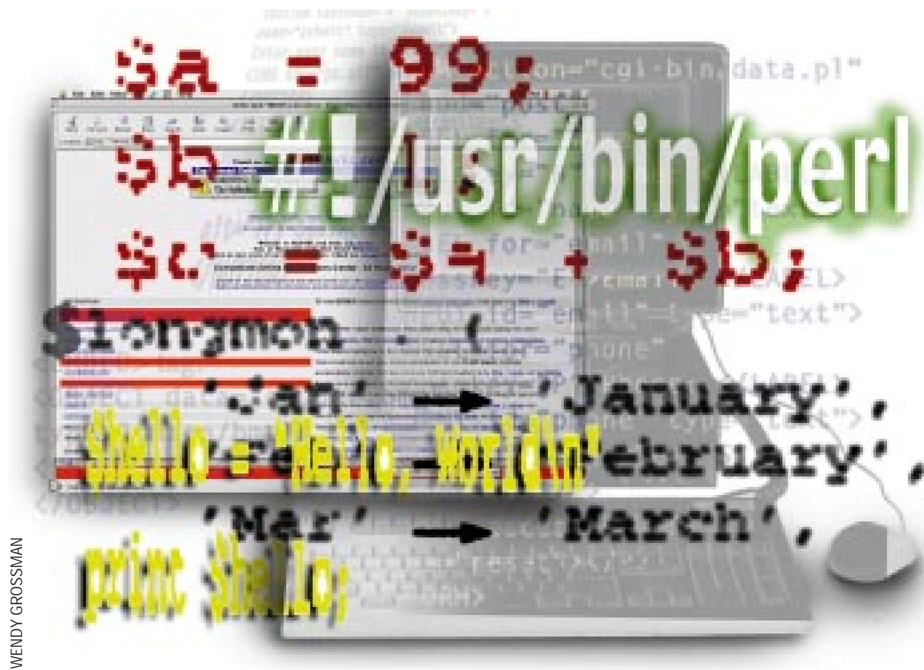
*by **Peter Collinson,*** Hillside Systems



WENDY GROSSMAN

Getting Started with Perl

I must confess that I have only picked up Perl in the last year or two. I was going to say that I was a recent convert to Perl, but that statement would have had religious overtones, and I cannot say that I have become a devotee. I just use it when it's appropriate. I've gone past the point where I become a zealot preaching for or against a particular system or programming language. Also, it seems I only manage to learn new languages when I have a need to do so. Usually, I am engaged in a task to get something done and having to fight with a new language gets in the way. I use what I know.

The force that has made me pick up Perl is the need to create CGI scripts for the Web, and I seem to be engaged in this activity more and more these days. Perl is a very "big" language, in the sense that there are a great many syntactic features, built-in functions and routines. You can treat this wealth strictly on a need-to-know basis, and I certainly did this at the start, picking up my Perl book when I reached a stage where I wanted to do something new. Actually, there are many different ways of programming a task in Perl, and it's possible to get by with only a small subset of what is available. Nevertheless it makes sense to keep reverting to the book to see if there's a better way to do something.

It never makes sense to reach the nirvana of the Nineteenth Stage of Perl Devoteedom, where you seek to express your program in the minimum number of characters and lines, making the result impenetrable to mortals. Perl is accused of being a "write once, read never" language. However, it is Perl programmers, not the language, that should be accused, I suppose.

I always felt that early versions had too much magic. Magic, in the sense that simple operations have side effects, setting some variable or other, and that programmers were originally encouraged to write obscure programs that used these side effects. Good code is easily understood by others, and this means that what is happening in the source code should be plain to see. You should be able to understand what is happening without having a huge background knowledge of special symbols, oddly named variables and the side effects of some statements. It seems to me the most recent version of Perl, Version 5, is beginning to learn this lesson and it's now possible to write straightforward, maintainable, easy-to-read code.

Another great Perl plus is there is a vast body of library routines you can use to create your programs more quickly. The language is distributed with a set of libraries that are documented in books and online. In addition, there's the Comprehensive Perl Archive Network (CPAN), a large set of user-contributed code available via the Internet. I am only now beginning to look at these libraries and use them. I made a conscious decision to avoid them at the start because I feel that when a language is new and unfamiliar, the last thing you want to do is spend time fighting with (and possibly debugging) someone else's code.

