Perl: Part 2

THINKING IN LINE NOISE

One of Perl's tenets is to make simple tasks easy. With the initial hurdle of Perl's extensive use of sigils behind us, Dean Wilson and Frank Booth move on to more practical aspects of the language; namely conditions, looping and files

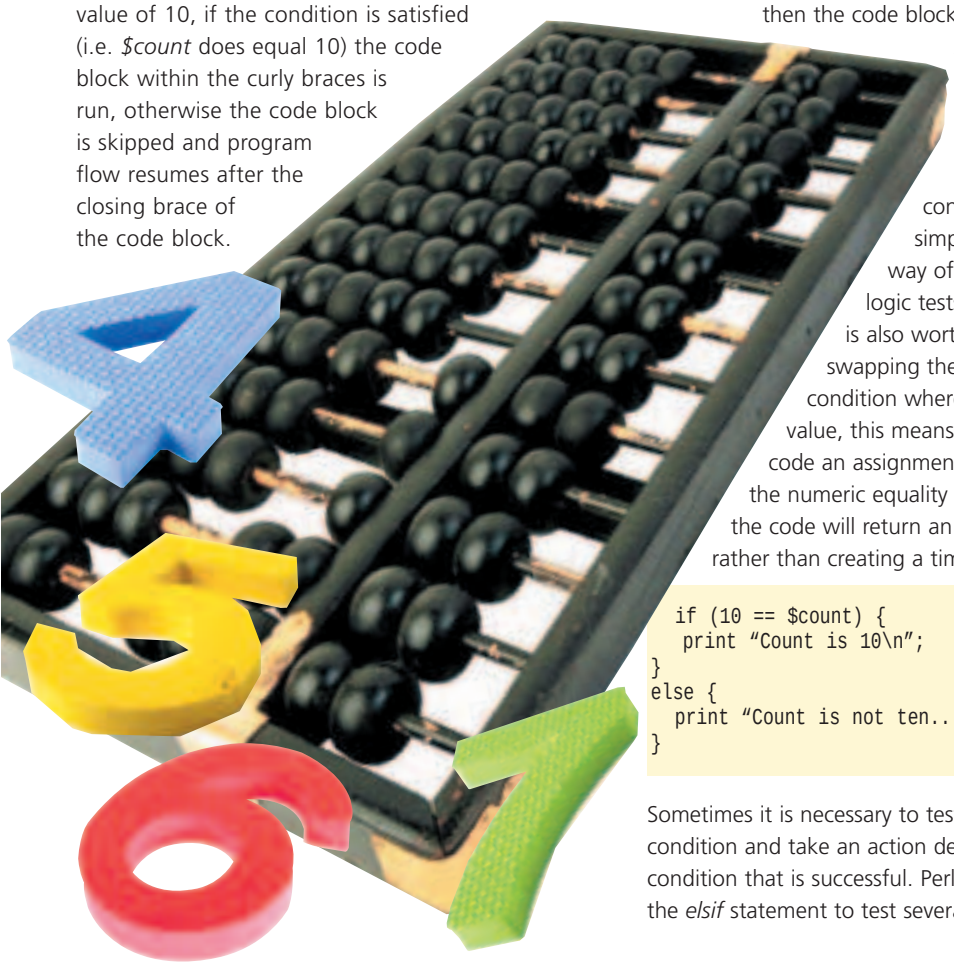
Not as loopy as it sounds

All programming languages use flow control statements to influence the execution of code. Flow control statements come in two varieties, 'conditional' and 'loops'; it may come as no surprise that Perl has an abundance of each.

The simplest of the conditions is the *if* statement, which evaluates conditions and executes a branch of code dependent on the result.

```
if ($count == 10) {
  print "Count is 10\n";
}
```

In this example we test to see if '$count' has a value of 10, if the condition is satisfied (i.e. *$count* does equal 10) the code block within the curly braces is run, otherwise the code block is skipped and program flow resumes after the closing brace of the code block.



It is sometimes desirable to execute an alternative code block if the condition is not met and for this we use *else*:

```
if ($count != 10) {
  print "Count is not ten...\n";
}
else {
  print "Count is 10\n";
}
```

The above example tests the value of *$count*, if *$count* is not 10, then the message "Count is not ten..." is printed and program control is returned AFTER the last curly brace. If the condition is not met then the code block defined after the *else* statement is executed, and the message "Count is 10" is printed.

Making a condition statement simple to read is the best way of ensuring that the logic tests work as expected. It is also worth considering swapping the values in a test condition where one is an immutable value, this means if you mistakenly code an assignment operator rather than the numeric equality test (= instead of ==) the code will return an error at compile time, rather than creating a time consuming bug.

```
if (10 == $count) {
  print "Count is 10\n";
}
else {
  print "Count is not ten...\n";
}
```

Sometimes it is necessary to test for more than one condition and take an action dependent upon the condition that is successful. Perl permits this by using the *elsif* statement to test several operations

Don't worry, it's only there to keep the 'C' programmers happy

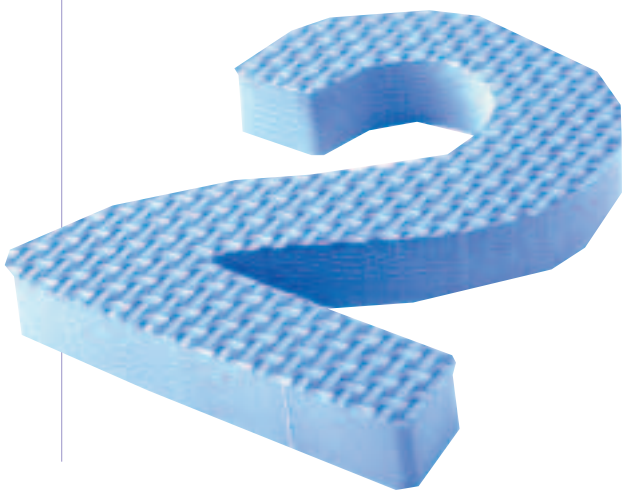
sequentially, note as soon as one set of conditions are satisfied only the code block belonging to that condition will be executed and the remaining conditions will not be evaluated. *else* can be added at the end of *if* and *elsif* statements as a catch-all that is executed when no other conditions are met.

```
$traffic_light = 'blue';

if ('red' == $traffic_light) {
  print "Stop!\n";
}
elsif ('amber' == $traffic_light) {
  print "Lights about to go to red\n";
}
elsif ('flashing amber' == $traffic_light) {
  print "Proceed with caution\n";
}
elsif ('green' == $traffic_light) {
  print "Go\n";
}
else {
 print "traffic lights are not
$traffic_light\n";
}
```

Boolean tests can sometimes become difficult to understand if they contain negatives. Perl provides the condition operator *unless* to help reduce the confusion. Compare these statements:

```
if ( !exists($valid{$user}) ) {
  print "Hey, you aren\'t allowed here!\n";
}

# The same thing but easier to read
unless ( exists($valid{$user}) ) {
print "Hey, you aren't allowed here!\n";
```

The difference between the previous two conditions may be slight as we've not yet encountered compound conditions, which depend on the outcome of a sequence of conditions. Each condition is connected by either an *&&* or the ll. If an event depends on two conditions being met we could use *&&*, in the example below I've used brackets to force the precedence in which



expressions are evaluated, it's not necessary, but it is good practice.

```
if ( ( exists($tb2{Virgil}) ) &&
     ( $cliff_door eq 'open') ) {
  print "Lift off!\n";
}

if ( ( exists($music{'ominous shaky violins'})
) ||
     ( $diary eq 'Tracy family holiday') ) {
  print "Warning, impending catastrophe with
few casualties\n";
}
```

In situations where the *if/unless* statement is used to perform a single task, and there is no *else* or *elsif* used, the code can be rewritten thus:

```
print "$count is not 10\n" unless $count == 10;
```

Note the brackets are not required around the condition, and that there are no curly braces around the code to be evaluated. This will only work for code that executes one line of code and does not use *else* or *elsif* (yes *unless* can use *else* and *elsif* too).

Looping structures enable programmers to eliminate a lot of the repeated functionality within a program. The most common looping structure is the *for* loop, which comes in many guises.

```
for ($i=0; $i < 5; $i++) {
  print "The standard C like 'for'
declaration\n";
}

@array = (1,2,3,4);

foreach $count (@array) {
  print "$count: To iterate over each item in
an array\n";
}

for $count (1..4) {
  print "$count: To iterate over a range
operator\n";
}

for (1..4) {
  print "$_: use \$_, the default variable\n";
}

print "$_: use \$_, the default variable\n" for
1..4;
```

Each of the above examples is a *for* loop, which will repeat four times. It is a matter of preference which method you prefer.

The first method is possibly obscure, if you are new to programming. Don't worry, it's only there to keep C programmers happy. The loop declaration is split into three parts, any of which may be empty. The first section initialises the variables for the loop – this is on the first pass only so it is traditionally where variables

are set to their start value. After the first semi-colon is the condition section, which holds the conditions required to exit the loop. This is tested before each iteration: if this was empty the loop would be infinite, standard condition operators are used. Lastly the modifier section is evaluated after the condition section is failed.

