Learning Perl Together

Ian
Norton

www.Shadow.cat

Table of Contents

Getting started

Boilerplate

Perl will always attempt to understand what we're trying to do, but sometimes we need to give it a little help.

The strict declaration tells the Perl interpreter to restrict some of the things that are generally considered unsafe. It forces you to declare variables before you use them and is generally considered best practice. We enable this with:

```
use strict;
```

The warning declaration tells the Perl interpreter that we want to know about anything it thinks we might be doing wrong.

```
use warnings;
```

Comments start with a '#' character and continue to the end of the line

```
# This is a comment
use strict; # This is also a comment
```

So let's have My First Perl Script(tm) with all of these things:

```
use strict;
use warnings;

# Print out a message
print("Hello workshop\n");
```

The “\n” is a newline character.

Simplicity

Some of the materials covered today will take a simplistic view as we only have a short time. We'll include links or manual pages to read for further information at the end of the document.

Data types

Data types allow us to declare variables to store information in memory.

Many languages have a large number of data types which define what sort of information they hold. Examples you may have seen in databases or other languages include:

- Boolean
- Character
- Integer
- Float
- String
- Array
- Datetime

Perl cares less about what kind of data you store. We only have three data types:

- Scalars
- Arrays
- Hashes

All of our different data types are declared with the 'my' keyword and distinguished by the first character of the variable name.

Scalars

What if we want to print something other than 'Hello workshop"?
Let's declare a scalar variable and use that instead.

Scalars can contain numbers, strings or references. We declare a scalar using the '$' character:

```
my $text = "Hello workshop\n";
my $number = 12;
```

And now we can use our defined variable:

```
print($text);
```

So:

```
use strict;
use warnings;

my $text = "Hello workshop\n";
print($text);
```

Arrays

What if we want more than one value? A shopping list!

- Bread
- Butter
- Jam

Arrays can only contain scalars. We declare an array using the '@' character:

```
my @shopping = ("Bread\n", "Butter\n", "Jam\n");
```

What we now discover is that our print function accepts an array:

```
print(@shopping);
```

We always get the data out in the order that we put it in with an array.

So we've seen how to create an array and to print the whole thing, but what if we only want one element of the array?

So far we've been discussing the whole of the array, now we want to talk about just one part of it.

Linguistically we'd say 'those' (plural) and 'this' (singular) for a collection or one part of it. Perl has a similar concept when talking about arrays.

We've already seen accessing the whole as a plural:

```
my @shopping = ("Bread\n", "Butter\n", "Jam\n");
print(@shopping);
```

Now let's look at just one element as a singular:

```
print($shopping[0]);
```

This prints the very first element of the array for us. Similarly we can set the singular element this way too:

```
$shopping[0] = "Brown bread";
```

It helps to think of these as accessing the same variable as either a singular or plural especially if your native language is not English.

Perl in similarity with other languages indexes arrays from zero.

Hashes

If we'd rather index our data via a key such as the name of a month in order to get the number of days in that month, then hashes are a better fit for representing our data.

January	31
March	31
April	30

Hashes can only contain scalars whilst the keys can only be strings. We declare a hash using the '%' character:

```
my %months = (
    "January",  31,
    "March",    31,
    "April",    30,
);
```

This relies on us making sure that we have an even number of arguments in our declaration. Our example is fairly clear due to the layout.

There's another, clearer way to declare this:

```
my %months = (
    January   =>  31,
    March     =>  31,
    April     =>  30,
);
```

The '=>' is known as the fat comma. It removes the need to quote the key string, assuming it has no spaces.

One more thing to note, after the value '30' in each of these declarations there's a trailing comma.

Perl allows you to have the trailing comma even though it's not needed. The advantage of this is that if and when we add another month or decide to reorder the lines, we don't need to remember to add the comma to the end of the line. It saves us having a problem later.

Linguistically Perl distinguishes between singular and plural. Our examples so far have all been of the plural form.

Let's look at one element as a singular:

```
print($months{"April"});
```

This prints the data in the 'April' key from the hash 'months'.

We need to consider how we're accessing a variable, either as a singular element or a plural whole and then select the appropriate access mechanism.

References

We've mentioned that one of the things that a scalar can contain is a reference. So what are references and why would we use one?

We first need to consider that Perl will flatten multiple arrays into one, so if we declare two lists of shopping and then put them both into one array, then we can't tell where one list ends and the second begins:

```
my @shopping1 = ("Bread\n", "Butter\n", "Jam\n");
my @shopping2 = ("Wrapping paper\n", "Birthday card\n", "Sellotape\n");
my @shopping  = (@shopping1, @shopping2);
```

What if we don't want the two arrays merged? Lets say that these two lists need to come from different shops.

This is when references become useful. So how do we create references? There are two techniques, one uses an existing array or hash and simply converts it into a reference:

```
my $array_ref = \@shopping;
my $hash_ref  = \%months;
```

The other approach uses different brackets to create the reference directly:

```
my $array_ref = ["Bread\n", "Butter\n", "Jam\n"];
```

Note the use of square brackets instead of round. Because we don't have a variable mapping directly to the array, this is called a reference to an anonymous array.

```
my $hashref   = {
    January => 31,
    March   => 31,
    April   => 30,
};
```

Note the use of curly brackets instead of round. Similarly, this is called a reference to an anonymous hash.

So our example becomes:

```
my $shopping = {
    supermarket => \@shopping1,
    giftshop    => \@shopping2,
};
```

Our two arrays are shown here in a hash linked to by references to our two arrays.

So things change very subtly here again, because we need to change the way in which we access the elements via a reference, now we need to insert a '->' into the mix.

Let's change our shopping 1 and 2 declarations to be anonymous array references:

```
my $shopping1 = ["Bread\n", "Butter\n", "Jam\n"];
my $shopping2 = ["Wrapping paper\n", "Birthday card\n", "Sellotape\n"];
```

Now, in order to access an individual element, we do this:

```
my $element0 = $shopping1->[0];
```

In order to fetch the supermarket anonymous array reference from the shopping hash, we'd do this:

```
my $shopping1 = $shopping->{"supermarket"};
```

Let's look at that in slow motion (this is long hand).

```
# Declare arrays shopping1 and shopping2
my @shopping1 = ("Bread\n", "Butter\n", "Jam\n");
my @shopping2 = ("Wrapping paper\n", "Birthday card\n", "Sellotape\n");

# Declare the hash 'shopping' with the keys 'supermarket' and 'giftshop'
# pointing to the arrays 'shopping1' and 'shopping2' by reference
my %shopping = (
    supermarket   => \@shopping1,
    giftshop      => \@shopping2,
);
```

Which is the same as (losing the arrays):

```
# Declare references shopping1 and shopping2 to anonymous arrays
my $shopping1 = ["Bread\n", "Butter\n", "Jam\n"];
my $shopping2 = ["Wrapping paper\n", "Birthday card\n", "Sellotape\n"];

# Declare the hash 'shopping' with the keys 'supermarket' and 'giftshop'
# pointing to the references 'shopping1' and 'shopping2'
my %shopping = (
    supermarket   => $shopping1,
    giftshop      => $shopping2,
);
```

Which is the same as (losing the references all together):

```
# Declare the shopping hash containing two anonymous arrays
# of shopping keyed by shop
my %shopping = (
    supermarket   => ["Bread\n", "Butter\n", "Jam\n"],
    giftshop      => ["Wrapping paper\n", "Birthday card\n", "Sellotape\n"],
);
```

So, to unpack that:

```
my @shopping1 = @{$shopping{"supermarket"}};
my @shopping2 = @{$shopping{"giftshop"}};
```

Prefixing the reference with the desired outcome and surrounding it with curly braces will convert back for us. This is something that Perl won't do for us.

There's always more than one way to do it. If you find yourself with a complex data structure, sometimes it can help to think about exactly how that's constructed and then break it down layer by layer.

The more experienced the developer, the more of these logic shortcuts they will take. Mostly, they're simpler than they look!

Here's a more complex example, but equally simple to understand and to break down:

```
# Declare an empty anonymous hashref called 'config'
my $config = {};

# Info on the exim MTA
$config->{"mta"}->{"exim"}=>{
    author    => "Phillip Hazel",
    licence   => "GPL2",
    protocols => [ "SMTP", "MBOX", "Maildir", "Maildir++", "SASL", "ESMTP"],
    site      => "http://www.exim.org/",
    version   => "4.80.1",
};
```

So taking that apart, we can first declare an array for the protocols supported and then declare a hash for the other information about Exim then go on to add the additional layers of hashes:

```
# Declare an array 'exim_protocols' with a list of protocols supported
my @exim_protocols = ("SMTP", "MBOX", "Maildir", "Maildir++", "SASL", "ESMTP");

# Declare a normal hash 'exim_hash' and add info
# including a reference to the @exim_protocols array
my %exim_hash = (
    author    => "Phillip Hazel",
    licence   => "GPL2",
    protocols => \@exim_protocols,
    site      => "http://www.exim.org/",
    version   => "4.80.1",
);

# Declare an 'mta_hash' hash and add exim to it by reference
my %mta_hash    = ( exim  => \%exim_hash );

# Declare a 'config' hash and add the 'mta_hash' to it by reference
my %config_hash = ( mta   => \%mta_hash );

# Declare a config hashref to 'config_hash'
my $config = \%config_hash;
```

And so, you know know how to put hashes in hashes, arrays in hashes, hashes in arrays, and arrays in arrays. Exactly the same principle is used.

