INTRODUCTION TO PERL

```
#!/usr/bin/perl
# This program prints out hello world!

print "hello world!  This is a very simple Perl program.\n";
```

- type in using your favourite text editor
- make it executable (on Unix) with

```
chmod +x hello.pl
```

Notes

- shell interpreter line at the top points to where perl is installed. This can have parameters, such as -w (for warnings), and -T (forces "taint" checks on).
- end all command lines with a semicolon (;)
- quotes around text
- # indicates a comment - rest of line is ignored

VARIABLES

$	scalar variable/context	
@	array variable	
	$ary[n]	single element of an array
	$#ary	offset of last element in ary (starts with 0)
	scalar(@ary)	returns number of elements in an array
	pop, push, shift, unshift - add or remove elements from an array	
%	hash variable	
	$hsh{key}	single value of a hash
	delete(key)	delete element of hash
	keys, values	return list of keys or values
	each (→ 2 element list: key, value)	
&	subroutine name (generally used for local subroutines, but not required)	
*	typeglob	

(functions for returning hash keys, values)

- FileHandles are variables with no special characters, traditionally use all uppercase. (eg. SESAME, PTO)
- scalar variables can be numeric or text, Perl interprets value based on context

```
#!/usr/bin/perl

$| = 1;

$foo = "snow";

@fruits = ("apple", "persimmon", "banana");

%roster = ( Monday => 'Cindy',
            Tuesday => 'Frank');
$roster{Wednesday} => 'Joe' );

print "Foo is $foo.\n\n";
print "Fruits has ",scalar @fruits, " values.  They are:\n";
foreach (@fruits) {
    print "    $_\n";
}
foreach $day (keys %roster) {
   print "$roster{$day} is rostered for $day.\n";
}
```

} → table "Mon: Cindy
Tue: Frank
Wed: Joe "

Notes

- Use double quotes (") when want values interpolated
- Use single quotes (') for no interpolation
- blocks start and end with {}, and aren't followed by a semicolon (;)
- list context for print - functions take list (array) as argument but don't need brackets ()
- $| =1 forces a flush of buffers after every print, off by default
- $_ is the default scalar variable (@_ is default array variable in a subroutine)

INPUT/OUTPUT

```
open(SESAME, "filename);                  # read from existing file
open(SESAME, "<filename);                 #  (same thing)
open(SESAME, ">filename);                 # create file and write to it
open(SESAME, ">>filename);                # append to existing file
open(SESAME, "| output-pipe-command")  # set up an output filter
open(SESAME, "input-pipe-command|"     # set up an input filter
```

```
#!/usr/bin/perl

open (IN, 'grep March time.data|');
open (RPT, '>mthly.rpt');

while (chomp($line = <IN>)) {
    ($project,$time) = split / /, $line;
    $totals{$project} += $time;
}

foreach $name (sort (keys(%totals))) {
    print RPT "$name:  $totals{$name}\n";
}
```

Notes

- input is a scalar context, use split to use in array context

<IN> reads a line at a time.

OPERATORS

Numeric

+ - / * %	add, subtract, divid, multiply, modulus
**	exponentiation
++ --	increment, decrement
	++$a # increment $a before use
	$a++ # use $a then increment $a

String

.	concatenation
x	replication

Assignment

=	assignment
+= -= *= /=	shortcuts

Logical

&& and	and
|| or	or
! not	not
||= &&=	shortcuts

Numeric comparison

== > <	equal, less than, greater than
>= <=	less than or equal, greater than or equal
!=	not equal
<=>	comparison (returns 1, 0 or -1)

String comparison

eq lt gt	equal, less than, greater than
ge le	greater than or equal, less than or equal
ne	not equal
cmp	comparison (returns 1, 0 or -1)

File test

-e	file exists
-r	file is readable
-w	file is writable
plus lots more	

Regular expression

=~	matches
!~	does not match

FLOW CONTROL

What is Truth

1. Any string is true except for **""** and **"0"**.
2. Any number is true except for 0.
3. Any reference (point) is true (because it points to an address).
4. Any **undefined** value is false.

Examples:

0	False
1	True
0.00	Becomes the numeric value 0, so false
"0"	False
2 - 2	Becomes 0, so false
"2 - 2"	Literal string, so true
"0.00"	Literal string, not "" or "0" so true
"\n"	String containing a linefeed, so true

If

```
#!/usr/bin/perl -w
$| = 1;

# Read a number from standard input
print "Enter a number between 1 and 10:  ";
$num = <STDIN>;

# Block construct
if ($num == 4)
    { print "4 compass directions\n";}
elsif ($num ==2)
    { print "You gave me a 2\n"; }
else
    { print "just a $num\n"; }

# Statement modifier
print "Larger than 5\n" if ($num > 5);
```

Notes

- block construct allows use of elsif (note spelling) and else
- statement modifier is for just one line, no else or elsif
- must use brackets () to enclose expression when using block construct

Unless

```
unless ($dest eq "home") {
    print "I am not going home\n";
}


$a += $b unless $b > $a;
```

While

```
while (expr) {
      BLOCK
}


print "Goooo!\n" while $count--;
```

Until

```
print "Going up . . .\n" until ++$f == $dest;


until ($I_am_dead) {
    &kiss('me');
}
```

Do ..while/until

```
do {
    $line = $get_line;
}
until ($line eq "\n";);
```

For

```
#!/usr/bin/perl
for ($i = 0; $i < 20; $i++) {
    $total += $i;
}

print "Total value is $total\n";
```

