Perl: Part 6

Thinking in Line Noise

This month we introduce some of the more powerful idioms and features of Perl and show why it's still one of the hackers languages of choice.

BY DEAN WILSON AND FRANK BOOTH

While the aspects of Perl that have been covered in this series so far are enough to start you upon your way to becoming yet another Perl hacker they've been the basics of the language and offer nothing other languages do not, albeit in a lot less lines of code.

Nested Data structures

Perl facilitates complex datastructures in several ways, by far the most readily understood are the "hash of hashes" and "list of lists"; these are nested data-structures. It is also possible to have "lists of hashes" and "hashes of lists".

What we mean when we say a "list of hashes" is that the data structure is a list that contains hashes as its elements. One example might be a list of people and for each person a list of their top five favourite shell commands:

```
Terry: rm -rf*, chmod 777, ➥
kill -9, ln -s, reboot
Billy: vim, df -h, ls -lah, ➥
ps -eaf, mutt
```

We could write that very quickly in Perl, using nested data structures. In this instance a hash of lists appears to be the most sensible as the user's names will be unique identifiers and the top five commands have no other significance but the order in which they occur. Using a hash structure for the names and a list for the commands for each user we can access the information in an intuitive fashion and pull out details such as the favourite command used by Billy. We'll show ways of obtaining this data later. First we'd better store the data. One way of writing this in Perl would be:

```
my %commands = (
 Terry => [ 'rm -rf*', 'chmod ➥
777', 'kill -9', 'ln -s', ➥
'reboot' ],
 Billy => [ 'vim', 'df -h', ➥
'ls -lah', 'ps -eaf', 'mutt' ]
);
```

Within the code sample above, the only unfamiliar symbol should be the square brackets, these are used to denote anonymous lists. Anonymous lists are arrays without a name. A clue to this is the square brackets '[]', usually seen when accessing elements of an array:

```
print "$some_array[4]\n";
```

So it's not really counter intuitive that square brackets be used elsewhere for arrays. Using this philosophy can you guess what sigils we use to create an anonymous hash? We use '{}' curly braces, as we use curly braces to retrieve a value from a hash:

```
print "$some_hash{four}\n";
```

Or if you prefer:

```
print $some_hash{'four'} . "\n";
```

Returning to our list of users favourite commands again, when we need to reference the data inside our nested structures, we need a means of specifying the element inside the parent data structure we want. We can access '%commands', in the normal fashion:

```
#returns a string akin to ➥
'ARRAY(0x1ab54d0)'
print "$commands{'Billy'}\n";
```

But this returns a string that looks like "ARRAY(0x1ab54d0)" which actually tells us a lot but not what we wanted. The uppercase 'ARRAY' tells us that the returned value is of an array reference type and the characters with in the parenthesis tell us where Perl stores the reference. To access the data from the list within the hash, we use the arrow operator '->', this enables us to access the list within the hash:

```
$commands{Billy}->[0];
```

This will return the first item from Billy's list of shell commands. For the purposes of this exercise, we'll say that the list places favourite items first.

We can now make a hash of lists using anonymous lists, we can make an array of hashes too. you may by now be asking yourself if there any other things that can be made anonymously.

Anonymous Subroutines

It's the Perl way, if you've got anonymous hashes and anonymous lists, then what about the functions and the scalars? Perl provides them too. An anonymous function seems like an odd thing to have, until you get sufficiently lazy, then you find yourself using them.

```
my %func = (
  stdout => sub { print @_ },
  log => sub { print LOG @_ },
  stderr => sub ➥
{ print STDERR @_ },
  not_stderr => sub {
    print @_;
    print LOG @_;
  },
  not_stdout => sub {
  print STDERR @_;
  print LOG @_;
  },
  all => sub {
    print STDERR @_;
    print LOG @_;
    print @_;
  }
);
# Print to all bar stdout:
&$func{not_stdout}( 'hello', ➥
```

```
'world' );
# Print lots of times to each:
&$func{$_} for keys %func;
```

The last command seems completely nonsensical as it's a hash data structure, there is no telling what order the elements will emerge when using the keys command, which could cause problems. If you require order, use an array. The following example will produce a set of error levels, increasing in urgency.

```
my @warn = (
  sub { print STDERR @_ },
  sub { print LOG @_ },
  sub { print @_ },
  sub { die "I can't function under these ➥
conditions: ", @_,"\n" }
);
 sub notify {
  my $error_level = shift || -1;
  &$func{$_}(@_) for ( 0..$error_level );
  -1
}
```

This example will report messages back according to the error level. If the error level was 1, it would write to 'STDERR' and the log file. At level 3, it would write to the 'STDERR', 'LOG', 'STDOUT' and the finally stop the program with a final message. It does this by looping through the array of anonymous subroutines. There are a few things that happen implicitly, that we'll examine now:

```
my $error_level = shift || -1;
```

