Perl 101 (Part 4) − Mind Games

By Vikram Vaswani and Harish Kamath

Table of Contents

The Lotus Position

So you've successfully navigated the treacherous bends of the "for" and "while" loops. You've gone to bed mumbling "increment $sheep" to yourself. You know the difference between "while" and "until", and you've even figured out which loop to use when. And, quite frankly, you're tired. You need a break from this stuff.

We hear you.

Relax. Close your eyes. Take a deep breath. Assume the lotus position. And get ready to explore your mind...

Handle With Care

Like all widely−used programming languages, Perl has the very useful ability to read data from, and write data to, files on your system. It accomplishes this via "file handles" − a programming construct that allows Perl scripts to communicate with data structures like files, directories and other Perl scripts.

Although you might not have known it, you've actually already encountered file handles before. Does this look familiar?

```
#!/usr/bin/perl # ask a question... print "Gimme a number! ";
# get an answer... $number = <STDIN>; # process the answer...
chomp($number); $square = $number * $number; # display the
result print "The square of $number is $square\n";
```

If you remember, we told you that the <STDIN> above refers to STanDard INput, a pre−defined file handler that allows you access information entered by the user. And just as <STDIN> is a file handler for user input, Perl allows you to create file handles for other files on your system, and read data from those files in a manner very similar to that used above.

For our first example, let's assume that we have a text file called "thoughts.txt", containing the following random thoughts:

```
We're running out of space on planet Earth. Scientists are
attempting to colonize Mars. I have a huge amount of empty
real estate in my mind. Imagine if I could rent it to the
citizens of Earth for a nominal monthly fee. Would I be rich?
Or just crazy?
```

Now, in order to read this data into a Perl program, we need to open the file and assign it a file handle − we can then interact with the data via the file handle.

```
#!/usr/bin/perl # open file and define a handle for it
open(MIND,"thoughts.txt"); # print data from handle print
<MIND>; # close file when done close(MIND); # display message
when done print "Done!\n";
```

And when you run this script, Perl should return the contents of the file "thoughts.txt", with a message at the end.

A quick explanation: in order to read data from an external file, Perl requires you to define a file handle for it with the open() function. We've done this in the very first line of our script.

```
open(MIND,"thoughts.txt");
```

You can specify a full path to the file as well:

```
open(MIND,"/home/user1/thoughts.txt");
```

In this case, MIND is the name of the file handle, and "thoughts.txt" is the text file being referenced. The file will then be read into the file handle <MIND>, which we can use in much the same manner as we would a variable. In the example above, we've simply printed the contents of the handle back out with the print() function.

Once you're done with the file, it's always a good idea to close() it − although this is not always necessary, it's a good habit!

Different Strokes

There's also another method of reading data from a file − a loop that will run through the file, printing one line after another:

```
#!/usr/bin/perl # open file and define a handle for it
open(MIND,"thoughts.txt"); # assign the first line to a
variable $line = <MIND>; # use a loop to keep reading the file
# until it reaches the end while ($line ne "") { print $line;
$line = <MIND>; } # close file when done close(MIND); #
display message when done print "Done!\n";
```

Well, it works − but how about making it a little more efficient? Instead of reading a file line by line, Perl also allows you to suck the entire thing straight into your program via an array variable − much faster, and definitely more likely to impress the girls!

Here's how:

```
#!/usr/bin/perl #pen file and define a handle for it
open(MIND,"thoughts.txt"); # suck the file into an array @file
= <MIND>; # close file when done close(MIND); # use a loop to
keep reading the file # until it reaches the end foreach $line
(@file) { print $line; } # display message when done print
"Done!\n";
```

As you can see, we've assigned the contents of the file "thoughts.txt" to the array variable @file via the file handle. Each element of the array variable now corresponds to a single line from the file. Once this has been done, it's a simple matter to run through the array and display its contents with the "foreach" loop.

A Little Brainwashing

Obviously, reading a file is no great shakes − but how about writing to a file? Well, our next program does just that:

```
#!/usr/bin/perl # open file for writing
open(BRAINWASH,">wash.txt"); # print some data to it print
BRAINWASH "You will leave your home and family, sign over all
your money to me, and come to live with forty−six other slaves
in a tent until I decide otherwise. You will obey my every
whim. You will address me as The Great One, or informally as
Your Greatness.\n"; # close file when done close (BRAINWASH);
```

Now, once you run this script, it should create a file named "wash.txt", which contains the text above.

```
$ cat wash.txt You will leave your home and family, sign over
all your money to me, and come to live with forty−six other
slaves in a tent until I decide otherwise. You will obey my
every whim. You will address me as The Great One, or
informally as Your Greatness. $
```

As you can see, in order to open a file for writing, you simply need to precede the filename in the open() function call with a greater−than sign. Once the file has been opened, it's a simple matter to print text to it by specifying the name of the appropriate file handle in the print() function call. Close the file, and you're done!