Foreach

```
# modifies contents of @recs
foreach $v (@recs)
    { $v += 4; }
```

Keywords

next	start the next cycle of the loop immediately
last	exit the loop immediately
continue	allows definition of continue block that is executed when loop completes or next is executed
redo	restarts innermost loop (or named loop) without evaluation of condition

Named Loops

```
L1:  foreach $x (@ary1, @ary2) {
        L2:  foreach $y ('a','b','c','d') {
                 $x .= $y;
                 next L1 if length($x) >2;
             }
     }
```

- can also have named blocks, with no loop construct

REGULAR EXPRESSIONS

Character classes

set of possible character values, enclosed in square brackets. E.g.,

[\dabc]	matches a digit or **a** or **b** or **c**
[^\dabc]	matches anything but a digit or **a** or **b** or **c**

Special codes

.	any single character
\w	word character (alphanumeric plus underscore)
\W	non-word character
\d	digit (0 - 9)
\D	non-digit
\s	whitespace character (space, tab or linefeed)
\S	non-whitespace character
^	start of string
$	end of string
\b	word boundary
\n	linefeed
\r	carriage return
\f	formfeed
\t	tab
\e	escape

Modifiers

*****	match zero or more occurrences of the previous string
+	match one or more occurrences of the previous string
?	match zero or one occurrences of the previous string
{n,m}	match at least **n** but no more than **m** occurrences

Switches

g	multiple match (used at end of match)
i	case-insensitive matching (used at end of match)
o	only compile pattern once (used at end of match)
m	use the next character as a delimiter (used at start; not used with **s**)
s	do a search and replace (used at start of match)

Backreferences

(pattern)	create backreference that is available as **$1**, **$2**, etc.

```
if ($text =~ /fred\b/i) {
    print "Found fred on a word boundary\n";
}
```

Will match "Fred Jones" but not "Frederick Jones".

```
#!/usr/bin/perl

$date = "Tuesday 15th August 1995 10:25pm";

$date =~ s/Tuesday/Tue/;

@bits = $name =~ /^(\w+)\s+(\w+).*(\d{2}:\d{2})/;

foreach $b (@bits) {
    print "Element:  $b\n";
}
```

SUBROUTINES

- large number of built in functions available
- user-defined subroutines normally called by prefixing subroutine name with **&** (required if subroutine is not declared before it is called)
- arguments are passed as a list - a set of scalars enclosed in round brackets
- many built-in functions are also list operators, which don't require brackets (they can still be used if desired)
- subroutines can be declared anywhere in the program (normally at beginning or end)
- parameters passed to a subroutine are in the default array (@_)

```
sub sum {
    my @vals = @_;
    my ($v, $total);

    foreach $v (@vals) {$total += $v; }

    return $v;
}
```

Scoping

```
#!/usr/bin/perl

# Assign a value to a lexical variable
my $name = "fred";

&my_sub();
&extra_sub();
exit 0;

sub my_sub {
  my ($name);
  $name = "jones";
  &extra_sub();
  print "from my_sub:  value is $name\n";
}

sub extra_sub {
# print the current value of the global variable $name
print "from extra_sub:  value is $name\n";
}
```

- **my** creates a variable not visible outside the current block
- can be predeclared
- **local** variables use dynamic scoping - the value of a global variable is changed until the current block is finished, then the original value is restored
- by default, variables are global (to the current package)

PACKAGES

- each package provides a global symbol table (or namespace); variables are accessed as **$package::a**.
- are included in your program with:
  - **use** command is executed at compile time; subroutines are exported into the symbol table of the application so explicit package references are not required.
  - **require** command is executed at runtime

```
package example
require Exporter;

# Export some symbols
@ISA = qw(Exporter);
@EXPORT = qw(hr br heading ul);

BEGIN {
  print "<html><head><title> Title </title></head>\n";
  print "<body>\n";
  print "Initialising package\n";
}

sub br {
  print '<br>';
}

sub heading {
  my $level = shift;
  my $text = shift;

  print "<H$level> $text </H$level>\n";
}

sub ul {
  print "<UL>\n";
  foreach (@_) {
    print "<LI> $_\n";
  }
  print "</UL>\n";
}

sub AUTOLOAD {
  my $program = $AUTOLOAD;
  my @args = @_;

  $program =~ s/.*::/;       # remove package name
  print "External process:  $program @args\n";
}

END {
  print "Shutting down ...\n";
  print "</body></html>\n";
}

1;
```

Perl 101 (Part 1) - The Basics

By Vikram Vaswani and Harish Kamath
Melonfire
May 23, 2000

The Big Picture...
Perl 101 Article Series:

- Perl 101 (Part 1) - The Basics
- Perl 101 (Part 2) - Of Variables And Operators
- Perl 101 (Part 3) - Looping The Loop
- Perl 101 (Part 4) - Mind Games
- Perl 101 (Part 5) - Sub-Zero Code

If you're a Web programmer, you're probably already well-versed with the intricacies of client-side scripting. But where there's a client, there must be a server - and so, this week, DevShed is kicking off a series of tutorials on server-side scripting. With power such as this, young Jedi, there is no limit to the evil you will be capable of...

First, though, let's start with the basics.

Server-side scripting is not new. It's been around for quite a while on the Web, and almost every major Web site uses some amount of server-side scripting. Amazon.com uses it to find the book you're looking for, Yahoo! uses it to store your personal preferences, and GeoCities uses it to generate page statistics.