Also, when you use a predefined

library you are buying into a view of how to achieve something, and that view can be enabling or it can be constraining. I worry about the constraint. There are some portability issues, for example, CPAN code will sometimes only work with certain versions of Perl (and also later versions, we hope). If you write a program that depends on a library and want to move it elsewhere, then you have to port the library with your code and there's a chance you'll end up working to get someone else's library to operate in the target environment. Debugging other people's code is not advised when you are new to a language and only understand a small part of what is available.

Finally, you have the decision of which version of Perl to choose. Perl underwent a large change with the release of Version 5. Perl 5 is not backwards compatible and will not run some Perl programs written for previous versions. On my BSD/OS system, this has led to the situation where Perl 5 is installed as `perl5` and the `perl` command is actually late Version 4. If you are new to the language, then I advise you to use Perl 5 because this is the version that all the books document. You can always find out the version in use on your machine by typing `perl -v`.

Scripts

Well, by now you should have gathered that Perl is a programming language. It's actually a scripting language. You write a program into a file, and when you use `perl` on that file, the program is compiled *and* run, assuming that the compilation succeeds. Getting a Perl program to execute is a one-stage operation. On UNIX, you can also make the file itself into an executable command by making the first line:

```
#!/usr/bin/perl
```

and setting the execute bits on the file:

```
$ chmod +x file
```

Now, when you attempt to use the file as a command, the kernel notices the `#!` and starts the program that is found by using the absolute path following the `#!` character pair. The file itself is passed into that running program and, for Perl, will be compiled and run. This general mechanism is available for all interpreters.

Script languages have the disadvantage that you need to run a compiler and syntax-check the code every time you use the command. This has become less of a problem now that CPUs are very fast. The advantage of scripts is speed of development, you just change the source and have a new program.

Perl was originally a "very unsafe" language, that is, there was not much help for the programmer to eliminate typos and other inaccuracies that would creep in. Recent versions have included somewhat stricter checking (that needs to be requested) and an option to the interpreter, `-w`, which makes the interpreter moan about variables that are used only once and variables that are used before being set (among other things). So you will often see

```
#!/usr/bin/perl -w
```

at the start of a Perl program to turn on this option. Incidentally, most systems only permit one argument after the path name at the start of a script file.

Constants and Variables

Probably the best way to learn Perl is to read other people's code, after having read as many books as you can cope with. The main book (see below) is dense and is often better dipped into to find what you are looking for. When I was on the learning curve, I spent a lot of time searching books for equivalent examples for what I was trying to do. This article should give you a basic grounding so you can examine other people's code to learn more.

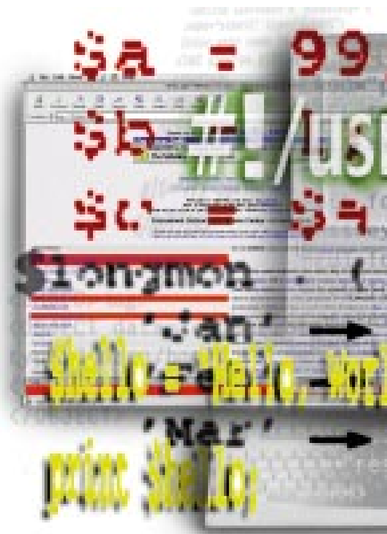
Let's start with a brief description of constants and variables. In any programming language, named variables are used to store information, and Perl is no different. Where Perl diverges from many languages is it uses an initial magic character to tell the interpreter how the variable is to be treated. The code

```
$hello = "Hello, World\n";
print $hello;
```

is a complete Perl program. You can type it into a file, perhaps called `hello.pl`, and say

```
perl hello.pl
```

As you can see, Perl programs consist of statements terminated with a semicolon–actually, Perl's syntax follows the rules of the C language. You could put the statements on the same text line if you wish, it's the semicolon that indicates the end of the statement. The first line sets the variable `hello` to the string `Hello, World` terminated by a newline character indicated by backslash-n. The double-quoted string is a constant, but one that will be examined by Perl to see if it can find special characters (or variables) that need replacing. Perl borrows the quote syntax from the Bourne shell and also supports single-quoted strings, where the constant value is not examined for possible character or variable replacement. There are also other methods of quoting strings.



