1 Perl Reference

1.1 Description

This document was born because some users are reluctant to learn Perl, prior to jumping into mod_perl. I will try to cover some of the most frequent pure Perl questions being asked at the list.

Before you decide to skip this chapter make sure you know all the information provided here. The rest of the Guide assumes that you have read this chapter and understood it.

1.2 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 at your fingertips is a great time saver. It always has the up-to-date information for the version of perl you're using.

Of course you can use online Perl documentation at the Web. The two major sites are http://perldoc.perl.org and http://theoryx5.uwinnipeg.ca/CPAN/perl/.

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 perl5.005_03 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.

The Perl FAQ (*perlfaq* manpage) is in several sections. To search through the sections for `open` you would execute:

```
% perldoc -q open
```

This will show you all the matching Question and Answer sections, still in POD format.

To read the *perldoc* manpage you would execute:

```
% perldoc perldoc
```

1.3 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

use strict;

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. Subroutine correct() passes the value to print, but in subroutine 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 strict;
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 subroutine, incorrect().

```
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. (See Finding the Line Which Triggered the Error or Warning for more information).

Getting the trace helps a lot.

1.4 Variables Globally, Lexically Scoped And Fully Qualified

META: this material is new and requires polishing so read with care.

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

1.4.1 Symbols, Symbol Tables and Packages; Typeglobs

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

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

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

You are always working in one package or another.

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

```
package mypackagename;
```

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

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

Most of the time, only one part of a typeglob is used (yes, it's a bit wasteful). You will by now know that you distinguish between them by using what the authors of the Camel book call a *funny character*. So if we have a scalar called '`line`' we would refer to it in code as `$line`, and if we had an array of the same name, that would be written, `@line`. Both would point to the same typeglob (which would be called `*line`), but because of the *funny character* (also known as *decoration*) perl won't confuse the two. Of course we might confuse ourselves, so some programmers don't ever use the same name for more than one type of variable.

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

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

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

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

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

process.

1.4.1.1 Lexical Variables and Symbols

The symbols for lexical variables (i.e. those declared using the keyword `my`) are the only symbols which do *not* live in a symbol table. Because of this, they are not available from outside the block in which they are declared. There is no typeglob associated with a lexical variable and a lexical variable can refer only to a scalar, an array, a hash or a code reference. (Since perl-5.6 it can also refer to a file glob).

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

The phrase "lexical variable" is a bit of a misnomer, we are really talking about "lexical symbols". The data can be referenced by a global symbol too, and in such cases when the lexical symbol goes out of scope the data will still be accessible through the global symbol. This is perfectly legitimate and cannot be compared to the terrible mistake of taking a pointer to an automatic C variable and returning it from a function--when the pointer is dereferenced there will be a segmentation fault. (Note for C/C++ programmers: having a function return a pointer to an auto variable is a disaster in C or C++; the perl equivalent, returning a reference to a lexical variable created in a function is normal and useful.)

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

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

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

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

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

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

1.4.2 Additional reading references

For more information see: Using global variables and sharing them between modules/packages and an article by Mark-Jason Dominus about how Perl handles variables and namespaces, and the difference between `use vars()` and `my ()` - http://www.plover.com/~mjd/perl/FAQs/Namespaces.html .

1.5 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.

1.5.1 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 fool you, the print_power_of_2() subroutine should print the square of the number passed to it. 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 the number 6? Let's try again using 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 the results of the first execution, or it ignores the arguments in subsequent executions.

1.5.2 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 `$x` variable. On subsequent calls the inner subroutine's `$x` variable won't be updated, no matter what new values are given to `$x` in the outer subroutine. There are two copies of the `$x` variable, no longer a single one shared by the two routines.

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

Let's verify:

```
% ./anonymous.pl

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

So we can see that the problem is solved.

1.6 Understanding Closures -- the Easy Way

In Perl, a closure is just a subroutine that refers to one or more lexical variables declared outside the subroutine itself and must therefore create a distinct clone of the environment on the way out.

