Perl: Part 3

THINKING IN LINE NOISE

Having introduced the basic elements of Perl over the past two issues, Dean Wilson and Frank Booth now explain how to combine many of the elements shown previously into a complete program that you can run and tinker with

The sample application given here will show how Perl earned its monikers, the duct tape of the Internet and the Swiss army chainsaw. Hopefully it will also illustrate how you can replace automations currently done with a combination of *shell*, *sed* and *awk* with a small amount of Perl to give faster, more coherent solutions.

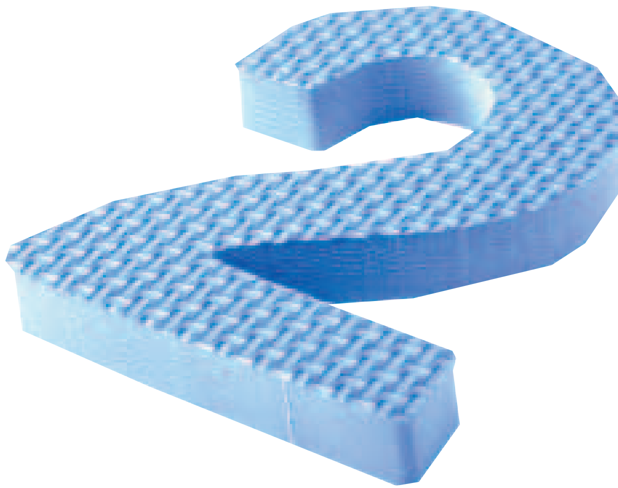
In addition to reinforcing old ground the example presented here also touches upon some aspects of Perl that we have not yet

covered; such as regular expressions. Rather than introduce each portion of the language in a piecemeal month-by-month fashion this approach enables you to start learning the language the right way: by using it. Over the coming months we will then focus on a more in-depth coverage of the new topics we introduce in this way.

The short application presented here as Example 1 is an example of a glue script: a Perl script that uses a standard command to do a lot of its work for it; maintaining simplicity in the Perl code, while bringing additional functionality to the command.

Example 1: count_logins.pl

```
01 # Sample script to count the number of logins a user has.
02 # Uses the 'who' command to get the user details.
03
04 #Location of the external binary.
05 my $whobin = "/usr/bin/who";
06
07 # Separates the command from its path
08 # and assigns the command name in $cmd.
09 my $cmd = (split("/", $whobin))[-1];
10
11 # Sanity check to ensure external dependencies are met.
12 die "No $cmd command found at '$whobin'\n" unless -e $whobin;
13 die "The $cmd command at '$whobin' is not executable\n" unless -x
$whobin;
14
15 my %usertally = getusers($whobin);
16
17 while (my ($user, $numlogins) = each %usertally) {
18   print "$user has $numlogins login", $numlogins > 1 ? "s" : "", "\n";
19 }
20
21 sub getusers {
22   my $whobin = shift;
23   my %user;
24
25   #Open a pipe to read response in from the 'who' command
26   open(WHO, "$whobin |") || die "Failed to open who: $!\n";
27
28   # loop over the output from who, assigning the line to $_
29   while (<WHO>) {
30     next if /^\s*$/; #Skip all empty lines
31     chomp;
32     m/(\w+)\s/;
33     $user{$1}++;
34   }
35
36   close WHO;
37   return %user;
38 }
```



An enhanced version of this command, which runs continually and notifies you of any changes in the number of logins on the system, can be found in the */Perl* directory on this month's CD with the name whoson.pl. This version has more functionality and makes a good demonstration of how to apply some of the theory we discuss in this article.

The *count_logins.pl* script, shown in Example 1, uses a surprising variety of Perl functionality considering its small size. The script is started with a short description of its intended functionality. Depending on whether you are coding for your own benefit or for a more public audience you could add more details such as a created and last modified date, an email address for author contact and any other short details that a user may need. For more comprehensive documentation (and you do write comprehensive documentation don't you?) you may be better served looking at POD, a subject we will cover in a future column.

Line 5 starts the actual code, we assign the full path of the external *who* command to *$whobin*. This is done both to avoid any path problems we may encounter if we assume the running user has a valid path set up and to allow us to do some checking on the state of the file at the given location.