The dollar symbol before the variable `hello` tells Perl to interpret the variable as a *scalar*, holding a single value. Perl supports arrays too. They are numbered from zero and individual elements are accessed

by writing something like this:

```
$array[6]
```

If we want to talk about the array as a whole, then @ is used before the name. The other magic characters that you will see before names are &, which is used to show that the variable is the name of a function or routine, and %, which is used to indicate an associative array (more on these later).

Scalar variables can hold numeric values (9, 9.9, 99.99 and so on) and text strings. You don't have to tell Perl that a variable is to hold one particular type or another. The way that a variable is interpreted is determined by context. Perl provides a set of primitive string operators and ensures that they are distinguishable from the set of numeric operators. So,

```
$a = 99;
$b = 1;
$c = $a + $b;
```

will give you the numeric answer "100" in the variable "c." Other languages will overload the plus operator to supply string concatenation; Perl doesn't, you use the dot operator to join the strings. The line

```
$c = $a . $b;
```

will place "991" into the "c" variable. The use of special operators to denote the type of operation is particularly relevant when you test values in if statements. For example,

```
if ($c == 999) { ...
```

performs a numeric equality test, while

```
if ($c eq "99") { ...
```

does a string comparison. This is easy to see when you are testing a variable against a constant, but consider the following:

```
if ($a eq $c) {
```

Here, we are forcing a string comparison to be made. Incidentally, Perl has acquired the ability to place an operator before an assignment statement from C. So,

```
$v = $v + 200;
```

will take the value of the "v" variable and add "200" to it, placing the answer back into "v." This can also be written as

```
$v += 200;
```

which saves keystrokes. I mention this because I recall having difficulty in understanding what

```
$s .= "appended";
```

meant. This is similar to our first example; that is, the string "appended" is added to the end of the "s" variable. In fact, Perl is very good at handling strings and provides a number of built-in routines that enable you to take the strings apart and put them back together again in a different order.

There's one other piece of assignment syntax that you'll see everywhere, the bind operator =~, whose significance is often lost when you first start reading other people's Perl. There are several built-in functions that apparently take no arguments. In early versions of Perl they operated on a hidden magic variable, $_, which accessed the "last" result. The functions are all string-processing ones and I don't want to go into their full ramifications in this article. Perhaps the simplest one to explain is the substitute command. The s command allows you to replace some part of one string with another. Therefore,

```
s/old/new/;
```

replaces the first occurrence of the string "old" in the magic variable $_ with "new." In many recent Perl programs, you'll see the bind operator being used to supply an operand for the command:

```
$changethis =~ s/old/new/;
```

The operator =~ applies the s operator to the value of $changethis and, in this case, also replaces the old value with the result of the substitution.

Arrays and Lists

Perl has good support for handling arrays. In fact, programmers often think of arrays in terms of lists of objects, and Perl uses round brackets ubiquitously to define lists. We can create a list as follows:

```
@hello = ("Hello,", "World\n");
```

This creates a list of two elements in an array. Notice that I am using the @ to indicate the loading of an array. Now we can talk about each element by using standard array syntax, as follows:

```
print $hello[0], $hello[1];
```

However, we have not printed the space between the words, so perhaps we can employ the following:

```
print "$hello[0] $hello[1]";
```

In fact, I rarely seem to have ordered data in my programs, apart from days of the week or months of the year. I'll use something like this:

```
@dow = ("Sunday", "Monday",
```
(and so on)

and access it via a statement such as

```
print $dow[$today]."\n";
```

assuming that "Sunday" is coded as a numeric value of zero in the variable "today." The last piece of the statement adds a new line using string concatenation and is another common idiom.

The other type of array that Perl supports is the *associative* array, where indexes are not increasing numerical values but may be any string. This type of array is often called a *hash* in Perl documentation. I first came across associative arrays in awk and I suspect Perl borrowed them from that language. However, Perl has its own syntax to set up constant values, for example,

```
%longmon = (
     'Jan' => 'January',
     'Feb' => 'February',
     'Mar' => 'March',
);
```

We can now access the data, turning a short-form name into a long-form:

```
print $longmon{'Jan'};
```

Note the use of curly braces to indicate the associative lookup.

I tend to use associative arrays in my scripts to store string constants used by the program for things like file names or URLs, so I can localize the values that I use in the script into one place, making for easy editing.