And both named subroutines and anonymous subroutines can be closures.

Here's how to tell if a subroutine is a closure or not:

```
for (1..5) {
  push @a, sub { "hi there" };
}
for (1..5) {
  {
    my $b;
    push @b, sub { $b."hi there" };
  }
}
print "anon normal:\n", join "\t\n",@a,"\n";
print "anon closure:\n",join "\t\n",@b,"\n";
```

which generates:

```
anon normal:
CODE(0x80568e4)
CODE(0x80568e4)
CODE(0x80568e4)
CODE(0x80568e4)
CODE(0x80568e4)

anon closure:
CODE(0x804b4c0)
```

```
CODE(0x8056b54)
CODE(0x8056bb4)
CODE(0x80594d8)
CODE(0x8059538)
```

Note how each code reference from the non-closure is identical, but the closure form must generate distinct coderefs to point at the distinct instances of the closure.

And now the same with named subroutines:

```
for (1..5) {
  sub a { "hi there" };
  push @a, \&a;
}
for (1..5) {
  {
    my $b;
    sub b { $b."hi there" };
    push @b, \&b;
  }
}
print "normal:\n", join "\t\n",@a,"\n";
print "closure:\n",join "\t\n",@b,"\n";
```

which generates:

```
anon normal:
CODE(0x80568c0)
CODE(0x80568c0)
CODE(0x80568c0)
CODE(0x80568c0)
CODE(0x80568c0)

anon closure:
CODE(0x8056998)
CODE(0x8056998)
CODE(0x8056998)
CODE(0x8056998)
CODE(0x8056998)
```

We can see that both versions has generated the same code reference. For the subroutine *a* it's easy, since it doesn't include any lexical variables defined outside it in the same lexical scope.

As for the subroutine *b*, it's indeed a closure, but Perl won't recompile it since it's a named subroutine (see the *perlsub* manpage). It's something that we don't want to happen in our code unless we want it for this special effect, similar to *static* variables in C.

This is the underpinnings of that famous *"won't stay shared"* message. A *my* variable in a named subroutine context is generating identical code references and therefore it ignores any future changes to the lexical variables outside of it.

1.6.1 Mike Guy’s Explanation of the Inner Subroutine Behavior

```
From: mjtg@cus.cam.ac.uk (M.J.T. Guy)
Newsgroups: comp.lang.perl.misc
Subject: Re: Lexical scope and embedded subroutines.
Date: 6 Jan 1998 18:22:39 GMT
Message-ID: <68tspf$9f0$1@lyra.csx.cam.ac.uk>

In article <68sc4k$3p2$1@brokaw.wa.com>, Aaron Harsh <ajh@rtk.com>
wrote:

> Before I read this thread (and perlsub to get the details) I would
> have assumed the original code was fine.
>
> This behavior brings up the following questions:
>  o Is Perl’s behavior some sort of speed optimization?

No, but see below.

>  o Did the Perl gods just decide that scheme-like behavior was less
> important than the pseduo-static variables described in perlsub?

This subject has been kicked about at some length on perl5-porters.
The current behaviour was chosen as the best of a bad job.  In the
context of Perl, it’s not obvious what "scheme-like behavior" means.
So it isn’t an option.  See below for details.

>  o Does anyone else find Perl’s behavior counter-intuitive?

*Everyone* finds it counterintuitive.  The fact that it only generates
a warning rather than a hard error is part of the Perl Gods policy of
hurling thunderbolts at those so irreverent as not to use -w.

>  o Did programming in scheme destroy my ability to judge a decent
>    language
> feature?

You’re still interested in Perl, so it can’t have rotted your brain
completely.

>  o Have I misremembered how scheme handles these situations?

Probably not.

>  o Do Perl programmers really care how much Perl acts like scheme?

Some do.

>  o Should I have stopped this message two or three questions ago?

Yes.

The problem to be solved can be stated as

  "When a subroutine refers to a variable which is instantiated more
  than once (i.e. the variable is declared in a for loop, or in a
```

```
  subroutine), which instance of that variable should be used?"

The basic problem is that Perl isn't Scheme (or Pascal or any of the
other comparators that have been used).

In almost all lexically scoped languages (i.e. those in the Algol60
tradition), named subroutines are also lexically scoped.  So the scope
of the subroutine is necessarily contained in the scope of any
external variable referred to inside the subroutine.  So there's an
obvious answer to the "which instance?" problem.

But in Perl, named subroutines are globally scoped.  (But in some
future Perl, you'll be able to write

  my sub lex { ... }

to get lexical scoping.)  So the solution adopted by other languages
can't be used.

The next suggestion most people come up with is "Why not use the most
recently instantiated variable?".  This Does The Right Thing in many
cases, but fails when recursion or other complications are involved.

Consider:

  sub outer {

      inner();
      outer();
      my $trouble;
      inner();
      sub inner { $trouble };
      outer();
      inner();
  }

Which instance of $trouble is to be used for each call of inner()?
And why?

The consensus was that an incomplete solution was unacceptable, so the
simple rule "Use the first instance" was adopted instead.

And it is more efficient than possible alternative rules.  But that's
not why it was done.

Mike Guy
```

