Perl 101 (Part 3) − Looping The Loop

By Vikram Vaswani and Harish Kamath

Table of Contents

Shampoo And Perl

Last time, we introduced you to Perl's basic control structure − the "if−else" family of conditional statements − and also taught you a little more about scalar variables. This week, we're going to proceed a little further down that same road, with a look at the different types of loops you can use in Perl, an introduction to a new type of variable, and a tongue−in−cheek look at the things modern managers do in the interests of a fatter bottom line.

As always, we'll begin with a definition for those of you coming at this series from a non−programming background. In geek−talk, a "loop" is precisely what you would think − a programming construct that allows you to execute a set of statements over and over again, until a pre−defined condition is met.

Every programming language worth its salt uses loops − and, incidentally, so do quite a few shampoo manufacturers. Think lather−rinse−repeat, and you'll see what we mean...

While You Were Sleeping...

The most basic loop available in Perl is the "while" loop, and it looks like this:

```
while (condition) { do this! }
```

Or, to make the concept clearer,

```
while (rich Uncle Ed's still breathing) { be nice to him }
```

The "condition" here is a standard Perl conditional expression, which evaluates to either true or false. So, were we to write the above example in Perl, it would look like this:

```
while (rich_old_guy_lives == 1) { be_nice(); }
```

How about an example you could actually use in your real−life job? Well, here's one − be warned, however, that it's primarily meant for managers who work in the Dilbert Zone.

```
#!/usr/bin/perl # weighted employee evaluation program # call
me WEEP! # ask the question print ("Are you satisfied with
your salary? [y/n] "); # get an answer $reply = <STDIN>;
chomp($reply);
# keep asking until you get the reply you want while($reply ne
'y') { print ("Are you satisfied with your salary? [y/n] ");
$reply = <STDIN>; chomp($reply); } print ("Employee
satisfaction has always been this company's goal.\n"); print
("Thank you for making this experience an enriching one!\n");
```

And here's what it looks like:

```
Are you satisfied with your salary? [y/n] n Are you satisfied
with your salary? [y/n] n Are you satisfied with your salary?
[y/n] n Are you satisfied with your salary? [y/n] n Are you
satisfied with your salary? [y/n] y Employee satisfaction has
always been this company's goal. Thank you for making this
experience an enriching one!
```

Let's take a closer look at this code. The first part of the program should be familiar to you by now − we're simply asking for input, assigning it to a variable, and then using the chomp() function to remove the trailing carriage return.

At this point, we begin checking the value of the variable − *while* this value is *not equal* to "y", or an affirmative reply, we continue printing the question over and over again, and stop only when the value

becomes equal to "y". At this point, the lines following the loop are executed, and the appropriate output displayed.

The end result? A company full of "satisfied" employees [whoever says Perl isn't powerful should take a look at that HR manager's Christmas bonus...]

Here's another example, this one using a "while" loop and a conditional expression that contains a number instead of a string:

```
#!/usr/bin/perl # factorials # initialize a variable
$factorial = 1; # ask for a number. print ("Gimme a
number!\n"); # process it $number = <STDIN>; chomp($number); #
assign the number to a "temp" variable $tmpnumber = $number; #
calculate the factorial while($number != 1) { $factorial =
$factorial * $number; $number−−; } print ("The factorial of
$tmpnumber is $factorial.\n");
```

In case you flunked math class, the factorial of a number X is the product of all the numbers between 1 and X. And here's what the output looks like:

```
Gimme a number! 7 The factorial of 7 is 5040.
```

And if you have a calculator handy, you'll see that

7*6*5*4*3*2*1 = 5040

Once the user enters a number, a "while" loop is used to calculate the product of that number and the scalar variable $factorial [initialized to 1] − this value is stored in the variable $factorial. Next, the number is reduced by 1, and the process is repeated, until the number becomes equal to 1. At this stage, the value of $factorial is printed.