Using an array in a *for* or *foreach* loop (the two functions are interchangeable) is easy, just put the array in parentheses and the loop will iterate over every element of the array. A variable can be specified to hold the current value from the array, or the default variable ($_) can be used implicitly.

The last example shows that as with the *if* and *unless* statements, the syntax of the *for* statement can be reversed if only a single action is desired. The rules for dropping the parenthesis and curly braces are the same as for the *if/unless* given above.

The last commonly encountered loop is the *while* loop. The *while* loop executes until its conditional test is no longer met.

```
$count = 0;

while ( $count++ < 4 ) {
  print "$_: again using a default variable\n";
}

# This code won't run as $count is 0 which is
'false'
$count = 0;

while ( $count ) {
  print "$_: again using a default variable\n";
}

$count = 3;
print "$_: again using a default variable\n"
while $count--;
```

There are additional, less used, control statements that will be introduced when their unique properties make them applicable.

The truth is rarely pure and never simple

Condition statements evaluate whatever they're given, that is the contents of the brackets: variables, strings, numbers and functions. Throughout the discussion of conditionals you may have noticed the references to truth and wondered what was considered to be 'true' in Perl. To illustrate the value of truth we will use comparison operators and some examples.

Comparison operators are used to determine the outcome of conditions. These consist of two types of comparison: 'relational operators' and 'equality operators', both compare two values and return an outcome. The outcome of a comparison operation is called 'true' or 'false'.

In reality, as with most things, Perl has a wealth of values to represent 'true' or 'false'. Most things are true with the exception of '',undef, \0, 0 and eof (end of file).

In the example below it can be seen that the order for equality operations is irrelevant. Put simply it either is or isn't equal.

```
$value = 4;

# Equality operators
print "the value is 4\n" if $value == 4:
print "the value is 4\n" if 4 == $value;
```

For relational operations the value to the left of the operator is the subject of the comparison, and whether it is true or false depends upon its comparison against the right-most value (comparator).

```
# Relational operators
print "the value is greater than 3\n" if $value
> 3;

# This will not print as 3 is less than 4
print "the value is less than 3\n" if $value <
3;
```

Comparison operators are used to determine the outcome of conditions

String-wise comparisons

These are the most commonly used comparison operators. The first operator on each line is used for comparing strings; the second for numeric values.

Equality operators

eq, ==	True if the values are equal
ne, !=	True if the variables don't match

Relational Operators

lt, <	True if the value is less than the comparator
le, <=	True if the value is less than or equal to the comparator
gt, >	True if the value is greater than the comparator
ge, >=	True if the value is greater than or equal to the comparator

A simple aide-memoire is: string comparisons consist of letters.

It's vital to ensure the right type of comparison is used to avoid introducing subtle bugs

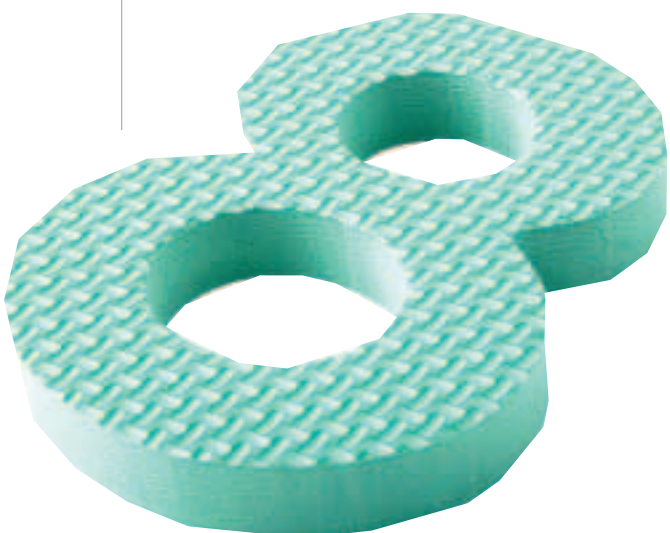
Different operators are used to compare strings and numbers. It's vital to ensure the right type of comparison is used to avoid introducing subtle bugs. When run, the code below shows one way in which the wrong comparison may used.

```
$count = '1';
$result ='';

if ( $count == 1.0 ) {
  $result ='It is';
}
else {
  $result ='It is not';
}
print "$result the same as the number 1.0\n";

if ( $count eq '1.0' ) {
  $result ='It is';
}
else {
  $result ='It is not';
}
print "$result the same as the string 1.0\n";
```

Numeric comparisons compare the numeric part of the value to the numeric part of the comparator. It is perfectly valid to compare the numeric parts of values in this fashion.

```
  print "There'll be no green bottles\n" if
$count < '1 Green Bottle';
```

String comparisons act upon the entire value and use a comparison method called **string-wise comparison**. String-wise comparisons use the ASCII representation for a character to determine whether it is greater or lesser than the comparator. Any character of a lower ASCII value than its comparator is said to be greater than the comparator in a string-wise comparison. String comparisons don't have magnitudes as numbers do, so 'byzantine' is less than 'Roman', this is because upper case characters occur earlier in ASCII representation.