Despite this, you're probably wondering why server-side scripting is such a big deal - after all, you've probably seen what a few simple JavaScripts can do. The reason for its popularity is very simple - JavaScript runs within a client application, usually the browser, and as such can only access resources, such as the current date and time, on the client machine. Since server-side scripts run on the Web server, they can be used to access server resources such as databases, system variables and other useful thingamajigs.

Just as there are different flavours of client-side scripting, there are different languages which can be used on the server as well. Here's a quick list of some of the more popular ones:

Active Server Pages [http://msdn.microsoft.com/workshop/server/asp/ASPover.asp] was introduced by Microsoft in its IIS Web server, ostensibly for the purpose of "web application programming". While ASP currently runs only on the Windows server platform, plans are afoot to port it to the UNIX platform as well.

Next up, ColdFusion, developed by Allaire [http://www.allaire.com]. ColdFusion syntax bears a remarkable resemblance to HTML, making it very easy for a budding web programmer to migrate to it. At the moment, it's available for both Windows NT and Linux. The only drawback: it ain't free, McGee!

Python [http://www.python.org] is an interpreted, object-oriented high-level scripting language for UNIX, often compared to Tcl, Perl or Java. It has modules, classes and interfaces to system calls, and is also extensible. It has been ported to Windows, DOS, OS/2 and the Macintosh, and has a devout following in the UNIX community.

And then there's the current flavour of the month, PHP [http://www.php.net]. Very easy to use, it's free, widely available for UNIX systems, and particularly strong in the areas of database access. The latest version is PHP4, and a final release is expected shortly.

And finally, Perl, one of the most popular languages around [and the language used throughout this tutorial - such is fame!]. Here's how its creator, Larry Wall, describes it: "PERL, an acronym for Practical Extraction and Report Language, is an interpreted language optimized for scanning arbitrary text files, extracting information from these files, and printing reports based on that information. It is also a good language for many system management tasks. The language is intended to be practical - easy to use, efficient, and complete - rather than beautiful - tiny, elegant, and minimal.

Perl 101 (Part 1) - The Basics

By Vikram Vaswani and Harish Kamath
Melonfire
May 23, 2000

...And The Little Language That Could!

Perl is a very popular language for server-side scripting primarily because of its close relationship with the UNIX platform. Most Web servers run UNIX or one of its variants, and Perl is available on most or all of these systems. The language is so powerful that many routine administration tasks on such systems can be carried out in it, and its superior pattern-matching techniques come in very useful for scanning large amounts of data quickly.

Geeks will be happy to hear Perl is an interpreted language. Why is this good? Well, one advantage of an interpreted language is that it allows you to perform incremental, iterative development and testing without going through a create/modify-compile-test-debug cycle each time you change your code. This can speed the development cycle drastically. And programming in Perl is relatively easy [famous last words!], especially if you have experience in C or its clones. Perl can even access C libraries and take advantage of program code written for this language, and the language is renowned for the tremendous flexibility it allows programmers in accomplishing specific tasks.

And then of course, there's cost and availability - Perl is available free of charge on the Internet, for the UNIX, Windows and Macintosh platforms. Source code and pre-compiled binaries can be obtained from http://www.perl.com/, together with installation instructions. The examples in this series of tutorials will assume Perl 5.x on Linux, although you're free to use it on the platform of your choice.

If you're on a UNIX system, a quick way to check whether Perl is already present on the system is with the UNIX "which" command. Try typing this in your UNIX shell:

```
$ which perl
```

If Perl is available, the program should return the full path to the Perl binary, usually

```
/usr/bin/perl
```

or

```
/usr/local/bin/perl
```

Or if you're really lazy, you could just ask your system administrator.

On any other platform, try checking your PATH environment variable for a directory containing the Perl executable.

Perl 101 (Part 1) - The Basics

By Vikram Vaswani and Harish Kamath
Melonfire
May 23, 2000

Your First Perl Program

And now that you've got Perl installed and configured, how about actually doing something with it? Use your favourite text editor to type the following lines of code:

```
#!/usr/bin/perl
# Perl 101
print ("Groovy, baby!\n");
```

Save this file as "groovy.pl".

Next, you need to tell the system that the file is executable. On a UNIX system, this is accomplished by setting the "executable bit" with the "chmod" command:

```
$ chmod +x groovy.pl
```

And now run the script - in UNIX, try

```
$ ./groovy.pl
```

On a Windows system, you need to pass the name of the script to the Perl executable as a parameter, like this:

```
> perl groovy.pl
```

or

```
> groovy.pl
```

In both cases, the script should return Austin Powers' trademark line. Ain't Perl groovy, baby!

In case things don't work as they should, it usually means that the system was unable to locate the Perl binary. This is a good time to holler for the system administrator.

Perl 101 (Part 1) - The Basics

By Vikram Vaswani and Harish Kamath
Melonfire
May 23, 2000

Your First Perl Program Dissected
Let's take a closer look at the script. The first line

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

is used to indicate the location of the Perl binary. This line must be included in each and every Perl script, and its omission is a common cause of heartache for novice programmers. Make it a habit to include it, and you'll live a healthier, happier life.

Next up, we have a comment.

```
# Perl 101
```

Comments in Perl are preceded by a hash [#] mark. If you're planning to make your code publicly available on the Internet, a comment is a great way to tell members of the opposite sex all about yourself - try including your phone number for optimum results.

And finally, the meat of the script:

```
print ("Groovy, baby!\n");
```

In Perl, a line of code like the one above is called a "statement". Every Perl program is a collection of statements, and a statement usually contains instructions to be carried out by the Perl interpreter.