In line 9 we try to establish the name of the binary we are calling so that we can tailor any error messages we emit to show which command the problem occurred with. When writing error messages a little extra work upfront can save hours of head scratching once you begin to create bigger applications. The *split* command takes the path and command name we assigned to *$whobin* and separates them based upon the first argument given. We then use a negative subscript (which works in the same way as the array subscripts in article one) to return the last item (the –1; which means count back one from the end) from the split, which is the command name and assign it to *$cmd*.

The sanity checks in lines 12 and 13 confirm that the file indicated by *$whocmd* is both present and executable. If either of these criteria fails then the program aborts with an error message detailing the problem.

Line 15 is where we encounter our custom subroutine, *getusers* which has its code in 21–38. We call the *getusers* subroutine with the location of the *who* command as its only argument and we assign its

a little extra work upfront can save hours

File handle refresher

In last month's article we dipped our toes in to Perl's file handling commands and showed how to open a file for reading and writing. Due to the large role file handles play in most programs, here is a brief recap on opening, writing to and closing a file handle:

```
open (HANDLE, '>afile') || die "$Failed to ↵
open HANDLE: $!\n";
print HANDLE "Hello\n";
close HANDLE;
```

The example opens the file "afile" in the current directory for output – clobbering the contents of any existing file of that name. We then use the common Perl idiom to test whether the open file operation is successful. If it fails then the *die* function is called, exiting the program and printing the error message given and setting the return code. In the string passed to the *die* function we also pass *$!* – another of Perl's internal variables. When used in string context *$!* reports the system error string related to the last command.

If the open was successful we carry on to the next line and then print the line "Hello\n" to afile. After printing the line the file is closed and the program exits. It is possible to check the return value of the closing of the file handle but in this example we gain nothing from it, as there is nothing we can check.

We then moved on and covered alternate ways of initialising the handle to allow us different ways of interacting with it:

* '<' – Read from a file.
* '>' – Over-write or create the file.
* '>>' – Create a file if none exists, append to a file if it does.

If no prefix is specified the default is '<'.

Introduction to IPC and piping

In the count_logins.pl script (in the main body of the article) we open a file handle as a pipe to an external system command. In order to understand how this works you will need a basic understanding of file handles, if you are unsure then please read the Filehandle Refresher boxout before continuing. Opening a pipe to an external command can be considered one of the more basic forms of Interprocess Communication. Interprocess Communication, or IPC as you'll often see it referred to, is a way that multiple processes can communicate with each other. This communication can range from merely knowing that an event has occurred, known as "signal handling" to sharing the output of one process with another on the same host, the same network or even across the Internet.

In the example script, count_logins.pl, the pipe is opened to the *who* command as shown below:

```
open(WHO, "$whobin |") || die "Failed to ➔
open who: $!\n";
```

The *who* command is executed and if it is successful then its output is available for reading from the *WHO* file handle in the same manner as if the handle referred to a plain text file. This simplicity in gathering the output of external commands is one of the main contributors to Perl's title of duct tape of the Internet. Taking this premise slightly further we can use the same syntax to set up whole pipelines of commands external to the Perl script. These chains eventually return the output of the last command in the chain. This behaviour fits in so well with the standard Unix ideal of filter chains that many people never advance beyond this form of IPC.

Now that we have shown how to read data in from an external command it seems fitting to show how easy it is to reverse this and send output from a Perl script to an external application:

```
open(PAGER, "| $pager") || die "Failed to ➔
open $pager: $!\n";
```

If you are curious as to Perl's other forms of IPC then 'perldoc perlipc' is a good place to start and Lincoln Stein's *Network Programming with Perl* is an excellent title that covers the subject in an unrivalled depth.

duct tape of the Internet

return value to the *%usertally* hash for our future use.

At this point we will make a leap to line 21 and have a closer look at what is happening in the subroutine that will give us the return value. We declare the subroutine with the *sub* keyword followed by the name of the subroutine and then an optional prototype. We will cover *sub* in a future article, or the impatient can take a look at *perldoc perlsub*. We then follow this with a curly brace to show we are starting the body of the sub.