The single greater−than sign indicates that the file is to be opened for writing, and will destroy the existing contents of the file, should it already exist. If, instead, you'd like to append data to an existing file, you would need to use two such greater−than signs before the filename, like this:

```
# open file for appending open(BRAINWASH,">>wash.txt");
```

Die! Die! Die!

It's strange but true − an incorrectly opened file handle in Perl fails to generate any warnings at all. So, if you specify a file read, but the file doesn't exist, the file handle will remain empty and you'll be left scratching your head and wondering why you're not getting the output you expected. And so, in this section, we'll be showing you how to trap such an error and generate an appropriate warning.

Let's go back to our first example and add a line of code:

```
#!/usr/bin/perl # open file and define a handle for it
open(MIND,"thoughts.txt") || die "Unable to open file!\n"; #
print data from handle print <MIND>; # close file when done
close(MIND); # display message when done print "Done!\n";
```

The die() function above is frequently used in situations which require the program to exit when it encounters a fatal error − in this case, if it's unable to find the required file.

If you ran this program yourself after removing the file "thoughts.txt", this is what you would see:

```
Unable to open file!
```

Obviously, this works even when writing to a file:

```
#!/usr/bin/perl # open file for writing
open(BRAINWASH,">wash.txt") || die "Cannot write to file!\n";
# print some data to it print BRAINWASH "You will leave your
home and family, sign over all your money to me, and come to
live with forty−six other slaves in a tent until I decide
otherwise. You will obey my every whim. You will address me as
The Great One, or informally as Your Greatness.\n"; # close
file when done close (BRAINWASH);
```

Testing Times

Perl also comes with a bunch of operators that allow you to test the status of a file − for example, find out whether it exists, whether it's empty, whether it's readable or writable, and whether it's a binary or text file. Of these, the most commonly used operator is the "−e" operator, which is used to test for the existence of a specific file.

Here's an example which asks the user to enter the path to a file, and then returns a message displaying whether or not the file exists:

```
#!/usr/bin/perl print "Enter path to file: "; $path = <STDIN>;
chomp $path; if (−e $path) { print "File exists!\n"; } else {
print "File does not exist!\n"; }
```

There are many more operators − here's a list of the most useful ones, together with an example which builds on the one above to provide more information on the file specified by the user.

```
OPERATOR: TESTS WHETHER: −−−−−−−−−−−−−−−−−−−−−−−−−−− −z File
exists and is zero bytes in size −s File exists and is
non−zero bytes in size −r File is readable −w File is writable
−x File is executable −T File is text −B File is binary
```

And here's a script that demonstrates how you could use these operators to obtain information on any file on your system:

```
#!/usr/bin/perl # ask for file path and process it print
"Enter path to file: "; $path = <STDIN>; chomp $path; # test
for existence if (−e $path) { print "File exists!\n"; # test
for size if (−z $path) { print "File is empty.\n"; } else {
print "File is not empty.\n"; } # test for read access if (−r
$path) { print "File is readable.\n"; } else { print "File is
not readable.\n"; } # test for write access if (−w $path) {
print "File is writable.\n"; } else { print "File is not
writable.\n"; } # test for executable bit if (−x $path) {
print "File is executable.\n"; } else { print "File is not
executable.\n"; } # test for whether file is text or binary if
(−T $path) { print "File is a text file.\n"; } elsif (−B
$path) { print "File is a binary file.\n"; } } else { print
"File does not exist!\n"; }
```

And here's what it might look like:

```
Enter path to file: /usr/bin/mc File exists! File is not
empty. File is readable. File is not writable. File is
executable. File is a binary file.
```

Popguns And Pushpins

If you've been paying attention, you should now know the basics of reading and writing files in Perl. And as you've seen, reading a file into an array is a simple and efficient way of getting data from an external file into your Perl program. But once you've got it in, what do you do with it?

Well, over the next couple of pages, we're going to be taking a look at some of Perl's most useful array functions − these babies will let you split, join and re−arrange array elements to your heart − followed by a real−life example that should help you put it all in context.