1.7 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 since compilation has already taken place.

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's memory, how do you call it again? If you could get all developers to code their scripts so each has a subroutine called run() that will actually execute the code in the script then we've solved half the problem.

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 a chance that there will be more than one script with the same name but residing in different directories, then in order to prevent namespace collisions the directory has to be a part of the package name too. And don't forget that the 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 the daemon is about to execute for the first time, the script 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 except the `%` 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 wanted. The only outstanding problem is how to pass the path to the script to the daemon. This detail is left as an exercise for the reader.

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.

1.7.1 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 have been many suggested in the past, and we discuss some of them here.

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

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

use strict;
use warnings;

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 `$counter` variable to 0, every time it is executed and then calls the inner subroutine increment_counter() twice. Sub increment_counter() prints `$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 !
```

Apparently, the `$counter` variable is not reinitialized on each execution of run(), it retains its value from the previous execution, and increment_counter() increments that. Actually that is not quite what happens. On each execution of run() a new `$counter` variable is initialized to zero but increment_counter() remains bound to the `$counter` variable from the first call to run().

The simplest of the work-rounds is to use package-scoped variables. These can be declared using `our` or, on older versions of Perl, the `vars` pragma. Note that whereas using `my` declaration also implicitly initializes variables to undefined the `our` declaration does not, and so you will probably need to add explicit initialisation for variables that lacked it.

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

use strict;
use warnings;

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

sub run {

  our $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.

In the above example we know `$counter` is just a simple small scalar. In the general case variables could reference external resource handles or large data structures. In that situation the fact that the variable would not be released immediately when run() completes could be a problem. To avoid this you can put `local` in front of the `our` declaration of all variables other than simple scalars. This has the effect of restoring the variable to its previous value (usually undefined) upon exit from the current scope. As a side-effect `local` also initializes the variables to `undef`. So, if you recall that thing I said about adding

explicit initialization when you replace `my` by `our`, well, you can forget it again if you replace `my` with `local our`.

Be warned that `local` will not release circular data structures. If the original CGI script relied upon process termination to clean up after it then it will leak memory as a registry script.

A varient of the package variable approach is not to declare your variables, but instead to use explicit package qualifiers. This has the advantage on old versions of Perl that there is no need to load the `vars` module, but it adds a significant typing overhead. Another downside is that you become dependant on the "used only once" warning to detect typos in variable names. The explicit package name approach is not really suitable for registry scripts because it pollutes the `main::` namespace rather than staying properly within the namespace that has been allocated. Finally, note that the overhead of loading the `vars` module only has to be paid once per Perl interpreter.

