Introduction to mod_perl (part V): More Perl Basics

By Stas Bekman

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

Table of Contents

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. This can be easily fixed by helping compiler with `#line` directive. If you put the following at the beginning of the line in your script:

```
#line 125
```

it will tell the compiler that the **next** line is number 125 for reporting needs. Of course the rest of the lines would be adapted as well.

Getting the trace helps a lot.

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.

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.

The Diagnosis

Let's follow the guidelines and use the `-w` flag:

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

Under Perl version 5.6.0+ we use the `warnings` pragma:

```
#!/usr/bin/perl
use warnings;
```

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.

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
```

So we can see that the problem is solved.

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.

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

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 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 -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;

    $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;

    $$r_counter++;
    print "Counter is equal to $$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 -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
```

The approach given above is generally not recommended because most Perl programmers will not expect `$counter` to be changed by the function; the example where we used `\$counter`, i.e. pass-by-reference would be preferred.

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.

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

use strict;
require 'multirun6-lib.pl' ;

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

Separate file:

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

```
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 six workarounds to choose from.

For more information please refer to perlref and perlsub manpages.

perldoc’s Rarely Known But Very Useful Options

It’s a known fact, that one cannot become a Perl hacker and especially mod_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. I prefer http://theoryx5.uwinnipeg.ca/CPAN/perl/ to the official URL: http://www.perl.com/pub/v/documentation is very slow :(. 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
```

References

- Online documentation: http://theoryx5.uwinnipeg.ca/CPAN/perl/
  http://www.perl.com/pub/v/documentation/

- The book ‘‘*Programming Perl*’’ 3rd edition by L.Wall, T. Christiansen and J.Orwant (also known as the ‘‘*Camel*’’ book, named after the camel picture on the cover of the book). You want to refer to Chapter 8 that talks about nested subroutines among other things.

- The *perlref* and *perlsub* man pages.