Let's start with a basic example:

```
#!/usr/bin/perl # open file and define a handle for it
open(MIND,"thoughts.txt") || die "Unable to open file!\n"; #
suck the file into an array @file = <MIND>; # close file when
done close(MIND); # use a loop to keep reading the file #
until it reaches the end foreach $line (@file) { print $line;
}
```

At this point, the contents of file "thoughts.txt" are stored in the array @file. Now, let's add some code that asks the user for his thoughts on what he's just seen, adds those comments to the end of the array, and writes the whole shebang back to the file "thoughts.txt"

```
#!/usr/bin/perl # open file and define a handle for it
open(MIND,"thoughts.txt") || die("Unable to open file!\n"); #
suck the file into an array @file = <MIND>; # close file when
done close(MIND); # use a loop to keep reading the file #
until it reaches the end foreach $line (@file) { print $line;
} # ask for input and process it print "What do you think?\n";
$comment = <STDIN>; # add comment to end of array push (@file,
$comment); # open file for writing open(MIND,">thoughts.txt")
|| die("Unable to open file!\n"); # print array back into file
foreach $line (@file) { print MIND $line; } # close file when
done close(MIND);
```

The important thing to note here is the push() function, which adds an element − the user's input − to the end of an array. The entire array is then written back to the file "thoughts.txt", destroying the original contents in the process. So, if you were to run

```
$ cat thoughts.txt
```

at your shell, you'd see

```
We're running out of space on planet Earth. Scientists are
attempting to colonize Mars. I have a huge amount of empty
```

```
real estate in my mind. Imagine if I could rent it to the
citizens of Earth for a nominal monthly fee. Would I be rich?
Or just crazy? This idea sucks!
```

Now, how about removing that last line? Simple − use the pop() function to remove the last item from the array:

```
#!/usr/bin/perl # open file and define a handle for it
open(MIND,"thoughts.txt") || die("Unable to open file!\n"); #
suck the file into an array @file = <MIND>; # close file when
done close(MIND); # remove last element of array pop @file; #
open file for writing open(MIND,">thoughts.txt") ||
die("Unable to open file!\n"); # print array back into file
foreach $line (@file) { print MIND $line; } # close file when
done close (MIND);
```

The pop() function removes the last element of the array specified. And when you "cat" the file again, you'll see that the last line has been removed.

Shifting Things Around

pop() and push() work on the last element of the array. If you'd prefer to add something to the beginning of the array list, you need to use unshift():

```
#!/usr/bin/perl # open file and define a handle for it
open(MIND,"thoughts.txt") || die("Unable to open file!\n"); #
suck the file into an array @file = <MIND>; # close file when
done close(MIND); # use a loop to keep reading the file #
until it reaches the end foreach $line (@file) { print $line;
} # ask for input and process it print "How about a title?\n";
$title = <STDIN>; # add title to beginning of array unshift
(@file, $title); # open file for writing
open(MIND,">thoughts.txt") || die("Unable to open file!\n"); #
print array back into file foreach $line (@file) { print MIND
$line; } # close file when done close(MIND);
```

And the file "thoughts.txt" will now contain a title, as entered by the user.

Obviously, removing the first element of an array requires you to use the shift() function − we'll leave you to experiment with that one yourself.

The Greatest Things Since Sliced() Bread

Next, two of Perl's most frequently−used functions − split() and join(). split() is used to split a string value into sub−sections, and place each of these sections into an array, while join() does exactly the opposite − it takes the various elements of an array, and joins them together into a single string, which is then assigned to a scalar variable.

Let's take a simple example:

```
#!/usr/bin/perl # set up a variable $string = "This is a
string"; # split on spaces and store in array @dummy = split("
", $string); # print the array foreach $word (@dummy) { print
"$word\n"; }
```

Here's the output:

```
This is a string
```

In this case, we're splitting the string using a space as the separator − each element of the split string is then stored in an array.

You can also join array elements into a single string:

```
#!/usr/bin/perl # set up a variable $string = "This is a
string"; # split on spaces and store in array @dummy = split("
", $string); # join the words back with a different separator
$newstring = join(":", @dummy); # print the result print
"$newstring\n";
```

And the output is:

```
This:is:a:string
```

And finally, we have the splice() function, which is used to extract contiguous sections of an array [if you don't know what contiguous means, this is a good time to find out!]. Here's an example which uses the splice() function to extract a section of an array, and assign it to a new array variable.

```
#!/usr/bin/perl # set up a variable $string = "Why did the fox
jump over the moon?"; # split on spaces and store in array
@dummy = split(" ", $string); # extract three words @newdummy
= splice(@dummy, 1, 4); # join them together and print
$newstring = join(" ", @newdummy); print "$newstring\n";
```

The splice() function takes three arguments − the name of the array variable from which to extract elements, the starting index, and the number of elements to extract. In this case, the output would be:

```
did the fox jump
```

You should note that splice() alters the original array as well − in the example above, the original array would now only contain

```
@dummy = ("Why", "over", "the", "moon?");
```

The Real World

Now, while all this is well and good, you're probably wondering just how useful all this is. Well, we've put together a little case study for you, based on our real−world experiences, which should help to put everything you've learned today in context [do note that for purposes of illustration, we've over−simplified some aspects of the case study, and completely omitted others].