In this particular statement, the print() function has been used to send a line of text to the screen. Like all programming languages, Perl comes with a set of built-in functions - the print() function is one you'll be seeing a lot of in the future. The text to be printed is included within double quotes, and the entire thing is then surrounded by parentheses. Note the n character, used to indicate a new line.

C programmers will be familiar with the print() function call above, as also with the fact that in C, it is mandatory to place function arguments within parentheses. Perl is more flexible than C in this regard - in many cases, parentheses can be excluded in function calls, as in this example:

```
#!/usr/bin/perl
# Perl 101
print "Groovy, baby!\n";
```

Every Perl statement ends with a semi-colon. Don't ask why - just do it!

And here's an interesting bit of trivia - every Perl statement can be further sub-divided into smaller units called "tokens". If you take a look at the statement above, you'll see three distinct tokens - print, ("Groovy, baby!\n") and the semi-colon. The interesting thing about tokens is that they can be separated with white space and tabs - which, translated, means that you could also write the above line as

```
print      ("Groovy, baby!\n")       ;
```

or

```
print("Groovy, baby!\n")                ;
```

and things would still work as advertised. Needless to say, this comes in very useful when dealing with long and complex scripts.

Perl 101 (Part 1) - The Basics

By Vikram Vaswani and Harish Kamath
Melonfire
May 23, 2000

Two Plus Two

So now you know how to print text - how about a little math? Try this:

```
#!/usr/bin/perl

# Some addition

print (10+2);
```

And this should give you the answer of that particular mathematical operation.

You can even try subtraction, multiplication and division:

```
#!/usr/bin/perl

# Some more math

print (10-2);
print (10 * 2);
print (10/2);
```

Note how the various results are concatenated together in the absence of the newline character.

And you can even combine text and mathematical operations - this will be covered in greater detail over the next few weeks, but for the moment, you can play with this simple example:

```
#!/usr/bin/perl

# Synthesis

print ("Hello there! And what are you doing ",
 1+1, "night, baby?n");
```

Perl 101 (Part 1) - The Basics

By Vikram Vaswani and Harish Kamath
Melonfire
May 23, 2000

To Err Is Human...To Debug, Divine!

Perl also comes with a very powerful debugger, which alerts you to mistakes in your code. As an example, try omitting the last double quote from the example above and run the script again. You should see something like this:

```
Can't find string terminator '"' anywhere
 before EOF at ./test.pl line 5.
```

As you can see, the error message tells you pretty much all you need to know to find and squash the bug. Of course, all Perl's error messages aren't quite that informative - some of them are liable to sound a lot like Greek spoken backwards, especially if you're a novice. And so it's a good idea to use the "-w" flag when running your program - this flag tells the debugger to print extra warnings when it encounters an error.

Take a look at the following example, which runs the Perl script above with the "-w" flag:

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

# Synthesis

print ("Hello there! And what are you doing ",
 1+1, "night, baby?n);
```

If you run this program, here's what you'll see:

```
syntax error at TEST line 5, near "print"
Unquoted string "doing" may clash with future reserved
word at TEST line 5.
String found where operator expected at TEST line 5,
near "doing ", 1+1, ""
        (Do you need to predeclare doing?)
Bare word found where operator expected at TEST line
5, near "", 1+1,
"night"
        (Missing operator before night?)
Unquoted string "night" may clash with future reserved
word at TEST line 5.
Unquoted string "baby" may clash with future reserved
word at TEST line 5.
Unquoted string "n" may clash with future reserved
word at TEST line 5.
Execution of TEST aborted due to compilation
errors.
```

Of course, there is sometimes such a thing as too much data...

Perl 101 (Part 1) - The Basics

By Vikram Vaswani and Harish Kamath
Melonfire
May 23, 2000

What's Next?

Well, that's about it for this introductory lesson - thus far, you've learned a little bit about server-side scripting, and the basic components of a Perl program. The next lesson will cover Perl's variables and operators, and you'll also be taken on a quick tour of Perl's conditional expressions. See you then!

Perl 101 (Part 2) - Of Variables And Operators

By Vikram Vaswani and Harish Kamath
Melonfire
June 01, 2000

Playing With Mud
Perl 101 Article Series:

- Perl 101 (Part 1) - The Basics
- Perl 101 (Part 2) - Of Variables And Operators
- Perl 101 (Part 3) - Looping The Loop
- Perl 101 (Part 4) - Mind Games
- Perl 101 (Part 5) - Sub-Zero Code

In the first part of this tutorial, we introduced you to the basics of Perl, Austin Powers-style. This week, we're going to get down and dirty with variables and operators, and also provide you with a brief introduction to Perl's conditional expressions.

To begin with, let's answer a very basic question for all those of you unfamiliar with programming jargon: what's a variable when it's at home?

A variable is the fundamental building block of any programming languages. Think of a variable as a container which can be used to store data; this data is used in different places in your Perl program. A variable can store both numeric and non-numeric data, and the contents of a variable can be altered during program execution. Finally, variables can be compared with each other, and you - the programmer - can write program code that performs specific actions on the basis of this comparison.

Every language has different types of variables - however, for the moment, we're going to concentrate on the simplest type, referred to in Perl as "scalar variables". A scalar variable can hold any type of value - integer, text string, floating-point number - and is usually preceded by a dollar sign.

The manner in which scalar variables are assigned values should be clear from the following example:

```
#!/usr/bin/perl

# a simple scalar variable
$name = "mud";

# and an example of how to use it
print ("My name is ", $name);
```

And here's what the output of that program looks like:

```
My name is mud
```

Really?! We feel for you...

It shouldn't be too hard to see what happened here - the scalar variable $name was assigned the text value "mud", and this value was then printed to the console via the print() statement.

Although assigning values to a scalar variable is extremely simple - as you've just seen - there are a few things that you should keep in mind here:

- A scalar variable name must be preceded by a dollar [$] sign - for example, $name, $id and the like. This helps differentiate between scalar variables and arrays or hashes, which are the other types of variables supported by Perl.
- Every scalar variable name must begin with a letter, optionally followed by more letters or numbers - for example, $a, $data123, $i_am_god
- The maximum length of a scalar variable name is 255 characters; however, if you use a name that long, you need therapy!
- Case is important when referring to scalar variables - in Perl, a $cigar is definitely not a $CIGAR!
- It's always a good idea to give your variables names that make sense and are immediately recognizable - it's easy to tell what $gross_income refers to, but not that easy to identify $ginc.

- Unlike other programming languages, a scalar variable in Perl can store both integers and floating-point numbers [also known as decimals]. This added flexibility is just one of the many nice things about Perl.

Here's another program, this one illustrating how variables can be used to store and manipulate numbers.

```
#!/usr/bin/perl

# declare a variable
$number = 19;

print ($number, " times 1 is ", $number,"\n");
print ($number, " times 2 is ", $number * 2, "\n");
print ($number, " times 3 is ", $number * 3, "\n");
print ($number, " times 4 is ", $number * 4, "\n");
print ($number, " times 5 is ", $number * 5, "\n");
```

And here's what the output looks like:

```
19 times 1 is 19
19 times 2 is 38
19 times 3 is 57
19 times 4 is 76
19 times 5 is 95
```

Perl 101 (Part 2) - Of Variables And Operators

By Vikram Vaswani and Harish Kamath
Melonfire
June 01, 2000

Q&A

Thus far, you've been assigning values to your variables at the time of writing the program itself [referred to in geek lingo as "design time"]. In the real world, however, many variables are assigned only after the program is executed [referred to as "run time"]. And so, this next program allows you to ask the user for input, assign this input to a variable, and then perform specific actions on that variable.

```
#!/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 try it out, you'll see something like this:

```
Gimme a number! 4
The square of 4 is 16
```

Let's go through this in detail. The first line of the program prints a prompt, asking the user to enter a number. Once the user enters a number, that input is assigned to the variable $number via the <STDIN> file handler. This particular file handler allows you to access data entered by the user at the command prompt, also referred to as STanDard INput.

At this point in time, the variable $number contains the data entered by you at the prompt, together with a newline [\n] character [caused by you hitting the Enter key]. Before the number can be processed, it is important that you remove the newline character, as leaving it in could adversely affect the rest of your program. Hence, chomp().

The chomp() function's sole purpose is to remove the newline character from the end of a variable, if it exists. Once that's taken care of, the number is multiplied by itself, and the result is displayed. Note our slightly modified usage of the print() function in this example - instead of using parentheses, we're simply printing a string and replacing the variables in it with their actual values.

Perl 101 (Part 2) - Of Variables And Operators

By Vikram Vaswani and Harish Kamath
Melonfire
June 01, 2000

2 + 2 ...

As you've already seen in last time's lesson, Perl comes with all the standard arithmetic operators - addition [+], subtraction [-], division [/] and multiplication [*] - and quite a few non-standard ones. Here's an example which demonstrates the important ones:

```
#!/usr/bin/perl

# get a number
print "Gimme a number! ";
$alpha = <STDIN>;

# get another number
print "Gimme another number! ";
$beta = <STDIN>;

# process input
chomp($alpha);
chomp($beta);

# standard stuff
$sum = $alpha + $beta;
$difference = $alpha - $beta;
$product = $alpha * $beta;
$quotient = $alpha / $beta;

# non-standard stuff
$remainder = $alpha % $beta;
$exponent = $alpha ** $beta;

# display the result
print "Sum: $sum\n";
print "Difference: $difference\n";
print "Product: $product\n";
print "Quotient from division: $quotient\n";
print "Remainder from division: $remainder\n";
print "Exponent: $exponent\n";
```

As with all other programming languages, division and multiplication take precedence over addition and subtraction, although parentheses can be used to give a particular operation greater precedence. For example,

```
#!/usr/bin/perl
print(10 + 2 * 4);
```

returns 18, while

```
#!/usr/bin/perl
print((10 + 2) * 4);
```

returns 48.

In addition to these operators, Perl also comes with the very useful auto-increment [++] and auto-decrement [--] operators, which you'll see a lot of in the next lesson. For the moment, all you need to know is that the auto-increment operator increments the value of the variable to which it is applied by 1, while the auto-decrement operator does exactly the same thing, but in the opposite direction. Here's an example:

```
#!/usr/bin/perl

# initial value
$a = 7;
print("Initial value: ", $a, "\n");

# increment and display
$a++;
print("After increment: ", $a, "\n");
```

Perl 101 (Part 2) - Of Variables And Operators

By Vikram Vaswani and Harish Kamath
Melonfire
June 01, 2000

... Or Two Plus Two

Of course, operators are not restricted to numbers alone - there are a couple of useful string operators as well. The string concatenation operator [.] is used to concatenate strings together, as in the following example:

```
#!/usr/bin/perl

$a = "The";
$b = "Incredible";
$c = "Shrinking";
$d = "Violet";

$result = $a . $b . $c . $d;
print "Here's one possibility - $result\n";

$result = $c . $a . $b . $d;
print "And here's another - $result\n";
```

And the string repetition operator[x] is used to repeat strings a specific number of times - take a look:

```
#!/usr/bin/perl

$a = "Loser!\n";
$insult = $a x 7;
print($insult);
```

Talk about using your powers for evil...