When variables are passed into a function they're passed in as an array (@_). The 'shift' operator removes the first item from an array, its default is '@_', so if no array is specified, it defaults to using @_. If the '@_' array is false (it has no contents) the value -1 is placed in the variable instead, why will become clear soon.

```
&$func{$_}(@_) for 0..$error_level;
```

This line uses Perl's 'for' looping construct to iterate over the first part of the line. In this instance it will repeat for every number from 0 to $error_level, the value for the current iteration will be put into the default variable '$_'. Since the 'for' loop occurs at the end of the line the loop condition doesn't need braces. It is worth noting that the range operator ('..') will not work backwards, it won't count from 10 to 1 using 10..1, it will merely skip the entire loop as having failed on the first attempt.

The first part of the line calls the function from the array '@warn', the element it references is the value of '$_', and the parameters the function is passed are the remainder of the parameters passed to the notify function. The ampersand '&' denotes that the thing in the hash is a function. It is necessary to explain that the contents are a

function, otherwise Perl would expect a normal scalar value and would interpret the function as such.

References

References are scalar variables used to point to anonymous data types and functions. In all the above examples we've relied on the containing data structure to ensure we look at the data we meant to or call the function we intended. We can just as easily use a scalar variable to do the same task.

```
my $array_reference = ➥
[ 1, 2, 3, 4, 5 ];
my $hash_reference = {
  beef => 'corned',
  cabbage => 'over-cooked'
};
```

We refer to the elements within the reference using the arrow operator '->':

```
$array_reference->[0];
@$array_reference[0];
$hash_reference->{beef};
%$hash_reference{beef};
```

We can refer to anonymous functions:

```
$func = sub { print "foo\n" };
&$func;
```

We can refer to scalar values too:

```
$func = \'3.14';
print $$func;
```

Here we've prefixed the variable we're applying with a data type constraint. Putting the wrong type in a data type constraint will result in the program concluding rather sooner than you'd hoped, if you don't know what type of data to expect try something like this:

```
sub handleref ($) {
  my $reference = ➥
shift or return;
  $_ = ref( $reference);
  /SCALAR/ and return ➥
$$reference;
  /ARRAY/ and return join ➥
(', ',@$reference );
  /HASH/ and return join ➥
(', ', keys %$reference );
  $_
}
```

DATA TYPES

Type	Action
SCALAR	Return the value.
HASH	Return a joined list of keys.
ARRAY	Return a joined list of values.
NOT A REF	Return the value itself.

This program uses the 'ref' function to determine the data type of a reference. 'ref' returns one of a number of possible values including the more common: SCALAR, ARRAY, HASH or ''. The last value indicates that the parameter sent was in fact not a reference at all.

What the code does is define the response taken when passed different data types: Here is a list of the input type and result output. 'ref' is extremely useful when using generic datastructures that can nest any type of data and have no defined limit of depth to which it is nested as it allows fully automatic determination of the references type. You may want to create a reference to an existing structure, to enable access from a function, or to link to a dynamically structured list. We use the '\' backslash operator to dereference a value:

```
# Makes a reference to a scalar
my $foo_ref = \$foo;
# Makes a reference to @foo ➥
called $foo_arrref
my $foo_arrref = \@foo;
# Make an array of references.
my @list_of_arrays = ➥
( \@foo, \@bar, \@baz );
# this can also be written:
@list_of_arrays = ➥
\( @foo, @bar, @baz );
```

Here be Dragons

Closures are one of the more complex features of Perl in that they build upon previous knowledge and require a grasp of a number of the language basics such as scope and pass by reference before they become readily comprehensible. However like most magic, you don't need to understand it to wield it.

A closure is a function that exploits both the lexical scope it is declared in and Perl's garbage collection algorithm to preserve a variable beyond its expected lifetime.

We've not yet discussed Perl's garbage disposal routine in any depth as it is unobtrusive and rarely falls to the programmer to know or care what it does and how it works. It tidies up after us and ensures that the memory no longer used in our programs is released.

The garbage collector in Perl works on a very simple (in theory) principle known as reference counting. Whenever a new variable is created it starts off with a reference count of 1 and each time a reference to that variable is taken the count increases by one.