```
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 your code is. 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
```

```
use strict;
use warnings;

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;

    $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

use strict;
use warnings;

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;

    $$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 as would be the case of calling the function with a literal, e.g. *increment_counter(5)*).

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

use strict;
use warnings;

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
```

The approach given above should be properly documented of course.

Here is a solution that avoids the problem entirely by splitting the code into two files; the first is really just a wrapper and loader, the second file contains the heart of the code. This second file must go into a directory in your `@INC`. Some people like to put the library in the same directory as the script but this assumes that the current working directory will be equal to the directory where the script is located and also that `@INC` will contain `'.'`, neither of which are assumptions you should expect to hold in all cases.

Note that the name chosen for the library must be unique throughout the entire server and indeed every server on which you many ever install the script. This solution is probably more trouble than it is worth - it is only oncluded because it was mentioned in previous versions of this guide.

```
multirun6.pl
------------
#!/usr/bin/perl

use strict;
```

```
use warnings;

require 'multirun6-lib.pl';

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

Separate file:

```
multirun6-lib.pl
----------------
use strict;
use warnings;

my $counter;

sub run {
  $counter = 0;

  increment_counter();
  increment_counter();
}

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

1 ;
```

An alternative verion of the above, that mitigates some of the disadvantages, is to use a Perl5-style Exporter module rather than a Perl4-style library. The global uniqueness requirement still applies to the module name, but at least this is a problem Perl programmers should already be familiar with when creating modules.

```
multirun7.pl
------------
#!/usr/bin/perl

use strict;
use warnings;
use My::Multirun7;

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

Separate file:

```
My/Multirun7.pm
---------------
package My::Multirun7;
use strict;
use warnings;
use base qw( Exporter );
our @EXPORT = qw( run );

my $counter;

sub run {
  $counter = 0;

  increment_counter();
  increment_counter();
}

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

1 ;
```

Now you have at least five workarounds to choose from (not counting numbers 2 and 6).

For more information please refer to perlref and perlsub manpages.

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

1.8.1 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.

1.8.2 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() statements. Before attempting to load a file or a module with use() or require(), Perl checks whether it's already in the `%INC` hash. If it's there, the loading and therefore the compilation are not performed at all. Otherwise the file is loaded into memory and an attempt is made to compile it. do() does unconditional loading--no lookup in the `%INC` hash is made.

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

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

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

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

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

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

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

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

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

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

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

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

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

test.pm => test.pm
```

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

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

test.pm => test.pm
```

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

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

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

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

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

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

The code:

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

can be replaced with the more elegant:

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

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

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

There is a module called `FindBin` which solves this problem in the plain Perl world, but unfortunately up untill perl 5.9.1 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. Perl 5.9.1 provides a new function `FindBin::again` which will do the right thing. Also the CPAN module `FindBin::Real` provides a working alternative working under mod_perl.

For the sake of completeness, I'll present the `FindBin` 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 `$FindBin::Bin` is equal to */tmp*. If we move the script somewhere else... e.g. */tmp/new_dir* in the code above `$FindBin::Bin` equals */tmp/new_dir*.

```
% /tmp/load.pl

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

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

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

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

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

1.8.3 Modules, Libraries and Program Files

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

- **Libraries**

  These are files which contain Perl subroutines and other code.

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

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

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

  Examples:

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

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

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

- **Modules**

  A file which contains perl subroutines and other code.

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

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

  Its last statement returns true.

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

  Example:

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

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

- **Program Files**

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

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

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

  Its last statement may return anything or nothing.

1.8.4 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() exception-handling block, as in this example:

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

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

When we execute the program:

```
% ./require.pl

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

Hello
```

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

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

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

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

% ./require1.pl

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

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

For more information refer to the perlfunc manpage.

1.8.5 use()

use(), just like require(), loads and compiles files containing Perl code, but it works with modules only and is executed at compile time.