...Or Until You Wake Up

The not−so−distant cousin of Perl's "while" loop is its "until" loop, which looks like this:

```
until (condition) { do this! }
```

To illustrate the relationship between the "while" and "until" loops, read these two sentences:

WHILE you're less than twenty−one years of age, you can't drink!

UNTIL you're over twenty−one years of age, you can't drink!

In other words, the conditional expression to be evaluated in a "while" loop will be exactly the opposite of the one to be evaluated in an "until" loop. So take another look at the WEEP, which we've rewritten using an "until" statement:

```
#!/usr/bin/perl # weighted employee evaluation program # call
me WEEP! # ask the question print ("Are you satisfied with
your salary? [y/n] "); # get an answer $reply = <STDIN>;
chomp($reply); # keep asking until you get the reply you want
until($reply eq 'y') { print ("Are you satisfied with your
salary? [y/n] "); $reply = <STDIN>; chomp($reply); } print
("Employee satisfaction has always been this company's
goal.\n"); print ("Thank you for making this experience an
enriching one!\n");
```

If you compare the "while" and "until" lines in the WEEP examples above, you'll see that the two conditional expressions are exactly the opposite of each other. And if you can't decide which one to use, try this little trick − translate your "until" or "while" statement into English, roll it around your tongue, and see if it sounds right to you...

Dos And Don'ts

The two control structures explained above test the conditional expression first, and only proceed to execute the statements within the loop if the expression evaluates to true. However, there are often occasions when you need to execute a particular set of statements at least once before you check for a valid conditional expression.

Perl has a solution to this problem too − it offers the "do−while" and "do−until" constructs, which allow you to execute a series of statements at least once before checking the conditional expression. Take a look:

```
do { do this! } while (condition)
```

or

```
do { do this! } until (condition)
```

In this case, Perl will only test for the condition *after* executing the loop once.

```
#!/usr/bin/perl # weighted employee evaluation program version
2.0 # call me WEEP II! # use DO to ensure that the statements
# within the loop are executed at least once do { print ("Can
we put you in a cubicle, cancel all your benefits, and pay you
less than minimum wage? [y/n] "); $reply = <STDIN>;
chomp($reply); } while($reply ne 'y'); print ("Heh heh! This
company couldn't do without employees like you!\n");
```

As you can see, the "do" construct can help to make your code more compact − compare WEEP version 2.0 with the original above.

For Pete's Sake!

Both "while" and "until" are typically used when you don't know for certain how many times the program should loop − in the examples above, for example, the program continues to loop until the user enters the right answer. But Perl also comes with a mechanism for executing a set of statements a specific number of times − and it's called the "for" loop:

```
for (initial value of counter; condition; update counter) { do
this! }
```

Doesn't make any sense? Well, the "counter" here refers to a scalar variable that is initialized to a specific numeric value [usually 0 or 1]; this counter is used to keep track of the number of times the loop has been executed.

Each time the loop is executed, the "condition" is tested for validity. If it's found to be valid, the loop continues to execute and the value of the counter is updated appropriately; if not, the loop is terminated and the statements following it are executed.

Take a look at this simple example of how the "for" loop can be used:

```
#!/usr/bin/perl for ($a=5;$a<12;$a++) { print("It's now $a PM.
Too early!\n"); } print("Let's party!\n");
```

Here's what the output looks like:

```
It's now 5 PM. Too early! It's now 6 PM. Too early! It's now 7
PM. Too early! It's now 8 PM. Too early! It's now 9 PM. Too
early! It's now 10 PM. Too early! It's now 11 PM. Too early!
Let's party!
```

How does this work? We've begun by initializing the variable $a to 5. Each time the loop is executed, it checks whether or not $a is less than 12; if it is, a line of output is printed and the value of $a is increased by 1 − that's where the $a++ comes in. Once the value of $a reaches 12, the loop is terminated and the line following it is executed.