Each time a reference to the variable falls out of scope the reference count decreases by one and when no more references point to it (IE the reference count is zero) the variable is 'reaped' by Perl's garbage collection and the memory it used is released automaticly, no explicit 'malloc' and 'free' for us! To clarify how closures work let's look at what we know. We know that a variable declared in the scope of a block only exists for that block…

```
{
   my $count;
   print "$count\n";
}
# this line fails compilation
# as $count is not visible
print "$count\n";
```

We also know that a function is global regardless of where it is defined:

```
{
  sub phrase {
    return ➥
' I can be called anywhere ';
  }
}
print phrase(); # this works.
```

So what happens when we mix the two?

```
{
   my $count = 0;
   sub set($){ $count = shift }
   sub incr(){ $count++ }
   sub getcnt() { $count }
}
set(5); # sets the count to 5
incr;   # adds 1 to count.
#this prints 6
print getcnt(), "\n";
```

We get a variable named '$count' that exists only for the functions 'set', 'incr' and 'getcnt' any other attempt to reference the variable will fail. This gives us a "tamed" global variable that has limited ways of being altered while also providing some data encapsulation; A global variable we can manage.

There are instances when global variables need to be used and there are instances when you can use a closure instead to make the code a little safer and avoid another global. If you think this looks a little like very primative Object Orientation (OO) then you may not be surprised to know that these principles will hold you in good stead when we get to Perl's OO facilities.

While the above is a useful application of a closure, it is not the most common use of closures. In the example below we use an anonymous subroutine to create a bespoke function. This is probably the most popular and often seen use of closures within Perl.

```
sub hello($) {
   my $message = shift;
   return sub ➥
{ print "Hello $message\n";
}
```

This is a customisable function. A closure can be created by calling the function like so:

```
my $std = hello('world');
my $song = hello('dolly');
my $phrase = hello('nurse!');
```

We can call all the separate closures using the ampersand symbol to signify its a function and the variable that holds the reference to the anonymous subroutine. So:

```
&$std #will print: Hello world.
&$song #will print: Hello dolly.
&$phrase # will print: ➥
Hello nurse!
```

These rather trite examples serve only to illustrate the basics of how closures work but hopefully they will whet your appetite for the advanced potential uses they provide once you have made it past the initial hurdle and understand how they work.

Data::Dumper

Once you've started to use more complex references you'll inevitably want to view the contents of a complex data structure.

While your first instinct may be to 'unroll' the structure with a number of loops, a better approach would be to use a module from the Perl core (it's installed by default) called 'Data::➥ Dumper'. We'll show uses of Data::➥ Dumper here without explaining all the details behind using modules as a gentle introduction.

A full explanation will be covered in a future column. 'Data::Dumper' is a module that is capable of serializing Perl data structures so they can be printed to screen or even written to a file while remaining valid Perl code.

The last point is an important one that warrants a deeper explanation, the stringified version of the data structure is still valid Perl code, this allows it to be used in an 'eval' to recreate the structures in the current application and even to be read in from a file and used as a simple persistence layer.

The example 'simple_dump.pl' below shows a rudimentary use of 'Data::➥ Dumper' to print a hash containing hash references. Although the example may look slightly contrived the principles can still be applied to larger code such as a function passing back a complex hash ref of configuration settings such as for example an 'ini' file style configuration.

```
#Example: simple_dump.pl
use Data::Dumper;
my (%config, $config_ref);
%config = (
   email => {
     workdir => ➥
'/home/dwilson/work',
     logdir  => '/var/log/➥
perlapps/examples/email'
  },
   news => {
     workdir => ➥
'/home/dwilson/work',
     logdir  => '/var/log/➥
perlapps/examples/news'
  }
);
$config_ref = \%config;
print Dumper($config_ref);
```

This example shows a simple use of Data::Dumper's procedural interface to print the representation to the console. The first line imports the 'Data::Dumper' module and allows any of its exported functions to be called. We then create both a hash and a scalar and immediately put some sample data in the hash. It's useful to note how the hash of hashes is built up manually as the 'Data::Dumper' representation is remarkable similar.

The row following should now be familiar as we take a reference to the hash. Finally we make use of Data::Dumper with the exported 'Dumper' function. If you run the code you'll see how closely the output resembles the original code.

The 'Data::Dumper' module itself can be used in either a procedural or object orientated (OO) fashion allowing it to fit inconspicuously in to the surrounding code as all good third party modules should. The example below uses the OO interfaces and requires only minimal changes:

```
#Example: simple_dump_oo.pl
#above here we would create ➥
the hash
my $dumper = Data::Dumper->➥
new([$config_ref]);
print $dumper->➥
Dump($config_ref);
```

We start the 'simple_dump_oo.pl' example with the same set up code used in the 'simple_dump.pl' example. The code changes begin in the last few lines as we create an instance of the Data::Dumper class and pass in the reference we would like to have it work on, notice the use of braces to force list context, Data::Dumper's constructor expects its first argument (A second optional argument is allowed) to be an array ref.

Once we have a variable holding the object we then call the 'Dump' method and get the same on screen information we did with the procedural version.