As a content production house, we have a substantial content catalog on a variety of subjects, much of it stored in ASCII text files for easy retrieval. Some time ago, we decided to use sections of this content on a customer's Web site, and so had to come up with a method of reading our data files, and generating HTML output from them. Since Perl is particularly suited to text processing, we decided to use it to turn our raw data into the output the customer needed.

Here's a sample raw data file, which contains a movie review and related information − let's call it "sample.dat":

```
The Insider 1999 Al Pacino and Russell Crowe Michael Mann
178.00 5 This is the body of the review. It could consist of
multiple paragraphs. We're not including the entire review
here...you've probably already seen the movie. If you haven't,
you should!
```

Within the data block, there are two main components − the header, which contains information about the title, the cast and director, and the length; and the review itself. We know that the first six lines of the file are restricted to header information, and anything following those six lines can be considered a part of the review.

And here's the script we wrote to extract data from this file, HTML−ize it and display it:

```
#!/usr/bin/perl # open the data file open(DATA, "sample.dat")
|| die ("Unable to open file!\n"); # read it into an array
@data = <DATA>; # clean it up − remove the line breaks foreach
$line (@data) { chomp $line; } # extract the header − the
first six lines @header = splice(@data, 0, 6); # join the
remaining data with HTML line breaks $review = join("<BR>",
@data); # print output print
"<HTML>\n<HEAD>\n</HEAD>\n<BODY>\n"; print "\"$header[0]\"
stars $header[2] and was directed by $header[3].\n"; print
"<P>\n"; print "Here's what we thought:\n<P>\n"; print
$review; print "Length: $header[4] minutes\n<BR>\n"; print
"Our rating: $header[5]\n<BR>\n"; print "</BODY>\n</HTML>\n";
```

And here's the output:

```
<HTML> <HEAD> </HEAD> <BODY> "The Insider" stars Al Pacino and
Russell Crowe and was directed by Michael Mann. <P> Here's
what we thought: <P> This is the body of the review.<BR><BR>It
```

```
could consist of multiple paragraphs.<BR><BR>We're not
including the entire review here...you've probably already
seen the movie. If you haven't, you should!<BR> Length: 178.00
minutes <BR> Our rating: 5 <BR> </BODY> </HTML>
```

As you can see, Perl makes it easy to extract data from a file, massage it into the format you want, and use it to generate another file.

Miscellaneous Stuff

The @ARGV variable
−−−−−−−−−−−−−−−−−

Perl allows you to specify additional parameters to your program on the command line − these parameters are stored in a special array variable called @ARGV. Thus, the first parameter passed to a program on the command line would be stored in the variable $ARGV[0], the second in $ARGV[1], and so on.

Using this information, you could re−write the very first file−test example above to use @ARGV instead of the <STDIN> variable for input:

```
#!/usr/bin/perl if (−e $ARGV[0]) { print "File exists!\n"; }
else { print "File does not exist!\n"; }
```

<STDIN> and other file handles
−−−−−−−−−−−−−−−−−−−−−−−−−−−−

Perl comes with a few pre−defined file handles − one of them is <STDIN>, which refers to the standard input device. Additionally, there's <STDOUT>, which refers to the default output device [usually the terminal] and <STDERR>, which is the place where all error messages go [also usually the terminal].

Logical Operators
−−−−−−−−−−−−−−−−

You've probably seen the || and &constructs in the examples we've shown you over the past few lessons. We haven't explained them yet − and so we're going to rectify that right now.

Both || and &are logical operators, commonly used in conditional expressions. The || operator indicates an OR condition, while the &operator indicates an AND condition. For example, consider this:

```
if (a == 0 || a == 1) { do this! }
```

In English, this would translate to "if a equals zero OR a equals one, do this!" But in the case of

```
if (a == 0 &b == 0) { do this! }
```

the "do this!" statement would be executed only if a equals zero AND b equals zero.

Perl also comes with a third logical operator, the NOT operator − it's usually indicated by a prefixed exclamation mark[!]. Consider the two examples below:

```
if (a) { do this! }
```

In English, "if a exists, do this!"

```
if (!a) { do this! }
```

In English, "if a does NOT exist, do this!"

And that's about it for this week. We'll be back in a couple of weeks with more Perl 101. Till then...stay healthy!