And, for something slightly more complex, take a look at our re−write of the factorial calculator above:

```
#!/usr/bin/perl # factorials version 2.0 # ask for a number.
print ("Gimme a number!\n"); # process it $number = <STDIN>;
chomp($number); # use the FOR loop to calculate the factorial
# note how we've initialized variables within the # loop
itself − you can do this too! for ($factorial=1,$counter =
$number; $counter > 1; $counter−−) { $factorial = $factorial *
$counter; } print ("The factorial of $number is
$factorial.\n");
```

Every Comedian Needs An Exit

The "for" loop also comes with a bunch of control statements, which can be used to modify its behaviour. We've listed the important ones below, together with examples:

next:

The "next" statement allows you to jump to the next iteration of the loop without executing the remaining statements of the current iteration. Consider this:

```
#!/usr/bin/perl for ($counter=1; $counter<=10; $counter++) {
if($counter == 9) { next; } print("$counter "); } print
("\nHey! Where did 9 go?\n"); print ("Simple. 7 8(ate) 9\n");
print ("Hey, if we were professional comedians, we wouldn't be
here, right?!\n");
```

Here's what it looks like:

```
1 2 3 4 5 6 7 8 10 Hey! Where did 9 go? Simple. 7 8(ate) 9
Hey, if we were professional comedians, we wouldn't be here,
right?!
```

As you can see, 9 is missing − this is because when the value of $counter hits 9, Perl uses the "next" statement to skip to the next iteration of the loop, and so 9 never gets printed.

last:

The "last" statement is used to exit the loop completely. Take a look:

```
#!/usr/bin/perl for($counter=1; $counter<=10; $counter++) {
if($counter == 9) { last; } print("$counter "); }
```

And here's the output:

```
1 2 3 4 5 6 7 8
```

redo:

And finally, the redo statement lets you restart a particular iteration of the loop:

```
#!/usr/bin/perl for($counter=1; $counter<=10; $counter++) {
print("$counter "); if($counter == 9 &$flag != 1) { $flag=1;
redo; } }
```

In this case, here's what you'll see:

```
1 2 3 4 5 6 7 8 9 9 10
```

Grade School

Before we go on to our last − also known as final − control structure of the day, we're going to take a quick detour and introduce you to a different kind of variable.

As you already know, Perl comes with scalar variables, which can be used to store a single value. But what you may not know is that it also comes with a mechanism to store multiple values in a single variable. This variable is known as an "array variable", and it is a useful method of storing and representing related information.

All the standard rules which apply when naming scalar variables also apply in the case of array variables, with one important exception − where a scalar variable name is preceded by a dollar [$] sign, an array variable name is preceded by an @ symbol.

Note also that Perl does not place any restrictions on array and scalar variables sharing the same name in Perl − so the array @stuff is different from the scalar $stuff.

Now, let's say that you wanted to create an array containing the names of your friends:

```
@friends = ("Rachel", "Monica", "Phoebe", "Chandler", "Joey",
"Ross");
```

Thus the array variable @friends contains six elements.

Each element of the array is a scalar variable, and can be accessed using scalar notation, with a suffix denoting the element's position in the array. So, in order to extract the first element of the array @friends, you'd use

```
$friends[0]
```

while

```
$friends[5]
```

would give you the sixth element of the array, Ross.

Similarly, if you wanted to modify a particular element of the array, you could use the scalar notation above to accomplish your task, like this:

```
$friends[3] = "Janice";
```

Your array would now contain

```
("Rachel", "Monica", "Phoebe", "Janice", "Joey", "Ross");
```

Note that the first element of an array is always referred to by the index 0 − this concept, known as "zero−based indexing" often confuses novice programmers, and is just one more reason why geeks have so few friends.

