn from this?

Well firstly 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 `::` and . with `_2e`. 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`.

If we were to add `use diagnostics` to the script we would also see a reference in the error text to 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();
}
```

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 for Inner Subroutines* in the Perl Reference chapter discusses other 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 = CGI->new;
my $name = $q->param('name');
print_response();

sub print_response{
  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.

## 4.2.2  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 your 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--and on a production server you'll soon run out of disk space if your site is popular.

Of course none of the warnings will be reported if the warning mechanism is not turned **On**.

# 4.3 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 is the result 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.

## *4.3.1 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.

# 4.4 @INC and mod_perl

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
      ```

  Note that this setting will be ignored if you have the `PerlTaintMode` mode turned on.

- 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
```

# 4.5 Reloading Modules and Required Files

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:

## 4.5.1 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.

## 4.5.2 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 immediately reloads any files that 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 (the `Apache::Status` module), fetch http://www.example.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.

# 4.6  Name collisions with Modules and libs

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 on 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:

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

Where some sample code could be:

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

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

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

```
./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 single server mode 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 libraries (i.e. *my-lib.pl"*) you require() as well:

Suppose that you have a directory structure like this:

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

and both scripts contain:

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

while *./tool1/config.pl* can be something like this:

```
$foo = 0;
1;
```

and *./tool2/config.pl*:

```
$foo = 1;
1;
```

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!

```
./tool/config.pl
./tool/tool1.pl
./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 structure so that they have different path prefixes. The file system layout will be something like:

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

And modify the scripts:

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

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

```
./tool1/tool1-lib/config.pl
./tool1/tool1.pl
./tool2/tool2-lib/config.pl
./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 lose 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 name in the required files! It should be unique in relation 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 though a package may not exist in the standard distribution now, a package may come along in a later distribution which collides with a name you've chosen. Using a two part package name will help avoid this problem.

Even a better approach is to use three level naming, like `CompanyName::Project-Name::Module`, which is most unlikely to have conflicts with later Perl releases. 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.

## 4.7  More package name related issues

If you have the following:

```
    PerlHandler Apache::Work::Foo
    PerlHandler Apache::Work::Foo::Bar
```

And you make a request that pulls in `Apache/Work/Foo/Bar.pm` first, then the `Apache::Work::Foo` package gets defined, so mod_perl does not try to pull in `Apache/Work/Foo.pm`

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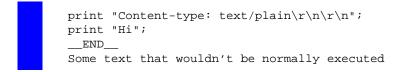
```
    print "Content-type: text/plain\r\n\r\n";
    print "Hi";
```

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.

# 4.9  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. It's best not to fork at all if you can avoid it.

# 4.10  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 purpose 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 = $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 always be 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 the `@EXPORT_OK` list 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 the `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 mod_perl, 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(Apache::Constants::DONE)` will cause the server to exit gracefully, completing the logging functions and protocol requirements etc. ( Apache::Constants::DONE == -2, Apache::Constants::OK == 0.)

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.

In an `Apache::Registry` script you would do:

```
    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.

Even if you don't need to call `child_terminate()` at the end of the request if you want the process to quit afterwards, here is an example of assigning the postprocessing handler. You might do this if you wanted to execute your own code a moment before the process quits.

```
    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* and `GTop.pm`) 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.

# 4.11  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 is unable to open the file, by aborting the script execution, printing the death reason and quitting the Perl interpreter.

You will hardly 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 the like.

A 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 was aborted by a `die()` statement.

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

Here is an example of such trapping code, although it isn't the real code:

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

# 4.12  Apache::print() and CORE::print()

Under mod_perl `CORE::print()` will redirect its data to `Apache::print()` since the STDOUT filehandle is tied to the *Apache* module. This allows us to run CGI scripts unmodified under `Apache::Registry` by chaining the output of one content handler to the input of the other handler.

`Apache::print()` behaves mostly like the built-in *print()* function. In addition it sets a timeout so that if the client connection is broken the handler won't wait forever trying to print data downstream to the client.

There is also an optimization built into `Apache::print()`. If any of the arguments to the method are scalar references to strings, they are automatically dereferenced for you. This avoids needless copying of large strings when passing them to subroutines. For example:

```
$long_string = "A" x 10000000;
$r->print(\$long_string);
```

# 4.13  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 code development harder and you will have to make certain that all the variables are initialized before they are used. Use `my()` scoped variables wherever you can.

You should be especially careful with Perl special variables which cannot be lexically scoped. You have to use `local()` instead.

# 4.14  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 the 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 which allows them to be set/unset in code. Consult the `perlvar` manpage for more details.

## *4.14.1  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
    }
    # $^W assumes its previous value here
    ```

    This turns it **Off**:

```
{
  local $^W = 0;
  # some code
}
# $^W assumes its previous value here
```

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 essential for a development server, you should turn it globally **Off** in a production server, since, for example, 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.

## *4.14.2  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)

If you have some scripts that won't run under Taint mode, run only the ones that run under mod_perl with Taint mode enabled and the rest on another server with Taint mode disabled -- this can be either a mod_cgi in the front-end server or another back-end mod_perl server. You can use the mod_rewrite module and redirect requests based on the file extensions. For example you can use *.tcgi* for the taint-clean scripts, and *cgi* for the rest.

When you have this setup you can start working toward cleaning the rest of the scripts, to make them run under the Taint mode. Just because you have a few dirty scripts doesn't mean that you should jeopardize your whole service.

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.

## 4.14.3  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`. Switches in this variable are treated as if they were on every Perl command line.

Only the `-[DIMUdmw]` switches are allowed.

When the `PerlTaintCheck` variable is turned on, the value of `PERL5OPT` will be ignored.

;o)

# 5   Perl Reference

## 5.1  What we will learn in this chapter

- perldoc's Rarely Known But Very Useful Options

- Tracing Warnings Reports

- Variables Globally, Lexically Scoped And Fully Qualified

- `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

- Exception Handling for mod_perl

## 5.2  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
```

# 5.3  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.

# 5.4  Variables Globally, Lexically Scoped And Fully Qualified

Also see the clarification of `my()` vs. `use vars` - Ken Williams writes:

> Yes, there is quite a bit of difference!  With use vars(), you are
> making an entry in the symbol table, and you are telling the
> compiler that you are going to be referencing that entry without an
> explicit package name.
>
> With my(), NO ENTRY IS PUT IN THE SYMBOL TABLE.  The compiler
> figures out C<at compile time> which my() variables (i.e. lexical
> variables) are the same as each other, and once you hit execute time
> you cannot go looking those variables up in the symbol table.

And `my()` vs. `local()` - Randal Schwartz writes:

> local() creates a temporal-limited package-based scalar, array,
> hash, or glob -- when the scope of definition is exited at runtime,
> the previous value (if any) is restored.  References to such a
> variable are *also* global... only the value changes.  (Aside: that
> is what causes variable suicide. :)
>
> my() creates a lexically-limited non-package-based scalar, array, or
> hash -- when the scope of definition is exited at compile-time, the
> variable ceases to be accessible.  Any references to such a variable
> at runtime turn into unique anonymous variables on each scope exit.

# 5.5  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.

## 5.5.1  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);
```
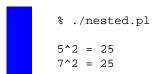
Don't let the weird subroutine names to fool you, the `print_power_of_2()` subroutine should print the square of the passed number. Let's run the code and see whether it works:

```
% ./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.

## 5.5.2  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.

## 5.5.3  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);
```
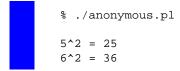
Now `$func_ref` contains a reference to an anonymous function, which we later use when we need to get the power of two. (In Perl, a function is the same thing as a subroutine.) Since it is anonymous, the function will automatically be rebound to the new value of the outer scoped variable $x, and the results will now be as expected.

Let's verify:

```
% ./anonymous.pl

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

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

# 5.6  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.

## 5.6.1  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() (lexically 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.

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

## 5.7.1  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.

## 5.7.2  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 won't work under mod_perl, since it's a module and as any module it's loaded only once. So the first script using it will have all the settings correct, but the rest of the scripts will not if located in a different directory from the first.

For a completeness of this section, I'll present this module anyway.

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

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

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

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

```
% /tmp/load.pl

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

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

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

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

You will have a slight overhead because you will load from disk and recompile the `FindBin` module on each request. So it can be not worth it.

## *5.7.3  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;
```

## *5.7.4  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.

## 5.7.5  *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()`.

## 5.7.6  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.

# 5.8  Using Global Variables and Sharing Them Between Modules/Packages

## 5.8.1  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.

## 5.8.2  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.

## 5.8.3  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;
-------------------
```

## 5.8.4  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.

## 5.9  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.

# 5.10  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";
  }
  # ...
}
```

# 5.11  Exception Handling for mod_perl

Provided here are some guidelines for clean(er) exception handling for mod_perl usage, although the technique presented here applies to all of your Perl programming.

The reasoning behind this document is the current broken status of $SIG{__DIE__} in the perl core - see both the perl5-porters and mod_perl mailing list archives for details on this discussion. (It's broken in at least Perl v5.6.0 and probably in later versions as well.)

## 5.11.1  Trapping Exceptions in Perl

To trap an exception in Perl we use the `eval{}` construct. Many people initially make the mistake that this is the same as the `eval EXPR` construct, which compiles and executes code at run time, but that's not the case. `eval{}` compiles at compile time, just like the rest of your code, and has next to zero run-time penalty.

When in an eval block, if the code executing `die()`'s for some reason, rather than terminating your code, the exception is *caught* and the program is allowed to examine that exception and make decisions based on it. The full construct looks like this:

```
eval
{
    # Some code here
}; # Note important semi-colon there
if ($@) # $@ contains the exception that was thrown
{
    # Do something with the exception
}
else # optional
{
    # No exception was thrown
}
```

Most of the time when you see these exception handlers there is no else block, because it tends to be OK if the code didn't throw an exception.

## 5.11.2  Alternative Exception Handling Techniques

An often suggested method for handling global exceptions in mod_perl, and other perl programs in general, is a __**DIE**__ handler, which can be setup by either assigning a function name as a string to `$SIG{__DIE__}` (not particularly recommended, because of the possible namespace clashes) or assigning a code reference to `$SIG{__DIE__}`, the usual way of doing so is to use an anonymous subroutine:

```
$SIG{__DIE__} = sub { print "Eek - we died with:\n", $_[0]; };
```

The current problem with this is that `$SIG{__DIE__}` is a global setting in your script, so while you can potentially hide away your exceptions in some external module, the execution of `$SIG{__DIE__}` is fairly magical, and interferes not just with your code, but with all code in every module you import. Beyond the magic involved, `$SIG{__DIE__}` actually interferes with perl's normal exception handling mechanism, the `eval{}` construct. Witness:

```
$SIG{__DIE__} = sub { print "handler\n"; };

eval {
    print "In eval\n";
    die "Failed for some reason\n";
};
if ($@) {
    print "Caught exception: $@";
}
```

The code unfortunately prints out:

```
In eval
handler
```

Which isn't quite what you would expect, especially if that `$SIG{__DIE__}` handler is hidden away deep in some other module that you didn't know about. There are work arounds however. One is to localise `$SIG{__DIE__}` in every exception trap you write:

```
eval {
    local $SIG{__DIE__};
    ...
};
```

Obviously this just doesn't scale - you don't want to be doing that for every exception trap in your code, and it's a slow down. A second work around is to check in your handler if you are trying to catch this exception:

```
$SIG{__DIE__} = sub {
    die $_[0] if $^S;
    print "handler\n";
};
```

However this won't work under `Apache::Registry` - you're always in an eval block there!

You should warn people about this danger of `$SIG{__DIE__}` and inform them of better ways to code. The following material is an attempt to just that.

## 5.11.3  Better Exception Handling

The `eval{}` construct in itself is a fairly weak way to handle exceptions as strings. There's no way to pass more information in your exception, so you have to handle your exception in more than one place - at the location the error occurred, in order to construct a sensible error message, and again in your exception handler to de-construct that string into something meaningful (unless of course all you want your exception handler to do is dump the error to the browser).

A little known fact about exceptions in perl 5.005 is that you can call die with an object. The exception handler receives that object in $@. This is how you are advised to handle exceptions now, as it provides an extremely flexible and scalable exceptions solution.

## 5.11.3.1 A Little Housekeeping

First though, before we delve into the details, a little housekeeping is in order. Most, if not all, mod_perl programs consist of a main routine that is entered, and then dispatches itself to a routine depending on the parameters passed and/or the form values. In a normal C program this is your `main()` function, in a mod_perl handler this is your `handler()` function/method.

In order for you to be able to use exception handling to its best extent you need to change your script to have some sort of global exception handling. This is much more trivial than it sounds. If you're using `Apache::Registry` to emulate CGI you might consider wrapping your entire script in one big eval block, but I would discourage that. A better method would be to modularise your script into discrete function calls, one of which should be a dispatch routine:

```
#!/usr/bin/perl -w
# Apache::Registry script

eval {
    dispatch();
};
catch($@);

sub dispatch {
    ...
}

sub catch {
    my $exception = shift;
    ...
}
```

This is easier with an ordinary mod_perl handler as it is natural to have separate functions, rather than a long run-on script:

```
MyHandler.pm
------------
sub handler {
    my $r = shift;

    eval {
        dispatch($r);
    };
    catch ($@);
}

sub dispatch {
    my $r = shift;
    ...
}

sub catch {
    my $exception = shift;
    ...
}
```

Now that the skeleton code is setup, let's create an exception class, making use of Perl 5.005's ability to throw exception objects.

## 5.11.3.2  An Exception Class

This is a really simple exception class, that does nothing but contain information. A better implementation would probably also handle its own exception conditions, but that would be more complex, requiring separate packages for each exception type.

```
My/Exception.pm
---------------
package My::Exception;

sub AUTOLOAD {
    my ($package, $filename, $line) = caller;
    no strict 'refs', 'subs';
    if ($AUTOLOAD =~ /.*::([A-Z]\w+)$/) {
        my $exception = $1;
        *{$AUTOLOAD} =
            sub {
                shift;
                push @_, caller => {
                                    package => $package,
                                    filename => $filename,
                                    line => $line,
                                    };
                bless { @_ }, "My::Exception::$exception";
            };
        goto &{$AUTOLOAD};
    }
    else {
        die "No such exception class: $AUTOLOAD\n";
    }
}

1;
```

OK, so this is all highly magical, but what does it do? It creates a simple package that we can import and use as follows:

```
use My::Exception;

die My::Exception->SomeException( foo => "bar" );
```

The exception class tracks exactly where we died from using the `caller()` mechanism, it also caches exception classes so that AUTOLOAD is only called the first time (in a given process) an exception of a particular type is thrown (particularly relevant under mod_perl).

## 5.11.4  Catching Uncaught Exceptions

What about exceptions that are thrown outside of your control? We can fix this using one of two possible methods. The first is to override die globally using the old magical $SIG{__DIE__}, and the second, is the cleaner non-magical method of overriding the global die() method to your own die() method that throws an exception that makes sense to your application.

## 5.11.4.1 Using $SIG{__DIE__}

Overloading using `$SIG{__DIE__}` in this case is rather simple, here's some code:

```
$SIG{__DIE__} = sub {
    my $err = shift;
    if(!ref $err) {
        $err = My::Exception->UnCaught(text => $err);
    }
    die $err;
};
```

All this does is catch your exception and re-throw it. It's not as dangerous as we stated earlier that `$SIG{__DIE__}` can be, because we're actually re-throwing the exception, rather than catching it and stopping there.

There's only one slight buggette left, and that's if some external code `die()`'ing catches the exception and tries to do string comparisons on the exception, as in:

```
eval {
    ... # some code
    die "FATAL ERROR!\n";
};
if ($@) {
    if ($@ =~ /^FATAL ERROR/) {
        die $@;
    }
}
```

In order to deal with this, we can overload stringification for our `My::Exception::UnCaught` class:

```
{
    package My::Exception::UnCaught;
    use overload '""' => \&str;

    sub str {
        shift->{text};
    }
}
```

We can now let other code happily continue.

## 5.11.4.2 Overriding the Core die() Function

So what if we don't want to touch `$SIG{__DIE__}` at all? We can overcome this by overriding the core die function. This is slightly more complex than implementing a `$SIG{__DIE__}` handler, but is far less magical, and is the right thing to do, according to the perl5-porters mailing list.

Overriding core functions has to be done from an external package/module. So we're going to add that to our `My::Exception` module. Here's the relevant parts:

```
use vars qw/@ISA @EXPORT/;
use Exporter;

@EXPORT = qw/die/;
@ISA = 'Exporter';

sub import {
    my $pkg = shift;
    $pkg->export('CORE::GLOBAL', 'die');
    Exporter::import($pkg,@_);
}

sub die {
    if (!ref($_[0])) {
        CORE::die My::Exception->UnCaught(text => join('', @_));
    }
    CORE::die $_[0];
}
```

That wasn't so bad, was it? We're relying on Exporter's export function to do the hard work for us, exporting the `die()` function into the `CORE::GLOBAL` namespace. Along with the above overloaded stringification, we now have a complete exception system (well, mostly complete. Exception die-hards would argue that there's no ''finally'' clause, and no exception stack, but that's another topic for another time).

## 5.11.5  Some Uses

I'm going to come right out and say now: I abuse this system horribly! I throw exceptions all over my code, not because I've hit an exceptional bit of code, but because I want to get straight back out of the current function, without having to have every single level of function call check error codes. One way I use this is to return Apache return codes:

```
# paranoid security check
die My::Exception->RetCode(code => 204);
```

Returns a 204 error code (`HTTP_NO_CONTENT`), which is caught at my top level exception handler:

```
if ($@->isa('My::Exception::RetCode')) {
    return $@->{code};
}
```

That last return statement is in my `handler()` method, so that's the return code that Apache actually sends. I have other exception handlers in place for sending Basic Authentication headers and Redirect headers out. I also have a generic `My::Exception::OK` class, which gives me a way to back out completely from where I am, but register that as an OK thing to do.

Why do I go to these extents? After all, code like slashcode (the code behind http://slashdot.org) doesn't need this sort of thing, so why should my web site? Well it's just a matter of scalability and programmer style really. There's lots of literature out there about exception handling, so I suggest doing some research.

## 5.11.6  Conclusions

Here I've demonstrated a simple and scalable (and useful) exception handling mechanism, that fits perfectly with your current code, and provides the programmer with excellent means to determine what has happened in his code. Some users might be worried about the overhead of such code. However in use I've found accessing the database to be a much more significant overhead, and this is used in some code delivering to thousands of users.

For similar exception handling techniques, see the section ''Other Implementations''.

## 5.11.7  The My::Exception class in its entirety

```
package My::Exception

use vars qw/@ISA @EXPORT $AUTOLOAD/;
use Exporter;
@ISA = 'Exporter';
@EXPORT = qw/die/;

sub import {
    my $pkg = shift;
    $pkg->export('CORE::GLOBAL', 'die');
    Exporter::import($pkg,@_);
}

sub die {
    if (!ref($_[0])) {
        CORE::die My::Exception->UnCaught(text => join('', @_));
    }
    CORE::die $_[0];
}

{
    package My::Exception::UnCaught;
    use overload '""' => \&str;

    sub str {
        shift->{text};
    }
}

sub AUTOLOAD {
    no strict 'refs', 'subs';
    if ($AUTOLOAD =~ /.*::([A-Z]\w+)$/) {
        my $exception = $1;
        *{$AUTOLOAD} =
            sub {
                shift;
                my ($package, $filename, $line) = caller;
                push @_, caller => {
                                    package => $package,
                                    filename => $filename,
                                    line => $line,
                                       };
                bless { @_ }, "My::Exception::$exception";
            };
        goto &{$AUTOLOAD};
    }
    else {
        die "No such exception class: $AUTOLOAD\n";
    }
}

1;
```

## *5.11.8  Other Implementations*

Some users might find it very useful to have a more C++/Java like interface of try/catch functions. These are available in several forms that all work in slightly different ways. See the documentation for each module for details:

- **Error.pm**

  Graham Barr's excellent OO styled ''try, throw, catch'' module (from CPAN).

- **Exception.pm and StackTrace.pm**

  by Autarch (from ftp://ftp.urth.org/pub/).

  `Exception` a bit cleaner than the AUTOLOAD method from the above examples as it can catch typos later on. Plus it lets you create actual class hierarchies for your exceptions, which could be nice if you want to create exception classes that do more stuff and then inherit from them.

- **Try.pm**

  Tony Olekshy's. Adds an unwind stack. Not on CPAN (yet?).

- **Exceptions.pm**

  Peter Seibel's `Exceptions` module is totally non-functional with modern Perl and has been superseded by Graham Barr's `Error` module.

  ;o)

# 6   Getting Help and Further Learning

## 6.1  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

## 6.2  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)

## 6.3  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 developers to share ideas, solve problems and discuss things related to mod_perl and the Apache::\* modules.** To subscribe to this list, send mail to modperl-subscribe@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/

## 6.4  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.

# 6.5  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://stason.org/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)

# 6.6  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/

# 6.7  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

# 6.8  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: