Perl
THINKING IN LINE NOISE

Perl is a language steeped in the history and evolution of Unix (and by extension Linux) platforms, so it's only right that it should have a place here at Linux Magazine. Dean Wilson and Frank Booth begin our journey with an overview of Perl and its syntax

Originally released in 1987, Perl has spread from niche to niche (including CGI, Databases and XML), assimilating buzzwords that stray unwittingly into its path, such as Object Orientation, Bio-informatics and Aspect Oriented Programming). For all of the above reasons, Perl is renowned as a 'glue-language': it interacts with most popular applications.

The Comprehensive Perl Archive Network (CPAN) repository is one of the jewels in Perl's crown (groan). It provides a library of language extension modules as comprehensive as J2EE and the .NET framework, providing a set of APIs that enables integration with other languages including C, C++ and Java, to name but a few.

Perl's strength has always been its active user community, which created and maintained sites such as CPAN (*http://www.cpan. org*), Perl.com (*http://www.perl.com*), use.perl (*http://use.perl.org*), Perl Monks (*http://www.perlmonks.org*) and various geographically diverse Perl Mongers groups.

As well as having sites devoted to it, Perl runs some of the busiest sites on the Net, including geek-havens Slashdot (*http://www.slashdot.org*) and Kuro5hin (*http://www.kuro5hin.org*). In fact Perl is so widely used on the Web that it's often referred to as the duct tape of the Internet.

Perl is a terse but high-level language that removes the burdens of memory allocation, the distinction between primitive data types, file handling and the need to constantly reinvent the wheel. It is because Perl allows the developer such freedom and functionality within so few key-strokes that Perl has the semi-deserved reputation of resembling line-noise.

Perl's integrated regular expression handling (a super-set of the POSIX standard) – the variety of operators provided to manipulate, describe and access Perl's data-structures – has meant Perl had to spill over to lesser-used areas of the keyboard or adopt a larger, more esoteric vocabulary. Of course, big words can sometimes obscure meaning – just take the last sentence as an example – so more obscure keyboard symbols were instead adopted.

Here's a snippet of Perl code seen in production

Getting Perl

Any moderately recent and well-stocked Linux distribution will come complete with an installed Perl interpreter, a full set of Perl core modules and the standard (and copious!) documentation in pod format.

If your install does not include Perl then there are two paths open to you; you can either get a binary distribution from your distro's package repository or download and compile your own. As this is a beginner's tutorial we will cover getting the package and installing it rather than compiling your own; this topic is more than adequately covered in the INSTALL file in the root of the source code tarball.

Installing the binary package varies more upon your Linux distribution but can be summarised as:

rpm-based

Step 1: Download the package from either your distro's repository or from one of the links at *http://www.rpmfind.net* or *http://www.perl.com*.
Step 2: As root, issue the *'rpm -i <perlpackage>'* command.

Debian

Debian saves you the wasted time fetching the package by hand and instead allows you to get by with the following:
Step 1: apt-get update.
Step 2: apt-get install perl.

While Debian makes the initial install simpler; for some packages that have external dependencies you are reliant upon the apt-get mechanism, as an example modules that use Libmagick or expat (an XML parser) must be installed via apt-get or will require modification of the source to allow a successful install.

The code is confusing due to the high frequency of special characters

software that illustrates why Perl's syntax is so easily misunderstood and consequently decried:

```
$/=\0; $_=<>; tr/A-Z/a-z/;
%_=map{$_,1}/[a-z0-9_.-]+@[a-z0-9._-
]{3,67}(?=\W)/g;@_=sort keys%_;
```

Although it has to be said that Perl needn't be written like this.

Scalars

In case you were wondering, the previous example finds email addresses in a file, removes duplicates and sorts them alphabetically. The code is confusing due to the high frequency of special characters (sigils). The most common and essential of these in everyday programming is *$*.

$ denotes a scalar variable. In Perl, scalar variables are use to hold numbers, text and many more types of data. For example:

```
$percent = 12.7; # Assign 12.7 to the ➥
variable $percent
$count = 1;  # Assign the value 1 to the ➥
variable $count
$name = 'Guido'; # Assign the string ➥
'Guido' to $name.
$beast = $name; # Copy the value of ➥
$name to $beast
```

Below are the most popular methods to alter numeric scalars:

```
$count = $count + 1;  # count now equals 2
$count +=1;  # count now equals 3
$count++;  # count now equals 4
```

The first example is probably the simplest to understand. *$count* is set to the value of *$count* + *1*. The operator += in the second example is shorthand for the same function, it can be applied to the multiply, subtract and division operators amongst others. The final line of code uses the post increment operator ++, this adds one to the existing value of *$count*. There is also a post decrement function –– that subtracts one from *$count*.

Perl has a rich variety of ways to assign and manipulate strings.

```
$curry = 'Chicken';  # This sets $curry ➥
to Chicken
$curry = "$curry Phaal"; # This sets ➥
$curry to: Chicken Phaal
```

In these examples the value of *$curry* is manipulated using string operators. As with numeric operators the strings are assigned using the equals operator. In the second example the use of a variable inside double quotes replaces the variable *$name* with its currently assigned value, the official term for this is "interpolation of the variable".

```
$mistake = '$curry';  # This sets $mistake ➥
to literally: $curry
```

Unlike its predecessor, the above example uses single quotes which prevents the variable from being interpolated: it returns the literal value within the quotes:

```
$word = $curry . ' is ';  # This sets ➥
$word to: Chicken Phaal is
$word .= 'tasty';   # This sets $word to: ➥
Chicken Phaal is tasty
```

The dot operator *'.'* is used to concatenate values together. In these last two examples the dot operator is used to append strings to *$word*; in the latter case using the same philosophy as the += operator. Note that concatenating single quoted strings to a variable does not affect the interpolation of the variable that is not wrapped in quotes.

Perl allows us to use the string operators on numbers (it treats the numbers purely as characters) and strings as numbers (by taking the numeric part of the string until the first non-numeric character):

```
$count = 3; # Set the value of $count to 3
$order = "$count $curry"; # Set $order to: ➥
3 Chicken Phaal

$count += $order;  # $count = 6
```

In this example the numeric part of the string (*3*) is added to the value of *$count*, the remaining part of the string *$order* is ignored.

```
$order = $count . $curry; # $order is now: ➥
63 Chicken Phaal
```

Using concatenation the value of *$count* is prepended to *$order*.

Listing the ways

While scalar variables are useful in day-to-day programming they alone are not adequate for more complex programs. Every modern language has developed more complex data types such as arrays and hashes; Perl is no exception. Perl's arrays are indexed by integers and dynamically sized – you don't need to set a maximum size of an array when you create it and the array will resize itself as elements are added and removed.

```
@Foodgroups = ('curry', 'kebabs', "ice cream");
```

In the previous example we create an array called Foodgroups and populate it with three values, note that the values can be single or double quoted and that the rules of scalar quoting apply in the assignment. All arrays in Perl are indicated by the @ character, indexed by integers and start at *0*, so in the example *curry* is at position *0* and *ice cream* is at position *2*.

```
# Prints "After curry we have ice cream"
print "After $Foodgroups[0] we have ➥
$Foodgroups[2]\n";
```

Notice that in the example's 'print' statement we use the scalar *$* sigil rather than the @ for array; this is because we are accessing a scalar at the position of the given value, called a subscript, that is in the square brackets. If you wish to change a value in an array and you know its position you can use the same syntax without impacting the rest of the array. If you try and retrieve a value from an index that does not exist then an *undef* will be returned and the size of the array will not be changed.

```
$Foodgroups[2] = 'beer';

# Prints "After curry we have beer"
print "After $Foodgroups[0] we have ➥
$Foodgroups[2]\n";
```

While being able to directly access a value by its index is useful in many cases for the programmer to work on the start or the end of the array. Determining the length on a dynamically sizing array is easier than you might think using what are known as negative subscripts:

```
print $Foodgroups[-1]; # Prints "beer"
```

If you try and retrieve a value from a non-existent negative position using a negative subscript then the *undef* value is returned and the size of the array is not modified. If you try and store a value in a non-existent negative position the Perl interpreter will generate a fatal error.

While working with arrays is comparatively simple, an area many people new to Perl find confusing is the difference between the length (number of elements) in an array and the last position in the array. Because the last position is a scalar value again we use the *$*.

```
print $#Foodgroups; # Last position. ➥
This prints 2
print scalar(@Foodgroups); # Number of ➥
elements. This prints 3
```

In the second line of the example we introduce a new function, 'scalar'. While Perl is often smart enough to do automatic conversion of variables to suit the current context, in places where the usage is ambiguous and more than one usage may appear correct we can give the interpreter a helping hand. By using the 'scalar' function we tell Perl to give us the length, if we run the snippet again without the 'scalar' function then we get a completely different result:

```
print @Foodgroups; # This prints ➥
'currykebabsice cream'
print "@Foodgroups"; # This prints 'curry ➥
kebabs ice cream'
$" = ' and ';
print "@Foodgroups"; # This prints 'curry ➥
and kebabs and ice cream'
```

In the first line of the example we print the array without telling Perl a context so it picks the most obvious one (to itself) and prints all of the array's literal values. The second line of code wraps the array in double quotes and the values are printed out in a more readable form. The spaces that are emitted from seemingly nowhere are dictated by another of Perl's implicit predefined variables, *$"* or the "List Separator" as its known in Perl parlance. If you set this variable directly, as we do in the third line, and then reprint the array in a double quoted string each element of the array is printed with the separator between them.