Associative arrays are of great use when indexes are taken from variables. For a real example, suppose we generate a Web enquiry form with a drop-down menu that specifies different destinations for where the data is to be sent, allowing the user to choose to send the form to sales, management, enquiries and the like. I am providing contact forms more and more now because it seems that email addresses on Web pages are targets for spammers. When the user makes a selection, we can arrange for the Web form to return a string that is a lookup string into the list of real email addresses, and use an associative array to perform the translation:

```
$emailto = $lookup{$formval};
```

We could use a regular array here but then we would have to code the values from the form appropriately, where "sales" returns zero, "management" returns one and so on. However, it's much more natural, and makes for easier maintenance, for the form to return a string that maps onto a readable string in the program.

Perl also allows you to place lists on the left-hand side of an assignment statement, moving multiple values from one place to another. So we could say

```
($sun, $mon, @rest) = @dow;
```

making use of the dow array I created earlier to assign "Sunday" to the $sun variable, "Monday" to the mon variable and the rest of the dow array to rest. You'll find that the construction is mostly used to process the returned list values from functions, and very nice it is too.

Functions and Procedures

Hopefully, every Perl program that you see will be well structured and split into small procedures, each doing one thing well. Some people only believe in using functions and procedures if the code is used more than once, thus saving code. Well, this is actually incorrect thinking. Procedures and functions should be used to separate code into logical sections. A good rule of thumb is to separate control (going round a loop several times, or testing values in if statements) from actions (doing the work that needs to be done in the loop or if statement branch). Control should be placed in one routine and actions in another, then, by picking sensible names for the action routines, all the code will become more readable.

Functions and routines are introduced by the keyword sub, which is followed by the name used to call the procedure. The body of the routine is enclosed in curly braces:

```
sub hello {
    return "Hello, World\n";
}
```

When you call the routine, you should precede it with &. Well, the "main" book says, "ampersand is optional when it's unambiguous," a statement of deep mystery to a novice. One place you need it is

```
$hello = &hello;
```

when you are calling a routine with no parameters. If the routine has bracketed parameters, then it's not needed (I think). Incidentally, it's perfectly OK to have a scalar variable called hello, a function called hello and an array called hello. All three names refer to distinct objects. However, if you actually do this, may deep woe be upon you and your house because it will confuse everyone who reads your code.

If you are used to other programming languages, you'll find the way Perl handles parameters seems somewhat arcane. You make a routine call such as

```
$val = r($a, $b, $c);
```

which is familiar. When you define the routine, there are no "formal" parameters, you pick up the variable values from the magic "last value" variable, which is assumed to contain a list of values. So we could define

```
sub r {
   @args = @_;
   ...
}
```

placing all the routine parameters into an array. Notice how we use the @ character before the underscore to tell Perl that we want to treat the arguments as a complete array. Alternatively, if we wanted to place the values into some variables, we could use the list-assign syntax that we saw above:

```
sub r {
    ($one, $two, $three) = @_;
    ...
}
```

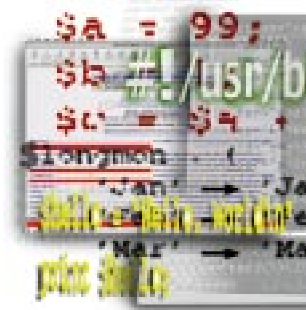
Another common idiom is to use the `shift` operator:

```
sub r {
    $one = shift;
    $two = shift;
    $three = shift;
    ...
}
```

The `shift` operator returns the first element of a list, removes it and leaves the remaining elements intact. All this seems pretty weird and wonderful, but it seems to work and provides great flexibility when creating routines.

Perl is a big topic, and I've only managed to scratch the surface here. Obviously, there are omissions from the above text, so please don't email me saying "you missed this." The intention was not to present an exhaustive syntactic description but to get across some of the basics, allowing further progress.

The "main" book you will need is *Programming Perl, 2nd Edition,* by Larry Wall, Tom Christiansen and Randal L. Schwartz, published by O'Reilly & Associates Inc., 1996, ISBN 1-56592-149-6. Just in case you didn't know, Larry Wall is the originator of Perl. This book is dense and is really a manual with examples. My copy is very well-thumbed. *Learning Perl, 2nd Edition,* by Randal L. Schwartz and Tom Christiansen, again from O'Reilly & Associates, 1997, ISBN 1-56592-284-0, is actually a good place to start.



You can get Perl source (and libraries) from CPAN. The main URL is `http://www.perl.com`, a site run by the aforementioned Tom Christiansen.

***Peter Collinson** runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever… He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.