IO, IO it's off to work we go

In Unix everything is a file so file handling is a very important part of any language using the Unix platform. Perl has one of the most comprehensive sets of file manipulation commands. The most common method of interacting with files is by opening a file and retaining what is called a 'handle' to the file. A file handle is a way of referring to a file when you wish to read or write to it.

```
open (HANDLE, '>afile');

print HANDLE "Hello\n";

close HANDLE;
```

This example opens the file *afile* in the current directory for output (clobbering the existing file of that name) it then prints the line "Hello\n" to the file *afile*. After printing the line the file is closed and the program exits. While this is a simple example these basic principles hold true for most of Perl's commonly used IO functionality.

In the previous example we opened the file so we could write out to it. The way we plan to use a file is used is determined by the redirection prefix before its name.

'<'	Read from a file. This is the default if no prefix is given.
'>'	Over-write or create the file.
'>'	Create a file if none exists, append to a file if it does.

Printing a line to a file is achieved by using the *print* function and specifying the file-handle to be used. The default file-handle is the standard output (*STDOUT*), usually the screen. The example below illustrates this:

```
print STDOUT "I'm on the big screen!\n";
print "I'm on the big screen too!\n";
```

When the program has finished executing the file-handle is implicitly closed. However it's good practice to close file-handles when you are done with them in case the file-handle name is reused in the same program.

File tests

Common file tests include:

–T	True if file is a text file
–e	True if file exists
–d	True if file is directory
–r	True if file is readable
–w	True if file is writable.
–x	True if file is executable.

In the following example we read the contents of a file using a *while* loop to read every line of the file. For every line it prints the line number and the line.

```
open ( I, '<file' ) || die "file: $!\n";
while (<I>) {
  print "$. : $_";
}
close I;
```

The first line of the example uses a common Perl idiom to test whether the open file operation is successful. If it fails then the *die* function is called, exiting the program with an error message defined in the same fashion as a *print* function and setting the script's 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.

The *$.* is yet another of Perl's internal variables and it contains the current line number of the file being read. When we concatenate *$.* and *$_* in the *print* statement we iterate over every line in the file and print both the line number and the contents of the line to the screen.

Before you open a file you often wish to determine some of the attributes of the file (often you want to know if it already exists) and to aid you in this Perl offers a large selection of **file tests** (Which are listed fully in *perldoc perlfunc*).

```
if (-T $file) {
  open ( I, '<file' ) || die "file: $!\n";
  while (<I>) {
    print "$. : $_";
  }
  close I;
}
```

The previous example has been modified so that we only enumerate lines in a file if it is a text file; printing binary files to screen is seldom rewarding.

Now that we have covered files and determining their type, let's move on to a related topic, directories. In essence, working with directories is similar to opening files:

```
opendir(DIRHANDLE, $dirpath) || die "Failed to
open current directory: $!\n";
@Files = readdir DIRHANDLE;
closedir DIRHANDLE;
```

We start by calling *opendir* with the name of the directory handle we want to retain and the directory we want to open, as with the file operation *open*, *opendir* returns true if the directory was opened successfully. The array *@Files* is populated with the name of each file in *$dirpath* by calling the function *readdir*. Filenames in *@Files* do not have a full path, only the filename itself. Finally, the directory handle is closed. As with closing file handles, this is done implicitly at the end of the script.

This example is a more practical demonstration of reading directories:

```
opendir(CURRENT, ".")
  || die "Failed to open current directory:
$!\n";

@Files = readdir CURRENT;
closedir CURRENT;

foreach $filename (@Files) {
  $linecount = 0;
   if (-T $filename) {
    open(TEXTFILE, "$filename") || die "Failed
to open: $!\n";
    while(<TEXTFILE>) {
      $linecount++;
    }
    close TEXTFILE;
    print "$filename has $linecount lines.\n";
  }
}
```

The example above illustrates all the principles in this section. We start by trying to open the current directory and assign it to the handle 'CURRENT', if this fails we exit the program and display the reason for failure. If everything is fine @Files is populated with the name of each file in the directory and the directory handle is closed. The *foreach* loop is the main body of the program, it iterates through the array and each text-file (*–T $filename* determines this) the file is opened the number of lines it contains counted before closing the file and printing the total. The line count is not reported if the file is not text or it can't be opened. If the file cannot be opened we print the failure message and exit the program. ■

Before you open a file you often wish to determine some of the file attributes