The only way to pass a module to load is by its module name and not its filename. If the module is located in *MyCode.pm*, the correct way to use() it is:

```
use MyCode
```

and not:

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

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

use() is exactly equivalent to:

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

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

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

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

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

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

The second pair exports only the tags passed as arguments.

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

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

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

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

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

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

1.9 Using Global Variables and Sharing Them Between Modules/Packages

It helps when you code your application in a structured way, using the perl packages, but as you probably know once you start using packages it's much harder to share the variables between the various packagings. A configuration package comes to mind as a good example of the package that will want its variables to be accessible from the other modules.

Of course using the Object Oriented (OO) programming is the best way to provide an access to variables through the access methods. But if you are not yet ready for OO techniques you can still benefit from using the techniques we are going to talk about.

1.9.1 Making Variables Global

When you first wrote `$x` in your code you created a (package) global variable. It is visible everywhere in your program, although if used in a package other than the package in which it was declared (`main::` by default), it must be referred to with its fully qualified name, unless you have imported this variable with import(). This will work only if you do not use `strict` pragma; but you *have* to use this pragma if you want to run your scripts under mod_perl. Read The strict pragma to find out why.

1.9.2 Making Variables Global With strict Pragma On

First you use :

```
use strict;
```

Then you use:

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

This declares the named variables as package globals in the current package. They may be referred to within the same file and package with their unqualified names; and in different files/packages with their fully qualified names.

With perl5.6 you can use the `our` operator instead:

```
our($scalar, %hash, @array);
```

If you want to share package global variables between packages, here is what you can do.

1.9.3 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 = CGI->new;

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 add the same `Exporter` code that we used in `My::HTML`, into `My::Doc`, so that it also exports `$q`.

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;
```

1.9.4 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 increases 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  => [qw(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_var};
print $c{array_var}[0];
print $c{hash_var}{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; # ERROR!
```

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.

1.9.5 Using Non-Hardcoded Configuration Module Names

You have just seen how to use a configuration module for configuration centralization and an easy access to the information stored in this module. However, there is somewhat of a chicken-and-egg problem--how to let your other modules know the name of this file? Hardcoding the name is brittle--if you have only a single project it should be fine, but if you have more projects which use different configurations and you will want to reuse their code you will have to find all instances of the hardcoded name and replace it.

Another solution could be to have the same name for a configuration module, like `My::Config` but putting a different copy of it into different locations. But this won't work under mod_perl because of the namespace collision. You cannot load different modules which uses the same name, only the first one will be loaded.

Luckily, there is another solution which allows us to stay flexible. `PerlSetVar` comes to rescue. Just like with environment variables, you can set server's global Perl variables which can be retrieved from any module and script. Those statements are placed into the *httpd.conf* file. For example

```
PerlSetVar FooBaseDir       /home/httpd/foo
PerlSetVar FooConfigModule  Foo::Config
```

Now we require() the file where the above configuration will be used.

```
PerlRequire /home/httpd/perl/startup.pl
```

In the *startup.pl* we might have the following code:

```
# retrieve the configuration module path
use Apache;
my $s             = Apache->server;
my $base_dir      = $s->dir_config('FooBaseDir')      || '';
my $config_module = $s->dir_config('FooConfigModule') || '';
die "FooBaseDir and FooConfigModule aren't set in httpd.conf"
    unless $base_dir and $config_module;

# build the real path to the config module
my $path = "$base_dir/$config_module";
$path =~ s|::|/|;
$path .= ".pm";
```

```
# we have something like "/home/httpd/foo/Foo/Config.pm"

# now we can pull in the configuration module
require $path;
```