In line 22 we assign the location of the 'who' command that we passed in to the subroutine in to the *$whobin* variable using the *shift* function. You may remember the *shift* function from our earlier encounter with it in article one when we used it to remove and return to us the first element of an array.

In Perl, subroutine arguments are accessed via the @_ array – an implicit variable and a relative of $_ – that you will be seeing more frequently from this point on. When you pass multiple scalars (variables with a single value and visually represented with a '$') to a subroutine each call to *shift* returns the next one in the array while removing it. This is one of the methods of iterating through subroutine arguments.

We then create the hash that is to hold the users on the system and the number of logins they currently have running before moving on to a variant of the file *open* we saw last month. Line 26 with the piped file *open* is explained in the Introduction to IPC boxout.

One of the important design considerations for programs that "pipe out", have other interactions or dependencies with external commands, or that utilise other forms of IPC is that of blocking. When you invoke an external command and attempt to read in its results your program will halt until the IPC or pipe

Linux signals and signal handlers

Events can happen at any point while a process runs: the operating system may terminate the application, file handle limits can be reached or the user may simply get tired of waiting and press Ctrl+C.

When one of these occurs a signal is sent to your program and it responds by taking an action such as immediately halting execution and exiting or rereading its configuration. To get a list of the signals Linux supports you can type *kill –l* at the command prompt.

While the default responses to signals can be enough to ensure that the program does the minimum of what's required, they can also cause it to exit in an incomplete state creating problems such as leaving temporary files on the machine or even just prevent it from logging the time the program stopped.

To get around some of these limitations you can write your own custom signal handlers to catch and process the signals as you see fit. Overriding signal handlers in Perl is simple, the %SIG hash can have references to user-defined signal handlers (subroutines by another name) that are called when Perl receives the corresponding catch-able signal.

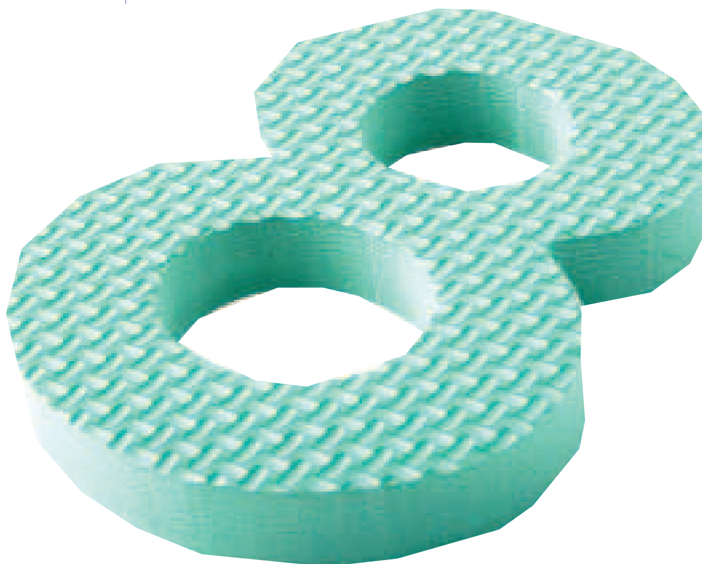
```
$SIG{INT} = sub { print "I got killed\n"; ➘
exit; };


while (1) {
  print "Still here\n";
  sleep 2;
}
```

In the above code snippet we enter an infinite loop that simply prints the same string until you get bored and press Ctrl+C. This then sends an *INT* signal to Perl, causing Perl to stop the section of the code it's currently executing and call the handler assigned to *INT*. While this example is slightly contrived, if you remove the *exit* in the handler the program does not terminate on a Ctrl+C and this is one example of how you could protect sections of the code that need to complete from being killed while running.

you can write your own custom signal handlers

has returned the data that is to be read. If not recognised and catered for this can have many negative effects on your program. A common example is the user prematurely terminating the script from the terminal, leaving any external resources it uses in an undermined state or interrupting the program in the middle of a series of actions that must either all be completed or none completed (this is known as atomic). A common way to deal with this in Perl on Linux is to use signal handlers to protect critical parts of your program and allow them to exit gracefully.