As arrays are collections of values it is often desirable to iterate through an array, repeating an operation for each element. There are two simple ways of doing this and the first way illustrates one of the places where inexperienced Perl programmers can confuse array position and array length. Given below are four small *for* loops, two that are valid and do as expected and two that do not. See if you can pick out which are which:

```
for ($i=0; $i <   @Foodgroup; $i++) {
   print "$Foodgroup[$i]\n";
}
for ($i=0; $i <= $#Foodgroup; $i++) {
   print "$Foodgroup[$i]\n";
}

for ($i=0; $i <= @Foodgroup; $i++) {
   print "$Foodgroup[$i]\n";
}

for ($i=0; $i <  $#Foodgroup; $i++) {
   print "$Foodgroup[$i]\n";
}
```

Inexperienced Perl programmers can confuse array position and array length

The first two examples are both valid, they will iterate through the array incrementing *$i* on each pass, so that each indexed value will be printed once.

The final two examples are both incorrect; The third line of the example executes the loop body for once too often (if there are three things in @Foodgroup, the loop executes when *$i* is 3, which is incorrect as it's not a valid position). The final loop executes the body of the loop one time too few (if the final element is at position 2, the loop stops after executing the body of the loop with *$i* set to 1).

It is common to use either a *for* loop (shown above) or a *foreach* loop to be able to operate on every item in an array without knowing anything about the array other than its existence. The most visible difference between the two is that *foreach* loops use an alias for the value rather than storing an

Perl's ability to use implicit values is both one of its benefits and banes

index. This is useful when it's unnecessary to know the index positions:

```
foreach $Food (@Foodgroups) {
  print "$Food is bad for you\n";
}
```

Or, if you want to make your code a little more implicit and you call a number of functions in the loop that use *$_* as their default variable you can execute the loop without an alias yet still have it process the values:

```
foreach (@Foodgroups) {
  print "$_ is bad for you";
  print length, "\n";
}
```

The above *foreach* loop will print out the message with each value and then print out the length of each value. This is possible because in the absence of an argument Perl refers *length* to *$_* and *print* then prints the value that *length* returns.

Perl's ability to use implicit values is both one of its benefits and banes, depending on how sensibly it's used. The *for* and *foreach* loops are almost identical in functionality and can be used interchangeably; you should use the version that is easier to read in your code.

Hashes – associative arrays for lazy typists

An associative array is a data structure, which is accessed by a string value called a "key" rather than an index, as seen in arrays. In Perl, associative arrays are used so frequently they're called "hashes" which is easier to say.

The % symbol denotes that a variable is a hash. As with arrays, hashes utilise brackets to access individual elements. For hashes, curly braces are used. To assign a variable to a hash we need to specify both the key and the value:

```
$hash{'key'} = 10; # %hash now has a key ➥
with a value 10
```

Again the *$* prefix is used when accessing an element of the data-structure because the element will be a scalar variable. So the only thing differentiating the hash from a normal array is the shape of the braces. For a hash the curly braces encapsulate the key.

The following example illustrates how those brackets alter the semantics of the entire line:

```
%a; # An associative array
@a; # A traditional array
$num = 3; # A numeric scalar value

# Assign a value to each
$a[$num]='Array'; # Puts "Array" in the ➥
4th element of @a
$a{$num}='Hash'; # Associates "Hash" to ➥
3 in the hash %a
```

The keys in a hash are unique, so if a value is assigned to a key, the previous value will be over-written and lost. At first this seems to be a disadvantage: it's one of Perl's most heavily exploited features, as we will discussed later.

```
$hash{six} = 6; # Value of the key 'six' to 6
$hash{six} = 9; # Value of the key 'six' ➥
to 9, no longer 6
```

Initialising a hash is similar to the methods used for arrays. A hash can be initialised with a full complement of keys and values. Hashes utilise array and list operators but the manner in which the data is manipulated is subtly different.

```
%numbers = ( 'one',1,'two',2,'three',3);
```

This expression assigns the following keys and values to the hash:

```
keyvalue
one1
two2
three   3
```

The hash knows to pick the first element as a key and the next as a value. Elements are read from the list and alternately given the role of key or value. Perl will complain (when run under warnings) if the list contains an odd number of elements. However, the last key will be included and its value will be a *undef*.