Now we know the module name and it's loaded, so for example if we need to use some variables stored in this module to open a database connection, we will do:

```
Apache::DBI->connect_on_init
("DBI:mysql:${$config_module.'::DB_NAME'}::${$config_module.'::SERVER'}",
 ${$config_module.'::USER'},
 ${$config_module.'::USER_PASSWD'},
 {
  PrintError => 1, # warn() on errors
  RaiseError => 0, # don't die on error
  AutoCommit => 1, # commit executes immediately
 }
);
```

Where variable like:

```
${$config_module.'::USER'}
```

In our example are really:

```
$Foo::Config::USER
```

If you want to access these variable from within your code at the run time, instead accessing to the server object `$c`, use the request object `$r`:

```
my $r = shift;
my $base_dir      = $r->dir_config('FooBaseDir')      || '';
my $config_module = $r->dir_config('FooConfigModule') || '';
```

1.10 The Scope of the Special Perl Variables

Special Perl variables like `$|` (buffering), `$^T` (script's start time), `$^W` (warnings mode), `$/` (input record separator), `$\` (output record separator) and many more are all true global variables; they do not belong to any particular package (not even `main::`) and are universally available. This means that if you change them, you change them anywhere across the entire program; furthermore you cannot scope them with my (). However you can local()ise them which means that any changes you apply will only last until the end of the enclosing scope. In the mod_perl situation where 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. Therefore localizing these variables is highly recommended, I'd say mandatory.

We will demonstrate the case on the input record separator variable. If you undefine this variable, the diamond operator (readline) 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; # BAD!
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 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.

Note that if you call a subroutine after you've set a global variable but within the enclosing block, the global variable will be visible with its new value inside the subroutine.

1.11 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 regex 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 the `grep()` or `map()` operators.

In long-lived mod_perl scripts, however, the variable may change with each 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; };
}
```

means that we recompile the regex for every element in the list even though the regex doesn't change.

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 regex to which you apply this technique. One common usage where a compiled regex 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 Regular Expressions*".

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

And of course you can use the qr() operator which makes the code even more efficient:

```
my $pat = '^foo$';
my $re  = qr($pat);
foreach( @list ) {
    print if /$re/;
}
```

The qr() operator compiles the pattern for each request and then use the compiled version in the actual match.

1.12 Exception Handling for mod_perl

Here are some guidelines for clean(er) exception handling in mod_perl, although the technique presented can be applied 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 the 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). In short summary, $SIG{__DIE__} is a little bit too global, and catches exceptions even when you want to catch them yourself, using an `eval{}` block.

1.12.1 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. For the hardcore C programmers among you, it uses the `setjmp/longjmp` POSIX routines internally, just like C++ exceptions.

When in an eval block, if the code being executed die()'s for any reason, an exception is thrown. This exception can be caught by examining the $@ variable immediately after the eval block; if $@ is true then an exception occurred and $@ contains the exception in the form of a string. 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.

Perl's exception handling is similar to that of other languages, though it may not seem so at first sight:

```
Perl                            Other language
-------------------------------  ------------------------------------
eval {                          try {
  # execute here                    // execute here
  # raise our own exception:        // raise our own exception:
  die "Oops" if /error/;            if(error==1){throw Exception.Oops;}
  # execute more                    // execute more
} ;                             }
if($@) {                        catch {
  # handle exceptions               switch( Exception.id ) {
  if( $@ =~ /Fail/ ) {                Fail : fprintf( stderr, "Failed\n" ) ;
      print "Failed\n" ;                     break ;
  }
  elsif( $@ =~ /Oops/ ) {             Oops : throw Exception ;
      # Pass it up the chain
      die if $@ =~ /Oops/;
```

```
  }
  else {                              default :
      # handle all other             }
      # exceptions here            }
  }                                 // If we got here all is OK or handled
}
else { # optional
  # all is well
}
# all is well or has been handled
```

1.12.2 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 set up 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 `$SIG{__DIE__}` handler is hidden away deep in some other module that you didn't know about. There are work arounds however. One is to localize `$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!

`$^S` isn't totally reliable in certain Perl versions. e.g. 5.005_03 and 5.6.1 both do the wrong thing with it in certain situations. Instead, you use can use the caller() function to figure out if we are called in the eval() context:

```
$SIG{__DIE__} = sub {
    my $in_eval = 0;
    for(my $stack = 1;  my $sub = (CORE::caller($stack))[3];  $stack++) {
        $in_eval = 1 if $sub =~ /^\(eval\)/;
    }
    my_die_handler(@_) unless $in_eval;
};
```

The other problem with `$SIG{__DIE__}` also relates to its global nature. Because you might have more than one application running under mod_perl, you can't be sure which has set a `$SIG{__DIE__}` handler when and for what. This can become extremely confusing when you start scaling up from a set of simple registry scripts that might rely on CGI::Carp for global exception handling (which uses `$SIG{__DIE__}` to trap exceptions) to having many applications installed with a variety of exception handling mechanisms in place.

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 do just that.

1.12.3 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). The other problem is that you have no way of automatically detecting where the exception occurred using `eval{}` construct. In a `$SIG{__DIE__}` block you always have the use of the caller() function to detect where the error occurred. But we can fix that...

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, potentially providing almost all of the power Java exceptions.

[As a footnote here, the only thing that is really missing here from Java exceptions is a guaranteed Finally clause, although its possible to get about 98.62% of the way towards providing that using `eval{}`.]

1.12.3.1 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. The exception to this rule seems to be Apache::Registry scripts, although the techniques described here can be easily adapted.

In order for you to be able to use exception handling to its best advantage 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 modularize your script into discrete function calls, one of which should be a dispatch routine:

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

eval {
   dispatch();
};
if ($@) {
   # handle exception
}

sub dispatch {
    ...
}
```

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($r);
    };
    if ($@) {
       # handle exception
    }
}

sub dispatch {
    my $r = 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.

1.12.3.2 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 {
    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;
```

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).

1.12.4 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.

1.12.4.1 Using $SIG{__DIE__}

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

```
$SIG{__DIE__} = sub {
    if(!ref($_[0])) {
        $err = My::Exception->UnCaught(text => join('', @_));
    }
    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. Even though $SIG{__DIE__} is a global handler, because we are simply re-throwing the exception we can let other applications outside of our control simply catch the exception and not worry about it.

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. Note that there is a bug in Perl 5.6 which may affect people here: Stringification does not occur when an object is operated on by a regular expression (via the =~ operator). A work around is to explicitly stringify using qq double quotes, however that doesn't help the poor soul who is using other applications. This bug has been fixed in later versions of Perl.

1.12.4.2 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 die (@); # prototype to match CORE::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]; # only use first element because its an object
}
```

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. If we don't want to overload die() everywhere this can still be an extremely useful technique. By just using Exporter's default import() method we can export our new die() method into any package of our choosing. This allows us to short-cut the long calling convention and simply die() with a string, and let the system handle the actual construction into an object for us.

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).

1.12.5 A Single UnCaught Exception Class

Until the Perl core gets its own base exception class (which will likely happen for Perl 6, but not sooner), it is vitally important that you decide upon a single base exception class for all of the applications that you install on your server, and a single exception handling technique. The problem comes when you have multiple applications all doing exception handling and all expecting a certain type of "UnCaught" exception class. Witness the following application:

```
package Foo;

eval {
   # do something
}
if ($@) {
   if ($@->isa('Foo::Exception::Bar')) {
      # handle "Bar" exception
   }
   elsif ($@->isa('Foo::Exception::UnCaught')) {
      # handle uncaught exceptions
   }
}
```

All will work well until someone installs application "TrapMe" on the same machine, which installs its own UnCaught exception handler, overloading CORE::GLOBAL::die or installing a $SIG{__DIE__} handler. This is actually a case where using $SIG{__DIE__} might actually be preferable, because you can change your handler() routine to look like this:

```
sub handler {
    my $r = shift;

    local $SIG{__DIE__};
    Foo::Exception->Init(); # sets $SIG{__DIE__}

    eval {
       dispatch($r);
    };
    if ($@) {
       # handle exception
    }
}

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

In this case the very nature of $SIG{__DIE__} being a lexical variable has helped us, something we couldn't fix with overloading CORE::GLOBAL::die. However there is still a gotcha. If someone has overloaded die() in one of the applications installed on your mod_perl machine, you get the same problems still. So in short: Watch out, and check the source code of anything you install to make sure it follows your exception handling technique, or just uses die() with strings.