Making decisions

Context (void, scalar and list)

Like most people, Perl cares about how you ask the question not only what you ask, context is important.

Void, scalar or list?

When we call a function and don't assign the returned value, that's void context:

```
print("Hello\n");
```

When we call a function and assign the value to a scalar, that's scalar context:

```
my $count = length(“Hello”);
```

And yes, when we assign the value to an array, that's array context:

```
my @array = (1, 2, 3);
my ($val1, $val2, $val3) = @array;
my ($val1) = @array;
```

If we call an array in a scalar context, we get the number of elements in the array:

```
my $count = @shopping;
```

If we're using references, then the context can get a little wierd.

```
my $arrayref = [1, 2, 3];
my $count = $arrayref;      # This doesn't do what we want
```

That's scalar context, because a reference is a scalar. So $count now points to the same anonymous array as arrayref. So how do we do that? Help the Perl interpreter and tell it we're working on an array:

```
my $arrayref = [1, 2, 3];
my $count = @$arrayref;
```

Functions will normally tell you how they operate if there is significance to the different contexts.

Looping (for, foreach, while and until)

The four types of commonly seen loops in Perl are:

- for
- foreach
- while
- until.

The Perl for loop is used in a manner familiar to those coming from a C based background such as C++, Java, .Net, PHP etc.

We have three portions:

- Initialisation
- Test
- Operation

Before entering the loop, the initialisation is performed.
At the start of each loop, the test is performed.
At the end of each loop iteration the operation is performed.

So we can count from zero to 10 like this:

```
for ( my $i = 0 ; $i <= 10 ; $i++ ) {
    print("$i\n");
}
```

foreach allows us to iterate over the contents of an array but also allows us to pass a range such as '1..10' and iterate through the values.

This is a good point for us to introduce the concept of the default variable '$_'.

```
foreach ( 1..10 ) {
    print("$_\n");
}
```

For a set of built-in functions, if no variable is specified but an operation or function call returns data (so in void context), then it ends up in the default variable. We can also specify a variable instead of relying on this though:

```
foreach my $i ( 1..10 ) {
    print("$i\n");
}
```

More details at http://perldoc.perl.org/perlvar.html#General-Variables.

A while on the other hand will loop until the test condition is false:

```
my $i = 0;

while($i <= 10) {
    print("$i\n");
    $i++;
}
```

Typically, a for or foreach loop is used when we know how much data we're dealing with in advance whilst a while loop is used when we don't know. A good example would be reading the contents of a file or waiting for user input as we don't know when the end will be reached until we get there.

Much in the same way that 'while' loops until a condition is false, 'until' loops until a condition is true:

```
my $i = 0;

until($i == 10) {
    print("$i\n");
    $i++;
}
```

Until is purely syntactic sugar that allows us to say "loop until this is true" rather than "loop while this isn't true".

Flow control

Comparing strings and numbers

The offshoot of Perl only having three data types is that it needs help to know how to compare two variables. Let's look at the following variables:

```
# The string "0.0"
my $value1 = "0.0";

# The string 0 (which is a numerical zero)
my $value2 = "0";
```

Are these two things the same? That depends on how we compare them.

If we compare them as numbers, then indeed they are the same, but when compared as strings they are not the same.

Perl attempts to convert the scalar from a string to a number as needed, in order to help the interpreter use the right comparison there are two sets of comparison operators, one for numbers and one for strings:

Numeric	String	Explanation
==	eq	Equals
!=	ne	Not equals
<	lt	Less than
>	gt	Greater than
<=	le	Less than or equal to
>=	ge	Greater than or equal to

So our tests would look like this:

```
# String
if($value1 eq $value2) {
    print("String equivalence\n");
}

# Numerical
if($value1 == $value2) {
    print("Numerical equivalence\n");
}
```


if, elsif and else.

We've briefly touched on the use of 'if' in the previous examples. Our 'if' statement looks to see if the comparison within the brackets is true or false.

We've mentioned that there's no boolean datatype in Perl, so how do we determine truth?

```
if($value == 0) {
   print("Value is zero\n");
}

elsif($value == 1) {
   print("Value is one\n");
}

else {
   print("The value was something else\n");
}
```

unless

If we want to negate an if condition to say:

```
if($value != 0) {
    print("Value is not zero\n");
}
```

We can use unless as a short hand:

```
unless($value == 0) {
    print("Value is not zero\n");
}
```

It's syntactic sugar that you may or may not find useful much like until.

If we consider the 'defined' function that checks to see if a variable has been set to something, then it's much clearer how this could be useful:

```
my $default_value = "3.142";

unless(defined($variable)) {
   $variable = $default_value;
}
```

This sets a default value in 'variable' if one isn't supplied.

Subroutines

Modular and reusable code.

