How PERL Works

by Marshall Brain

PERL is a fairly straightforward, widely known and well-respected scripting language. It is used for a variety of tasks (for example, you can use it to create the equivalent of DOS batch files or C shell scripts), but in the context of Web development it is used to develop CGI scripts.

One of the nice things about PERL is that, because it is a scripting language, people give away **source code** for their programs. This gives you the opportunity to learn PERL by example, and you can also download and modify thousands of PERL scripts for your own use. One of the bad things about PERL is that much of this free code is impossible to understand. PERL lends itself to an unbelievably cryptic style!

This article assumes that you already know how to program (if you know the C programming language, this will be especially easy for you). PERL is easy to use once you know the basics. In this edition of **HowStuffWorks**, we're going to start at the beginning and show you how to do the most common programming tasks using PERL. By the end of this article, you will be able to write your own PERL scripts with relative ease, and read cryptic scripts written by others with somewhat less ease, but this will be a good starting point.

Getting Started

To start with PERL you need the **PERL interpreter**. On any UNIX machine there is a 99.99-percent probability that it's already there. On a Windows machine or a Mac, you need to download the latest release of the language and install it on your machine. (See the links at the end of this article for more information.) PERL is widely available on the Web and is free.

Next, make sure you look in the **DOCS directory** that comes with PERL -- there will be user-manual-type stuff in there. At some point, it would not hurt to read through all of the documentation, or at least scan it. Initially it will be cumbersome, but after reading this article it will make much more sense.

Hello World

Once you have PERL loaded, make sure you have your path properly set to include the PERL executable. Then, open a text editor and create a text file. In the file, place the following line:

```
print "Hello World!\n";
```

Name the file "test1.pl". At the command prompt, type:

```
perl test1.pl
```

PERL will run and execute the code in the text file. You should see the words "Hello World!" printed to stdout (standard out). As you can see, it is extremely easy to create and run programs in PERL. (If you are using UNIX, you can place a comment like **#! /usr/bin/perl** on the first line, and then you will not have to type the word "perl" at the command line.)

The **print** command prints things to stdout. The **\n** notation is a line feed. That would be more clear if you modified the test program to look like this (**#** denotes a comment):

```
# Print on two lines
print "Hello\nWorld!\n";
```

Note that the print command understood that it should interpret the "\n" as a line feed and not as the literal characters. The interpretation occurred not because of the print command, but because of the use of double quotes (a practice called **quoting** in PERL). If you were to use single quotes instead, as in:

```
   print 'Hello\nWorld!\n';
```

the \n character would not be interpreted but instead would be used literally.

There is also the backquote character: **`**. A pair of these imply that what is inside the quotes should be interpreted as an operating system command, and that command should be executed with the output of the command being printed. If you were to place inside the backquotes a command-line operation from the operating system, it would execute. For example, on Windows NT you can say:

```
   print `cmd /c dir`;
```

to run the DIR command and see a list of files from the current directory.

> **PERL Note**
>
> In **Windows NT**, you cannot say:
>
> ```
>    print `dir`;
> ```
>
> because dir is not a separate executable -- it's part of the command interpreter cmd. Type **cmd /?** at a DOS prompt for details.

You will also see the ***/*** character used for quoting regular expressions.

The print command understands **commas** as separators. For example:

```
   print 'hello', "\n", 'world!';
```

However, you will also see a **period**:

```
   print 'hello'. "\n". 'world!';
```

The period is actually a **string concatenation** operator.

There is also a **printf** operator for C folks.