Perl 101 (Part 2) - Of Variables And Operators

By Vikram Vaswani and Harish Kamath
Melonfire
June 01, 2000

Comparing Apples And Oranges

In addition to the various arithmetic and string operators, Perl also comes with a bunch of comparison operators, whose sole raison d'etre is to evaluate expressions and determine if they are true or false. Here's a list - you should use these operators for numeric comparisons only.

Assume $x=4 and $y=10

Operator	What It Means	Expression	Result
==	is equal to	$x == $y	False
!=	is not equal to	$x != $y	True
>	is greater than	$x > $y	False
<	is less than	$x < $y	True
>=	is greater than or equal to	$x >= $y	False
<=	is less than or equal to	$x <= $y	True

If, however, you're planning to compare string values, the two most commonly used operators are the equality and inequality operators, as listed below.

Assume $x="abc", $y="xyz"

Operator	What It Means	Expression	Result
eq	is equal to	$x eq $y	False
ne	is not equal to	$x ne $y	True

You can also the greater- and less-than operators for string comparison - however, keep in mind that Perl uses the ASCII values of the strings to be compared when deciding which one is greater.

Assume $x="m", $y="M"

Operator	What It Means	Expression	Result
gt	is greater than	$x gt $y	True
lt	is less than	$x lt $y	False
ge	is greater than or equal to	$x ge $y	True
le	is less than or equal to	$x le $y	False

The reason for this - the ASCII value of "m" is greater than the ASCII value of "M".

Perl 101 (Part 2) - Of Variables And Operators

By Vikram Vaswani and Harish Kamath
Melonfire
June 01, 2000

Decisions! Decisions!

Why do you need to know all this? Well, comparison operators come in very useful when building conditional expressions - and conditional expressions come in very useful when adding control routines to your code. Control routines check for the existence of certain conditions, and execute appropriate program code depending on what they find.

The first - and simplest - decision-making routine is the "if" statement, which looks like this:

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

The "condition" here refers to a conditional expression, which evaluates to either true or false. For example,

```
if (bus is late)
          {
          con Dad into giving you a ride
          }
```

or, in Perl language

```
if (bus_arrival ==  0)
          {
          con_Dad();
          }
```

If the conditional expression evaluates as true, all statements within the curly braces are executed. If the conditional expression evaluates as false, all statements within the curly braces will be ignored, and the lines of code following the "if" block will be executed.

Here's a simple program that illustrates the basics of the "if" statement.

```
#!/usr/bin/perl

# ask for a number
print ("Gimme a number! ");
$alpha = ;
chomp ($alpha);

# ask for a different number
print ("Now gimme a different number! ");
$beta = ;
chomp ($beta);

# check
if ($alpha == $beta)
          {
          print("Can't you read, moron?
I need two *different* numbers!\n");
          }

print("This is the last line! Go away now!n");
```

And they say that the popular conception of programmers as rude, uncouth hooligans is untrue!

In addition to the "if" statement, Perl also allows you a couple of variants - the first is the "if-else" statement, which allows you to execute different blocks of code depending on whether the expression is evaluated as true or false.

The structure of an "if-else" statement looks like this:

```
if (condition)
          {
          do this!
```

```
        }
else
        {
        do this!
        }
```

In this case, if the conditional expression evaluates as false, all statements within the curly braces of the "else" block will be executed. Modifying the example above, we have

```
#!/usr/bin/perl

# ask for a number
print ("Gimme a number! ");
$alpha = <STDIN>;
chomp ($alpha);

# ask for a different number
print ("Now gimme a different number! ");
$beta = ;
chomp ($beta);

# check
if ($alpha == $beta)
        {
        print("Can't you read, moron?
I need two *different* numbers!\n");
        }
else
        {
        print("Finally! Someone with active brain cells!\n");
        }
```

And here's another example, this time using strings - the trigger condition here is the input string "yes".

```
#!/usr/bin/perl

# ask a question
print ("Did you like Austin Powers 2:
The Spy Who Shagged Me?\n");
$response = ;
chomp ($response);

# check
if ($response eq "yes")
        {
        print("Groovy, baby!\n");
        }
else
        {
        print("Loser! May your soul rot in hell!\n");
        }
```

Perl 101 (Part 2) - Of Variables And Operators

By Vikram Vaswani and Harish Kamath
Melonfire
June 01, 2000

Handling The Gray Areas

Both the "if" and "if-else" constructs are great for black-and-white, true-or-false situations. But what if you need to handle the gray areas as well?