Now that the basic use of Data::➥ Dumper has been shown we move on to some useful options that can be configured to customize how *Data::Dumper* represents its output. These options are

set differently depending upon the way in which you are using the module, for the moment don't worry about their purpose but rather how they are set. For the procedural version:

```
$Data::Dumper::Useqq = 1;
$Data::Dumper::Varname = 'foo';
```

These configuration settings are global so it is prudent to limit the scope the changes affect by using them within a separate often anonymous block, this is best done using 'local':

```
{ #start anonymous block
  local $Data::Dumper::Useqq =
= 1;
  local $Data::Dumper::Varname =
= 'foo';
} # changes are lost when the
code reaches here.
```

The options are set using methods in the OO style of use and look like this:

```
$dumper->Useqq('1');
$dumper->Varname('foo');
```

When the settings are changed via methods they do not need require the jumping through hoops to limit the scope of the change as any change applies only to the one object:

```
my $dumper_cust = Data::Dumper
->new([$config_ref]);
$dumper_cust->Varname('foo');
print $dumper_cust->
Dump($config_ref);
my $dumper_raw = Data::Dumper->
new([$config_ref]);
print $dumper_raw->
Dump($config_ref);
```

When the second Data::Dumper instance ('$dumper_raw') prints its output it will use 'VAR' instead of 'foo'. Now we have covered setting the values it is useful to know that the methods also act as accessors and if you call one with no parameters it returns the current value:

```
my $prefix = $dumper_cust->
Varname();
#prints 'default prefix is VAR'
print "default prefix is
$prefix\n";
```

While the default settings are often enough you may occasionally need to tweak the settings to suit the use the module is put to. Two modified settings are *$Data::Dumper::Indent* and *$Data::Dumper::Useqq* or in OO parlance *$OBJ->Indent* and *$OBJ->Useqq*

The first of these two '$Data::Dumper::Indent', controls the general human readability of the output structure. From the minimum value of '0' which strips out all but the essential white space leaving the output as valid perl code but not easily human readable through to a maximum value of '3'. The default value is '2' and this causes the output to have newlines, nicely lines up entries in hashes and similar and sensible indentation.

While a value of '2' is often enough if you are dealing with a large number of with complex arrays then it is worth at least considering a value of '3' as its main benefit is to put out the array index along with the data allowing quick visual look-ups at the cost of doubling the output size. In practical terms it is often enough to leave the setting at its default value but if you are using Data::Dumper to serialize the structures to disk then you can get away with a lower level as it only needs to be machine readable.

The second of the more useful options is the '$Data::Dumper::Useqq' option which causes the data to be put out in a more normalized form which includes white space represented as meta-characters ([\n\t\r] instead of literal white space) characters that are considered unsafe (Such as the '$' will be escaped and non-printable characters will be printed as quoted octal integers.

```
#Example: multi_oo_escape.pl
use Data::Dumper;
my %chars;
 %chars = (
  #one tab and one space
  whitespace => '	 ',
  unsafe => '$',
  #literal carriage return
  unprintable => '^M'
);
my $dumper = Data::Dumper->
new([\%chars]);
$dumper->Useqq(1);
print $dumper->Dump(%chars);
```

In the 'multi_oo_escape.pl' example above we have a one of each type of character used as values in a hash that we then pass as a reference to the Data::Dumper constructor. We then set the 'Useqq' to a positive value to turn it on and then call the Dump getting an output like this:

```
$VAR1 = {
          "unsafe" => "\$",
          "unprintable" => "\r",
          "whitespace" => "\t "
        };
```

Notice that the unprintable carriage return (generated in vi using CTRL-V and then return) is printed as '\r' the tab is printed as '\t' and the single dollar is escaped to prevent it from having any special meanings. The downside to the additional functionality of 'Useqq' is that it will incur a performance penalty due to the fact that most of Data::Dumper is implemented in C (Using XSUB) whereas this function is implemented in pure Perl which has a performance hit.

Now we have covered the basic and more useful of the features Data::Dumper provides if you want to carry on experimenting with it you should look at *perldoc Data::Dumper*

They think it's all over...

The use of references is often the difference between an easy to follow and maintainable piece of code and a tangled mess of line noise and remains one of the more important areas of Perl 5 syntax to understand. Fortunately the best documentation on references (although the examples are quite terse) are included in the Perl distribution itself:

A good place to start is with 'perlreftut', its a lighter read than the others and has a number of easy to follow examples. *perldoc perlreftut*

Once you have the basics down you can either go for the in-depth details with *perldoc perlref* or go for more example code and explanations in *perldoc perllol* which focus's on arrays of arrays. More varied examples in the data structure cookbook in *perldoc perldsc*

A good final note is the reference page for Data::Dumper itself, possibly the best way of viewing or debugging references *perldoc Data::Dumper.* ■