1.12.6 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 call stack, 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 a lot of literature out there about exception handling, so I suggest doing some research.

1.12.7 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 an 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".

1.12.8 The My::Exception class in its entirety

```
package My::Exception;

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

sub die (@);

sub import {
    my $pkg = shift;
    # allow "use My::Exception 'die';" to mean import locally only
    $pkg->export('CORE::GLOBAL', 'die') unless @_;
    Exporter::import($pkg,@_);
}

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

```
}

{
    package My::Exception::UnCaught;
    use overload '""' => sub { 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 {
        CORE::die "No such exception class: $AUTOLOAD\n";
    }
}

1;
```

1.12.9 Other Implementations

Some users might find it very useful to have the 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). This should be considered your best option for structured exception handling because it is well known and well supported and used by a lot of other applications.

- **Exception::Class and Devel::StackTrace**

  by Dave Rolsky both available from CPAN of course.

  `Exception::Class` is a bit cleaner than the `AUTOLOAD` method from above as it can catch typos in exception class names, whereas the method above will automatically create a new class for you. In addition, it lets you create actual class hierarchies for your exceptions, which can be useful if you want to create exception classes that provide extra methods or data. For example, an exception class for database errors could provide a method for returning the SQL and bound parameters in use at the

time of the error.

- **Try.pm**

  Tony Olekshy's. Adds an unwind stack and some other interesting features. Not on the CPAN. Available at http://www.avrasoft.com/perl6/try6-ref5.txt

1.13 Customized __DIE__ handler

As we saw in the previous sections it's a bad idea to do:

```
require Carp;
$SIG{__DIE__} = \&Carp::confess;
```

since it breaks the error propogations within eval {} blocks,. But starting from perl 5.6.x you can use another solution to trace errors. For example you get an error:

```
"exit" is not exported by the GLOB(0x88414cc) module at (eval 397) line 1
```

and you have no clue where it comes from, you can override the exit() function and plug the tracer inside:

```
require Carp;
use subs qw(CORE::GLOBAL::die);
*CORE::GLOBAL::die = sub {
    if ($_[0] =~ /"exit" is not exported/){
        local *CORE::GLOBAL::die = sub { CORE::die(@_) };
        Carp::confess(@_); # Carp uses die() internally!
    } else {
        CORE::die(@_); # could write &CORE::die to forward @_
    }
};
```

Now we can test that it works properly without breaking the eval {} blocks error propogation:

```
eval { foo(); }; warn $@ if $@;
print "\n";
eval { poo(); }; warn $@ if $@;

sub foo{ bar(); }
sub bar{ die qq{"exit" is not exported}}

sub poo{ tar(); }
sub tar{ die "normal exit"}
```

prints:

```
$ perl -w test
Subroutine die redefined at test line 5.
"exit" is not exported at test line 6
      main::__ANON__('"exit" is not exported') called at test line 17
      main::bar() called at test line 16
      main::foo() called at test line 12
      eval {...} called at test line 12

normal exit at test line 5.
```

the 'local' in:

```
local *CORE::GLOBAL::die = sub { CORE::die(@_) };
```

is important, so you won't lose the overloaded `CORE::GLOBAL::die`.

1.14 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

1.15 Authors

- Stas Bekman [http://stason.org/]
- Matt Sergeant <matt (at) sergeant.org>

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents: