Perl of wisdom

Nigel Whitfield walks through the basics of Perl, a hugely versatile programming language.

If you're a regular reader of the *Hands On* section of *PCW*, you'll have seen countless references in the *Internet* and *Web Development* sections to Perl. It's a programming language that has become almost a standard part of the Unix operating system and can also be found in versions for Windows, Macintosh and pretty much anything else you care to name.

Each mention of Perl in *PCW* is usually accompanied by many requests for more information – far more than can be fitted within the constraints of a *Hands On* column. Hence this introduction, which will hopefully help you to start writing your own useful scripts.

Part of the appeal of Perl is that it's powerful and flexible – it's been called a Swiss Army chainsaw by some. One of the most useful aspects of the language, especially for web and database work, is the powerful string-handling facilities, which allow you to search for complicated patterns and replace them with something else, or to edit text quickly and easily.

Out of type

Those who are used to other programming languages might be confused – or even startled – by some aspects of Perl. You don't need to declare any variables, for instance. And if you need to use a counter as a number in one place, and a string elsewhere, you don't need to convert it. Treat it how you need to, when you want to, and Perl will convert it on-the-fly.

You can, for example, say:

```
$incvat = $price * 1.175 ;
print "The total price ↙
```

A glossary of some basic Perl

This is not a comprehensive list of commands and structures, but should get you started. The *Perl Reference Guide* (free from the Internet) has a summary of language and core functions.

Variables

`$a`
A string or numeric variable

`@a`
The whole array ($a[0], $a[1] ....$a[x]).

`%a`
An associative (indexed) array.

`$a{x}`
An element of the associative array %a. For example $members{'nigel'}.

`@ARGV`
An array containing the arguments given when the script was started.

`$_`
The current line being read from a file, or in a loop.

Program flow

A block of commands can be grouped together within curly brackets, as in the examples here.

```
do { commands } ↙
until (condition)
```
while they can also be used at the end of the loop.

```
if (condition) ↙
{ commands }
if (condition) ↙
{ commands }
 Else { alternative
 commands }
```
Use elsif to add another condition.

`last`
Finish executing a loop.

`next`
Go to the start of a loop.

```
while (<HANDLE>) ↙
{ commands }
```
Perform commands as long as there is data available from the file referenced by HANDLE. $_ is set to each line in turn.

Useful functions

`close HANDLE`
Close the file referred to by HANDLE.

`keys(%a)`
Return an array of all keys for an associative array.

```
open( HANDLE, ↙
$filename)
```
Open a file, which can be referred to by HANDLE. Files are open for reading by default. Use "> $filename" to write to a file, ">> $filename" to append.

`print`
Display output. Use print FILEHANDLE 'something' to send data to an open file.

`printf`
Similar to print, but uses formatting similar to the C language printf function.

`s/search/replace/`
Substitute a regular expression for another in a string. Add a trailing i to make it case-insensitive, or g for global to change occurrences in a string.

`sort( @array )`
Return a sorted array, eg @name = sort(keys (%members)) ;

```
split( /pat/, ↙
$variable, parts)
```
Return an array by splitting a string into two or more parts, eg ($firstname, $surname) = split(/,/, $wholename) ;

`values(%a)`
Return all the values held in the array %a

Operators

`=`
Set a variable to a value, eg $vatrate = 0.175.

`==`
Test if variable has a numeric value, eg if ($counter == 0) .... Use != for not equal.

`eq`
Test to see if a variable has a string value, eg if ($message eq 'hello') ... Use ne for not equal

`&&`
And. True if both left and right-hand expressions are true, eg if (($a == $b) && ($msg eq 'hello'). Or use as a short form of if... then, such as ($command eq 'quit') && exit ;

`||`
Or. Used same way as &&.

`=~`
String matching, used either to match a pattern, or in a substitution, eg if ($a =~ /pcw/) or $a =~ s/search/replace/ ; Use !~ for not matching.

`.=`
String concatenation, eg $name .= $lastname . ', ' . $firstname ;

Beyond the web

While many use Perl as a web tool, it has a lot of other uses. Whenever you need to process text files, it can make life easier.

Some organisations use Perl to take the text files exported from a database and mark them up with codes for DTP programs. QuarkXPress and Ventura Publisher can read files with 'tags' which look like HTML, but with more options. By laying out a page in a program such as this, and saving it as a tagged text file, you can see what you need to add to your database output to allow the DTP program to set the attributes when you import it.

But how do you get your text there? One technique is to write your script so it reads all the files in a particular directory when it's run.

This example code will look for files ending in .csv and process them in turn, leaving a file with the same name, but ending in .gen instead.

```
# loop over all ↙
the csv files
foreach $in ↙
( <*.csv> ) {

$out = $in ; $out ↙
=~ s/\.CSV/\.GEN/i ↙
; printf ↙
("Converting %s to ↙
%s\n", $in, $out) ;

open( INPUT, $in ) ↙
|| die("Can't open ↙
input file $in\n") ;
open( OUTPUT, "> ↙
$out") ;

# now read the ↙
lines from the ↙
input file
while (<INPUT>) {

# put code here to ↙
process the file,
# sending your ↙
results to OUTPUT ↙
using
# print OUTPUT ↙
$results
}

close INPUT ;
close OUTPUT ;

# now do the next ↙
file, if any
}

exit ;
```

And that's it. When the program's run, the <*.csv> in the foreach statement is expanded to the CSV files in the current directory. The i on the end of the substitution command is needed, since a Windows system will match .CSV and .csv – while the filenames look different, Windows doesn't really understand upper and lower case. So how do you turn the CSV file into something useful? Take advantage of the fact an array in Perl can be written as ('value 1' , 'value 2', 'value 3'), and use the eval command. As long as there are no special characters to cause problems, you can treat a line from a CSV file as Perl code, splitting it into individual fields, as in the following:

- Assume the line of text is called $text
- The CSV file looked like 'field 1', 'field 2'
- Start by prefixing single quotes, and turning double to single

```
$text =~ ↙
s/\'/\\\'/g ;
$text =~ s/\"/\'/g ;

( $field1,
 $field2,
 $field3 ) = eval( ↙
"(" . $text . ")" ↙
) ;
```

(*Key: ↙ code string continues*)

```
including VAT is $incvat\n" ;
```
(*Key: ↙ code string continues*)

There are a few other basic parts of Perl there – each command ends with a semicolon, variables start with a $, and if you enclose a string in double quotes, instead of single ones, then it's interpreted to substitute variables and special characters – such as the \n, which is a new line.

Before we get ahead of ourselves though, first things first. Where do you get Perl? If you have a Unix or Linux system at your disposal, chances are that it is already installed. If you're using Windows or a Mac, you can download a free version. And best of all, the scripts that you write on one Perl platform will run on others, provided you've not used any platform-specific features. So you can test your program on Windows, and be sure that when you upload it to a Unix web server, you won't have any problems.

SolutionSoft's PerlBuilder will even simulate the actions of a web server, allowing you to fill in dummy forms and check that your script processes them correctly.

The biggest difference that you are likely to see between platforms is simply the way you launch your scripts. On a Unix system, starting them with the line

```
#!/usr/bin/perl
```

will let you type the name of the script, and the system will find the path to the Perl interpreter from the first line.

On Windows you will need to set up the shell to recognise files with an extension (such as .pl) as Perl scripts to run when you double-click on them, or start them by typing

```
perl myscript
```

on the command line.

Scripts themselves are simple text files, so you can write them in just about any editor you like. And since Perl is interpreted, you don't need to do any compiling or linking. You just run the script when you've written it. Or, if you want to check that you haven't made any silly mistakes, such as forgetting to close quotes, or missing semi-colons off the end of lines, type

```
perl -c myscript
```

to have your script's syntax checked without being run.

How long is a piece of string?

So, Perl can be written easily without any special tools, it's free, and it is cross-platform. But when would you want to use it?

The most common use for the language these days is almost certainly for writing scripts on web servers – so much so that some servers, such as Apache, have Perl built in, saving time and system resources.