Well, that's why we have the "if-elsif-else" construct:

```
if (first condition is true)
          {
          do this!
          }
elsif (second condition is true)
          {
          do this!
          }
elsif (third condition is true)
          {
          do this!
          }

          ... and so on ...

else
          {
          do this!
          }
```

You can have as many "elsif" blocks as you like in this kind of construct. The only rule is that the "elsif" blocks must come after the "if" block but before the "else" block.

```
#!/usr/bin/perl

# movie quote generator

# set up the choices
print("[1] The Godfather\n");
print("[2] American Beauty\n");
print("[3] The Matrix\n");
print("[4] The Usual Suspects\n");
print("[5] Casino\n");
print("[6] Star Wars\n");
print("Gimme a number, and I'll give you a quote!\n");

# get some input
$choice = ;
chomp($choice);

# check input and display
if ($choice == 1)
          {
        print("I'll make him an offer he can't refuse.\n");
        }
elsif ($choice == 2)
          {
        print("It's okay. I wouldn't remember me
either.\n");
        }
elsif ($choice == 3)
        {
        print("Unfortunately, no one can be told what the
Matrix is. You
have to see it for yourself. \n");
        }
elsif ($choice == 4)
        {
        print("The greatest trick the devil ever
pulled was convincing the
world he didn't exist.\n");
        }
elsif ($choice == 5)
        {
        print("In the casino, the cardinal rule is to keep
them playing and
to keep them coming back.\n");
        }
elsif ($choice == 6)
        {
        print("I suggest a new strategy, Artoo:
let the Wookie win.\n");
        }
else
        '
```

```
}
print("Invalid choice!\n");
}
```

Depending on the data you input at the prompt, the "elsif" clauses are scanned for an appropriate match and the correct quote is displayed. In case the number you enter does not correspond with the available choices, the program goes to the "else" routine and displays an error message.

Perl 101 (Part 2) - Of Variables And Operators

By Vikram Vaswani and Harish Kamath
Melonfire
June 01, 2000

Miscellaneous Notes

Before you go, here are a couple of things that you might find interesting:

chomp() versus chop()

In addition to the chomp() function used above, Perl also has a chop() function. While the chomp() function is used to remove the trailing newline if it exists, the chop() function is designed to remove the last character of a variable, irrespective of whether or not it is a newline character.

```
#!/usr/bin/perl

# set up the variables
$sacrificial_goat1 = "boom";
$sacrificial_goat2 = "boom";

# chomp it!
chomp($sacrificial_goat1);
print($sacrificial_goat1, "\n");

# chop it!
chop($sacrificial_goat2);
print($sacrificial_goat2, "\n");
```

Assignment operators versus equality operators

An important point to note - and one which many novice programmers fall foul of - is the difference between the assignment operator [=] and the equality operator [==]. The former is used to assign a value to a variable, while the latter is used to test for equality in a conditional expression.

So

```
$a = 47;
```

assigns the value 47 to the variable $a, while

```
$a == 47
```

tests whether the value of $a is equal to 47.

Special characters and print()

As you've seen, print() can be used in either one of two ways:

```
#!/usr/bin/perl

$day = "Tuesday";
print("Today is ", $day);
```

or

```
#!/usr/bin/perl

$day = "Tuesday";
print "Today is $day";
```

both of which are equivalent, and return this output:

```
Today is Tuesday
```

But now try replacing the double quotes with singles, and watch what happens:

```
#!/usr/bin/perl

$day = "Tuesday";
print 'Today is $day';
```

Your output should now read

```
Today is $day
```

Thus, single quotes turn off Perl's "variable interpolation" - simply, the ability to replace variables with their actual value when executing a program. This also applies to special characters like the newline character - single quotes will cause Perl to print the newline character as part of the string, while double quotes will allow it to recognize the character correctly.

You should note this difference in behaviour, if only to save yourself a few minutes of debugging time.

The second thing to note about print() is that you need to "escape" special characters with a backslash. Take a look at this example:

```
#!/usr/bin/perl

print("She said "Hello" to me, and my
heart skipped a beat.");
```

When you run this, you'll see a series of error messages - this is because the multiple sets of double quotes within the print() function call confuse Perl. And so, if you'd like your output to contain double quotes [or other special characters], it's necessary to escape them with a preceding backslash.

```
#!/usr/bin/perl

print("She said \"Hello\" to me, and my
heart skipped a beat.");
```

As to what happens next - you'll have to wait for the next lesson in this series, when we'll be teaching you a few more control structures and introducing you to the different types of loops supported by Perl. See you then!

Perl 101 (Part 3) - Looping The Loop

By Vikram Vaswani and Harish Kamath
Melonfire
June 15, 2000

Shampoo And Perl

Perl 101 Article Series:

- Perl 101 (Part 1) - The Basics
- Perl 101 (Part 2) - Of Variables And Operators
- Perl 101 (Part 3) - Looping The Loop
- Perl 101 (Part 4) - Mind Games
- Perl 101 (Part 5) - Sub-Zero Code

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...

Perl 101 (Part 3) - Looping The Loop

By Vikram Vaswani and Harish Kamath
Melonfire
June 15, 2000

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.

Perl 101 (Part 3) - Looping The Loop

By Vikram Vaswani and Harish Kamath
Melonfire
June 15, 2000

...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...

This

Perl 101 (Part 3) - Looping The Loop

By Vikram Vaswani and Harish Kamath
Melonfire
June 15, 2000

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.