An array can contain both string and numeric data − for example, this is perfectly valid:

```
@mix = ("hello", 34747, 3, "bonjour");
```

How about a quick example to illustrate how data can be stored in a single array variable:

```
#!/usr/bin/perl # this example demonstrates how a single #
array variable can hold a student's gradea # in six subjects #
set up some variables @subjectlist = ("Math", "Lit.",
"Physics", "Biology"); @gradelist = (); $total = 0; # get and
store input for($x=0; $x<4; $x++) { print("What was your grade
in $subjectlist[$x]? "); $gradelist[$x] = <STDIN>;
chomp($gradelist[$x]); } # display it in neatly formatted rows
print ("SUBJECT\t\tGRADE\n"); for($y=0; $y<4; $y++) { print
("$subjectlist[$y]\t\t$gradelist[$y]\n"); $total +=
$gradelist[$y]; } # print a total print ("TOTAL: $total\n");
```

And here's what it looks like:

```
What was your grade in Math? 10 What was your grade in Lit.?
20 What was your grade in Physics? 30 What was your grade in
Biology? 40 SUBJECT GRADE Math 10 Lit. 20 Physics 30 Biology
40 TOTAL: 100
```

Let's walk you through this: we've begun by initializing two array variables and one scalar variable. The array @subjectlist contains a list of the subjects to be graded, and the array @gradelist will be populated by the user with the actual grades. The scalar variable $total will be used to display a total figure at the end.

Next, we've used a "for" loop to display a question, and assign the user's input to the @gradelist array in the appropriate slot via the $x counter. Once all four subjects are taken care of [note the conditional expression in the "for" loop, which automatically stops looping when the counter reaches 4], we've simply used the print() function and a second "for" loop to display a neatly−tabulated row of grades and subjects.

The second loop also adds each grade to the variable $total, and this displays this total value at the end of the program.

A couple of other points to note:

* The \t character used in the print() statements above is used to generate a single "tab" space.

* The notation

$total += $gradelist[$y];

is simply a Perl shortcut for

$total = $total + $gradelist[$y];

Playing With Friends

Now that you've got the basics of array variables down, it's time for us to introduce the "foreach" loop, one of the most useful control structures in Perl for dealing with array variables. It looks like this:

```
foreach some_scalar_variable (some_array_variable) { do this!
}
```

Here's an example:

```
#!/usr/bin/perl @friends = ("Rachel", "Monica", "Phoebe",
"Chandler", "Joey", "Ross"); foreach $item (@friends) {
print("$item is a true Friend!\n"); }
```

And the output is:

```
Rachel is a true Friend! Monica is a true Friend! Phoebe is a
true Friend! Chandler is a true Friend! Joey is a true Friend!
Ross is a true Friend!
```

The "foreach" loop works its way through the elements of an array, assigning each element to the defined scalar variable, and then executing the statements within the curly braces. In the example above, each subsequent iteration of the loop sees a new value being assigned to the scalar variable $item − this continues as many times as there are elements in the array. Once all the array elements are exhausted, the statements following the loop are executed.

An important point to note here is that the scalar variable used in the "foreach" loop is only assigned a value temporarily − once the loop has finished executing, the original value of the scalar variable, if any, is restored. Take a look:

```
#!/usr/bin/perl @friends = ("Rachel", "Monica", "Phoebe",
"Chandler", "Joey", "Ross"); $item = "Superman"; foreach $item
(@friends) { print("$item is a true Friend!\n"); } print
("Loop done!\n"); print ('The value of $item is now ', $item);
```

Here's what you'll get

```
Rachel is a true Friend! Monica is a true Friend! Phoebe is a
true Friend! Chandler is a true Friend! Joey is a true Friend!
Ross is a true Friend! Loop done! The value of $item is now
Superman
```

So Many Choices...

Now that you know all about Perl's control structures, you're probably wondering which one to use when. Well, this section will try to clear that dilemma up for you.

* If you have a set of statements that need to be executed a number of times until a certain condition is met, but you have no idea what that number is, the "while" and "until" loops are a good bet.

* If you have a set of statements that need to be executed at least once, the "do−while" and "do−until" loops are the ones to go with.

* If you need to execute a set of statements a specific number of times, pick the "for" loop and don't think twice!

* If you need to process every element of an array, go with the "foreach" loop.

Next time, we'll be teaching you a little more about array variables, and some of Perl's more interesting array functions; we'll also take a look at Perl's file manipulation capabilities.

What will you do until then? Well, they're bound to be showing re−runs of "Friends" on your local cable channel...