Variables

Variables are interesting in PERL. You do not declare them, and you always use a **$** to denote them. They come into existence at first use. For example:

```
   $s = "Hello\nWorld\n";
   $t = 'Hello\nWorld\n';
   print $s, "\n", $t;
```

Or:

```
   $i = 5;
   $j = $i + 5;
   print $i, "\t", $i + 1, "\t",  $j;     # \t = tab
```

Or:

```
$a = "Hello ";
$b = "World\n";
$c = $a . $b;    # note use of . to concat strings
print $c;
```

Since **.** is string concatenation, .= has the expected meaning in the same way that "+=" does in C. Therefore, you can say:

```
$a = "Hello ";
$b = "World\n";
$a .= $b;
print $a;
```

You can also create **arrays**:

```
@a = ('cat', 'dog', 'eel');
print @a, "\n";
print $#a, "\n";  # The value of the highest index, zero based
print $a[0], "\n";
print $a[0], $a[1], $a[2], "\n";
```

The **$#** notation gets the highest index in the array, equivalent to the number of elements in the array minus 1. As in C, all arrays start indexing at zero.

You can also create **hashes**:

```
%h = ('dog', 'bark', 'cat', 'meow', 'eel', 'zap');
print "The dog says ", $h{'dog'};
```

Here, 'bark' is associated with the word 'dog', 'meow' with 'cat', and so on. A more expressive syntax for the same declaration is:

```
%h = (
   dog => 'bark',
   cat => 'meow',
   eel => 'zap'
);
```

The **=>** operator quotes the left string and acts as a comma.

Loops and Ifs

You can create a simple **for** loop like you do in C:

```
for ($i = 0; $i < 10; $i++)
{
   print $i, "\n";
}
```

PERL Note

You must use the "begin" and "end" **braces** -- **{** and **}** -- even for a single line.

While statements are easy:

```
$i = 0;
while ( $i < 10 )
```

```
   {
      print $i, "\n";
      $i++;
   }
```

If statements are similarly easy:

```
   for ($i = 0; $i < 10; $i++)
   {
      if ($i != 5)
      {
         print $i, "\n";
      }
   }
```

The **boolean operators** work like they do in C:

- **&&** and
- **||** or
- **!** not
- For numbers:
    - **==** equal
    - **!=** not equal
    - **<**, **<=**, **>**, **>=** (as expected)
- Others:
    - **eq**
    - **ne**
    - **lt**
    - **le**
    - **gt**
    - **ge**

If you have an array, you can loop through it easily with **foreach**:

```
   @a = ('dog', 'cat', 'eel');
   foreach $b (@a)
   {
      print $b, "\n";
   }
```

Foreach takes each element of the array @a and places it in $b until @a is exhausted.

Functions

You create a **subroutine** with the word **sub**. All variables passed to the subroutine arrive in an array called _. Therefore, the following code works:

```
   show ('cat', 'dog', 'eel');

   sub show
   {
      for ($i = 0; $i <= $#_; $i++)
      {
         print $_[$i], "\n";
      }
   }
```

Remember that $# returns the highest index in the array (the number of elements minus 1), so

$#_ is the number of parameters minus 1. If you like that sort of obtuseness, then you will love PERL.

You can declare **local variables in a subroutine** with the word **local**, as in:

```
sub xxx
{
   local ($a, $b, $c)
   ...
}
```

You can also call a **function** using **&**, as in:

```
&show ('a', 'b', 'c');
```

The & symbol is required only when there is ambiguity, but some programmers use it all the time.

To **return a value from a subroutine**, use the keyword **return**.

Reading

Reading from STDIN

To read in data from the stdin (standard in), use the **STDIN** handle. For example:

```
print "Enter high number: ";
$i = <STDIN>;
for ($j = 0; $j <= $i; $j++)
{
   print $j, "\n";
}
```

As long as you enter an **integer** number, this program will work as expected. **<STDIN>** reads a line at a time. You can also use **getc** to read one **character**, as in:

```
$i = getc(STDIN);
```

Or use read:

```
read(STDIN, $i, 1);
```

The 1 in the third parameter to the read command is the length of the input to read.

Reading Environment Variables

PERL defines a global hash named **ENV**, and you can use it to **retrieve the values of environment variables**. For example:

```
print $ENV{'PATH'};
```

PERL Note
The name of the **environment variable** must be **upper case.**

Reading Command Line Arguments

PERL defines a global array **ARGV**, which contains any command line arguments passed to the script. $#ARGV is the number of arguments passed minus 1, $ARGV[0] is the first argument passed, $ARGV[1] is the second, and so on.