Another more advanced use of signals to help mitigate the problems of blocking is with the *alarm* function – full details of which can be found in *perldoc –f alarm*, and a custom signal handler but due to its advanced nature we will return to this at a future point when we cover different IPC mechanisms.

With line 29 we begin to get the data in from the *WHO* handle and process it to give us our totals. We set up a standard *while* loop which will iterate over the file handle assigning the line read in to *$_* until no more data is left. This was covered in article two if you need a refresher.

On line 30 we get our first view of a regular expression. A regular expression (regex) is a way of expressing a desired set of text to match using special meta-characters. While this explanation may seem less than enlightening regular expressions play such a large part in Perl we will cover them in great depth in a separate article. This regex is a simple one and has been placed just to whet your appetite.

Breaking down the regex we have a forward slash that indicates that anything between it and the next un-escaped forward slash is the target we wish to match. The ^ indicates that the regex should be checked from the start of the string and the *$* indicates the end of the string.

The \s is called a character class and represents the

different forms of whitespace (including spaces and tabs) while the asterisk is similar to a wildcard and means zero or more of the expression preceding it. Putting these together we end up with code that says "If the line from start to finish is empty or comprised only of whitespace then do not process this line and jump to the next.". While this example may not be crystal clear if you have no previous exposure to regexs hopefully it has shown how terse yet powerful they can be when used correctly.

Still operating on the implicit *$_* we remove the new line from the end of the string (Using the *chomp* on line 31) and then we work another bit of regex magic on line 32. This time we use a character class that represents words. A word in this context is a letter in the ranges of a-z or A-Z. A number in the range 0-9 or the underscore ('_'). We match as many word characters as we can from the start of the line up until the first piece of whitespace we encounter, using \s again, as described above. The parentheses are another regex meta-character, known as capturing or grouping depending on the context they are used in. They cause the value matched by the regex expression inside them to be assigned to one of the special regex match variables, in this case it gets assigned to *$1* as it is the first match so that we can use the text matched outside of the regular expression.

In line 33 we use a Perl idiom that you will see in the wild. If we matched a new user name on line 32 it will not yet be present in the *$user* hash. We then add the user to the *$user* hash and increment the number of times we have seen that user by one. In order to understand why this is successful you must remember that when an empty string is used in numeric context it is a zero, we then increment the zero by one and have the correct number of logins – one login. If the user is already in the *$user* hash the ++ ups by one the number of logins as expected. This is an oft-used idiom as it reduces a four line operation to one cleaner line of code.

We then act like responsible coders and close the *WHO* handle at line 36 – even knowing that implicit closes will occur it's good practice to close them manually. We then return the *%user* hash to line 15, where we called the subroutine, and where the values are assigned to *%usertally* and we finish the subroutine with the closing curly brace.

Jumping back up to line 17 we iterate through each of the key and value pairs in the hash with a *while* loop and in the body of the loop print out the user and the number of logins they had.

The last interesting example of Perl code in this small application is at line 18. When we list the number of logins the person had we want to say 'dwilson had 1 login' or 'dwilson had 2 logins' with the *s* added to the end of the string for anything more than a single login.

We do this by building a longer string from composite strings and including a ternary operator. The building of the string is achieved by passing a list of arguments to the *print* function with each argument separated with a comma. The ternary operator (also called the trinary operator in some Perl books) is in essence a shorthand if-then-else statement. It is actually an expression, so it can be added in places such as function calls where an *if* statement is not permitted.

A ternary maps out like this:

```
condition ? then : else

$numlogins > 1 ? "s" : ""
```

So if *$numlogins* is greater than one, meaning the condition is true, then the *then* part is called and an *s* is added to the string. If the condition evaluates to false the *else* part is called and, seeing as, in this case we not do wish to add anything, we return an empty string.

another bit of regex magic

In closing

Now that you have seen a complete, albeit, small example of a functional Perl script, the more abstract concepts that were covered in the first two articles should be better understood now. We have lightly touched upon subroutines and regular expressions and handed out some pointers on further reading for those eager to move ahead before we come back to them in the near future.

Source code

The source code for the examples used in this article can be found on this month's cover CD in the */Perl* directory.