There comes a point when code passes from a few lines of simplicity to being something greater and more complex, kind of a graduation if you will.

A simple example to count from 1 to 10:

```
use strict;
use warnings;

print("Starting count\n");

for(my $i = 1 ; $i <= 10 ; $i++) {
    print("Count $i\n");
}

print("Count complete\n");
```

Lets consider that we want to count multiple times, once to ten and then to twenty:

```
use strict;
use warnings;

print("Starting count\n");

for(my $i = 1 ; $i <= 10 ; $i++) {
    print("Count $i\n");
}

for(my $i = 1 ; $i <= 20 ; $i++) {
    print("Count $i\n");
}

print("Count complete\n");
```

This is a horrible copy and paste job if you imagine that we do this ten times. Time to consider splitting code into subroutines.

Writing a subroutine

Subroutines allow us to re-use sections of code. The keyword for declaring a subroutine is 'sub'.

Here's a 'count' subroutine:

```
sub count {
    for( my $i = 1 ; $i <= 10 ; $i++ ) {
        print("$i\n");
    }
}
```

This takes no arguments and simply outputs the count one to ten. We'd call it like this:

```
count();
```

So this doesn't get us much further because we can still only count to ten.

Arguments to a subroutine

So what if we wanted to pass the upper limit to the subroutine instead of hard coding it? We'd need to pass an argument:

```
count(20);
```

So our subroutine now has to take the passed argument, it does this by using the shift function which operates on the default array variable @_:

```
sub count {
   my $max = shift;

   for( my $i = 1 ; $i <= $max ; $i++ ) {
      print("$i\n");
   }
}
```

Each argument to the subroutine is an element within the default array so it becomes clear why using references can be significant when passing two arrays to a function.

The problem with this approach to unpacking the default array is that we have to do it in the same order each time and we can't make an argument optional.

Enter named arguments. Instead of supplying a list of arguments, let's supply an anonymous hash by reference:

```
count({ max => 20 });
```

So within our subroutine, we need to take the reference to the hash and then we can look at the supplied arguments:

```
sub count {
    my $args = shift;
    my $max = $args->{"max"};

    for( my $i = 1 ; $i <= $max ; $i++ ) {
        print("$i\n");
    }
}
```

Perl Best Practices recommends this latter named arguments style of subroutine argument parsing once you have more than three parameters or any subroutine that's ever *likely* to have more than three parameters.

Now we can see where our unless statement becomes useful:

```
unless(defined($args->{"max"})) {
    die("Count maximum wasn't defined.");
}
```

Die embraces the Klingon Programming Paaradigm.

Return successful or die!

Learning more - the reading list...

Manual pages

syntax	http://perldoc.perl.org/perlsyn.html
datatypes	http://perldoc.perl.org/perldata.html
references	http://perldoc.perl.org/perlref.html
	http://perldoc.perl.org/perlreftut.html
subroutines	http://perldoc.perl.org/perlsub.html
	Don't read about prototypes, they're really not recommended any more.

Books

Damian Conways book ***Perl Best Practices*** inspired perlcritic. This is a set of modules that checks your code against the recommendations in the book. Whilst you might not agree with 100% of the book, it's certainly a damn fine way of mustering your views and rationalising your reasoning. You don't have to agree, but you should probably have a reason if you don't ;)

The C Programming Language (a.k.a. K&R). One of the defining books on the C programming language is still a brilliant grounding for modern computer science and programming techniques. You don't have to read it cover to cover, but dipping in and out for some of the explanations of algorithm design and techniques are excellent.

Beginning Perl is a free download from http://www.perl.org/books/beginning-perl/
Modern Perl is also a free download from
http://www.onyxneon.com/books/modern_perl/index.html

Programming Perl a.k.a. The Camel Book is now in it's fourth edition, and available today from O'Reilly :) One of the definitive works on Perl.

Perl community

There are lots of channels available on irc.perl.org supporting just about everything Perlish.

If you're looking for somewhere to ask beginner questions then irc.freenode.net #perl is a really good place to go, as is your local Perl Monger group (check http://www.pm.org/ for a list).

Workshops, conferences and hack days

London Perl Workshop
Watch out for things on the Enlightened Perl Organisations Ironman blog aggregator at http://ironman.enlightenedperl.org/

Organisations

The Enlightened Perl Organisation fund raises and supports work within the Perl community. Go join them, or at least give them a few quid for your love of Perl.

The Perl Foundation fund raises and supports the core of the Perl language. They want your money too!

CPAN

No Perl talk would be complete without a mention of the Comprehensive Perl Archive Network or CPAN. This is a massive archive of just about anything you might want. Don't forget to contribute by submitting bug reports and fixes. Documentation fixes are just as important as code.

Other tutorials and resources

http://perl-tutorial.org/