But that's not all you can do. As the box above shows, if you have batches of files that you need to process, transforming them from one form to another, it's an excellent tool to use – and is far more flexible than using macros in a word processor, for example.

Two of the keys to making the most of Perl are related to its handling of text

Top Perl resources

Websites
HTTP://perl.oreilly.com has a comprehensive list of O'Reilly publications for Perl, and links to other online resources.
www.activestate.com contains downloadable Perl for Win32 systems, plus a host of other tools.
www.macperl.com is a site that deals with Perl for the Macintosh.
www.perl.com is the official Perl home page, with links to downloadable code and lots more.
www.perl.org offers a ton of useful resources for Perl programmers everywhere.

Development systems
www.solutionsoft.com includes a downloadable trial version of PerlBuilder – a programming editor and debugger for Perl under Windows, which will simulate web-form processing to help develop scripts.

Books – the two bibles
Learning Perl, Randal Schwartz, O'Reilly and Associates, ISBN 1-56592-284-0
Programming Perl, Larry Wall & Randall Schwartz, O'Reilly and Associates, ISBN 1-56592-149-6

Perl in *PCW*
Hands On, *Web Development*, November 1999: Processing CGI input.
Hands on, *Internet*, July 1999: Using Perl to generate HTML pages from a database.
Hands On, *Internet*, June 1998: Putting a database on the web with Perl.

strings – associative arrays, and a powerful system for matching and replacing text using 'regular expressions'.

A regular expression is quite simply a way of expressing patterns using certain special characters, of which Perl has plenty.

If you have used the advanced search-and-replace features of MS Word, then you will be familiar with the concept, using things such as ^p to represent a new paragraph.

Perl actually goes way beyond that, allowing very complex searches which can include multiple alternatives, automatically saving parts of the search string if you need them.

For example, if you want to look for a line that begins with either the word 'hello' or 'goodbye', regardless of case, you could use the expression:

```
if ( $text =~ /^(hello|goodbye) (.*)/i ) { ... }
```

When the pattern matches, anything in brackets is automatically used to set the values of temporary variables $1, $2 and so on. So if the text was 'hello mother' then, as well as knowing that you had found a matching line, $1 would be 'hello' and $2 would be set to 'mother', so that you can use a single expression, and take action based on the text that was matched very easily.

The number of options for regular expressions is astonishing – with a bit of work, you can match just about anything you need; the manual pages for Perl give you all the details. Alongside regular expressions and the functions, such as substitution and searching that rely on them, associative arrays can be key to using Perl efficiently.

Put very simply, an associative array is an array that is automatically indexed, but instead of using a number, it uses a string. Which ultimately means that it is like a database.

If, for example, you want to store membership information in an array, you don't need to search through numeric indices to find someone's entry.

All you need to do is use an associative array, with their name as the index, so that you can refer to $memberinfo{'fred'} and access their information directly.

Using associative arrays can also be a boon when you're processing web forms, allowing you to use a simple piece of code for any form, thereby saving all the fields quickly and easily.

More to it than boy meets Perl
The key to using Perl is, in many cases, learning to master the tools it provides for dealing with strings. But there are many other things you can do; Perl 5 provides loadable modules that make handling CGI easy, or special functions to call the Windows 32 API directly.

On Unix, you can make connections directly to TCP/IP, allowing you to write things such as servers in Perl, or you can embed it in other languages such as C. There's even a PerlScript plug-in for web browsers, so you can use it as a scripting language on your web pages.

While we have only just touched on some of what makes Perl such a powerful tool here, there is a whole lot more that can be done with it.

If you are in any way familiar with other languages, you'll find that it is remarkably easy to start writing your first scripts, even with the most basic of commands as listed in our glossary on the opening page.

PERL – ITS SUPPORTERS BELIEVE IT'S THE MOTHER OF ALL PROGRAMMING LANGUAGES

If your appetite has been whetted, delve into the manual pages, look at the websites that we have listed – or you could even pick up a copy of *Learning Perl* (see box above).

After a while, you'll start to wonder how you ever managed without it.

PCW CONTACTS

Nigel Whitfield welcomes your feedback. Contact him via the PCW editorial office or email Internet@pcw.co.uk