The => operator is used to improve the legibility of list assignments when initialising a hash. It allows us to quickly differentiate the keys and values within the list. The value to the left of => is the key of the element, the item to the right is the value. Using the => operator also means that the key needn't be wrapped in quotes if it's a single word.

```
%hash = ( six => 6, seven => 7, ten => 10);
```

Hashes have a few explicit functions as well as borrowing many of the list functions, the most popular are:

keys which returns the keys found in a hash as a list

values which returns the values of the hash as a list

Each of these functions returns a list, which will be in seemingly random order. If order is needed it must be imposed using the function 'sort'.

```
%a = ( ten=>10, nine=>9, eight=>8, seven=>7);
@b = keys %a # Places keys in array
@c = values %a # Places values in array
```

The order of elements in *@b* (the array of key elements) may be: nine, seven, eight and ten. The order of elements in *@c* will then be: 9, 7, 8, 10. Regardless of the order of elements, key and value are returned at the same point.

The functions keys and values are frequently used to traverse the contents of a hash; there are several methods of accessing every element of a hash:

```
for my $key (keys %a){
   $value = $a{$key};
   print "$key => $value\n";
}
```

This is probably the most common way of accessing all elements in an array. Using a *for* loop in the same manner, we would use it to access all the elements of an array.

There are two more functions used with hashes, the *delete* and *exists* functions can also be used with arrays, but are more commonly seen in code relating to hashes.

The *delete* function removes elements from hashes. Removing an element with the key *'fred'* is expressed in the following way:

```
delete $passwd{'fred'};
```

exists is a function that, given a key in a hash, will return true if that element is present. To make full use of the *exists* function we need to use it with a conditional operator. In the example we use *if*. When running under warnings, it is prudent to use *exists* before calling an element of a hash if it's doubtful that the key is present.

```
%passwd=( fred => 'xSSx13A0Oav', root='root');

if ( exists($passwd{$user}) ){
   print "Success, that user was found\n";
}
else {
   print "Sorry, that user was not found\n";
}
```

if tests the return value from the *exists* function; if the hash element does exist then *if* will run the code wrapped in the curly braces that follow it. When a function evaluates to false the *'if'* statement disregards the first set of curly braces and executes the contents of the curly braces following *else* if *else* is present.

Here are some simple examples of the common uses for hashes in Perl:

● Creating a look-up table of values to substitute:

```
%dns=(10.3.1.0 =>'firewall', 10.3.2.0 => ➥
'email', 10.3.0.1 => "bob's machine");
print "$dns{$ip-address}\n";
```

● Removing duplicates from a structure by exploiting a hash's use of unique keys:

```
@a=(1,2,1,2,4,6,7,2,1,10,6,7,8,8); # ➥
Initialise the array
%a=map{$_=>1}@a; # Make a hash where the ➥
keys are the elements of @a
@a=keys %a; # Reassign @a so that it ➥
contains unique values
```

At the beginning of this example *@a* contains (1,2,1,2,4,6,7,2,1,10,6,7,8,8), after being filtered through the hash *@a* contains (7,8,1,2,10,4,6).

Focusing on line 2 of the above example, *map* is used to create a key value pair for the hash. The value being assigned is irrelevant. The only important function occurring is taking place implicitly: for keys that are already in existence the value will be over-written (by an identical value) since keys in a hash are unique.

Perl documentation

Perl has a wealth of documentation which comes with the standard distribution. It covers every aspect of the Perl language and is viewed using your computer's default pager program. Perldoc pages resemble manpages, citing examples of use and pertinent advice.

There are many parts to the Perl documentation. To list the categories, type the following command at the shell prompt:

```
perldoc perl
```

The page displayed for the previous example has two columns; the left column lists the mnemonic titles and the right column a description of the topic.

```
perlsyn  Perl syntax
perldata Perl data structures
perlop   Perl operators and precedence
perlsub  Perl subroutines
```

To invoke documentation for a subject, simply type perldoc and the mnemonic for that topic on the command line. The example below will display the documentation for "Perl Syntax".

```
perldoc perlsyn
```

A further use of perldoc is to read the usage for any of Perl's functions, this is done by calling perldoc with the -f option and the function name as an argument. The following example will display the documentation for the function *map*.

```
perldoc -f map
```

Perldoc also provides quick access to frequently asked questions about Perl.

```
perldoc -q punctuation
```