Perl 101 (Part 3) - Looping The Loop

By Vikram Vaswani and Harish Kamath
Melonfire
June 15, 2000

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");
```

Perl 101 (Part 4) - Mind Games

By Vikram Vaswani and Harish Kamath
Melonfire
June 29, 2000

The Lotus Position

Perl 101 Article Series:

- Perl 101 (Part 1) - The Basics
- Perl 101 (Part 2) - Of Variables And Operators
- Perl 101 (Part 3) - Looping The Loop
- Perl 101 (Part 4) - Mind Games
- Perl 101 (Part 5) - Sub-Zero Code

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...

Perl 101 (Part 4) - Mind Games

By Vikram Vaswani and Harish Kamath
Melonfire
June 29, 2000

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!

Perl 101 (Part 4) - Mind Games

By Vikram Vaswani and Harish Kamath
Melonfire
June 29, 2000

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.

Perl 101 (Part 4) - Mind Games

By Vikram Vaswani and Harish Kamath
Melonfire
June 29, 2000

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");
```

Perl 101 (Part 4) - Mind Games

By Vikram Vaswani and Harish Kamath
Melonfire
June 29, 2000

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");
```

Perl 101 (Part 4) - Mind Games

By Vikram Vaswani and Harish Kamath
Melonfire
June 29, 2000

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);
```

Perl 101 (Part 4) - Mind Games

By Vikram Vaswani and Harish Kamath
Melonfire
June 29, 2000

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";
```

```
                    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.
```

Perl 101 (Part 4) - Mind Games

By Vikram Vaswani and Harish Kamath
Melonfire
June 29, 2000

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.

Perl 101 (Part 4) - Mind Games

By Vikram Vaswani and Harish Kamath
Melonfire
June 29, 2000

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.

Perl 101 (Part 4) - Mind Games

By Vikram Vaswani and Harish Kamath
Melonfire
June 29, 2000

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?");
```

Perl 101 (Part 4) - Mind Games

By Vikram Vaswani and Harish Kamath
Melonfire
June 29, 2000

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.

Perl 101 (Part 4) - Mind Games

By Vikram Vaswani and Harish Kamath
Melonfire
June 29, 2000

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!

Perl 101 (Part 5) - Sub-Zero Code

By Vikram Vaswani and Harish Kamath
Melonfire
July 21, 2000

A Little Knowledge

Perl 101 Article Series:

- Perl 101 (Part 1) - The Basics
- Perl 101 (Part 2) - Of Variables And Operators
- Perl 101 (Part 3) - Looping The Loop
- Perl 101 (Part 4) - Mind Games
- Perl 101 (Part 5) - Sub-Zero Code

If you've been paying attention these last few weeks, you should know enough about Perl to begin writing your own programs in the language. As a matter of fact, you might even be entertaining thoughts of cutting down your weekly visits to this Web site and doing away with this tutorial altogether.

Well, a very wise man once said that a little knowledge was a dangerous thing...and so, as your Perl scripts become more and more complex, you're going to bump your head against the principles of software design, and begin looking for a more efficient way of structuring your Perl programs.

Enter this week's tutorial, which attempts to address that very problem by teaching you all you need to know about a programming construct called a "subroutine". And - since all work and no play is no way to live your life - we'll also be tossing in a few pop culture references that should help make the lesson an interesting [maybe even entertaining?] one.

Perl 101 (Part 5) - Sub-Zero Code

By Vikram Vaswani and Harish Kamath
Melonfire
July 21, 2000

Great Movies...

Ask a geek to define the term "subroutine", and he'll probably tell you that a subroutine is "a block of statements that can be grouped together as a named entity." Since this definition raises more questions than answers [the primary one being, what on earth are you doing hanging around with geeks in the first place], we'll simplify things by informing you that a subroutine is simply a set of program statements which perform a specific task, and which can be called, or executed, from anywhere in your Perl program.

There are two important reasons why subroutines are a "good thing". First, a subroutine allows you to separate your code into easily identifiable subsections, thereby making it easier to understand and debug. And second, a subroutine makes your program modular by allowing you to write a piece of code once and then re-use it multiple times within the same program.

Let's take a simple example, which demonstrates how to define a sub-routine and call it from different places within your Perl script:

```
#!/usr/bin/perl

# define a subroutine
sub greatest_movie
{
print "Star Wars\n";
}

# main program begins here
print "Question: which is the greatest movie of all time?\n";

# call the subroutine
&greatest_movie;

# ask another question
print "Question: which movie introduced the world to Luke Skywalker, Yoda
and Darth Vader?\n";

# call the subroutine
&greatest_movie;
```

Now run it - you should see something like this:

```
Question: which is the greatest movie of all time?
Star Wars
Question: which movie introduced the world to Luke Skywalker, Yoda and
Darth Vader?
Star Wars
```

Let's take this line by line. The first thing we've done in our Perl program is define a new subroutine with the "sub" keyword; this keyword is followed by the name of the subroutine. All the program code attached to that subroutine is then placed within a pair of curly braces - this program code could contain loops, conditional statements, calls to other subroutines, or calls to other Perl functions. In the example above, our subroutine has been named "greatest_movie", and only contains a call to Perl's print() function.

Here's the typical format for a subroutine:

```
sub subroutine_name
{
   statement 1...
   statement 2...
   .
   .
   .
   statement n...
}
```

Perl 101 (Part 5) - Sub-Zero Code

By Vikram Vaswani and Harish Kamath
Melonfire
July 21, 2000

...And Memorable Friends

Of course, defining a subroutine is only half of the puzzle - for it to be of any use at all, you need to invoke it. In Perl, this is accomplished by calling the subroutine by its name, as we've done in the last line of the example above. When invoking a subroutine in this manner, it's usually a good idea to precede the name with an ampersand [&]- this is not essential, though, and the following would also work:

```
#!/usr/bin/perl

# define a subroutine
sub greatest_movie
{
print "Star Wars\n";
}

# main program begins here
print "Question: which is the greatest movie of all time?\n";

# call the subroutine
greatest_movie;

# ask another question
print "Question: which movie introduced the world to Luke Skywalker, Yoda
and Darth Vader?\n";

# call the subroutine
greatest_movie;
```

However, it's good programming practice to precede the name of a Perl subroutine with an ampersand when invoking it - this helps to differentiate the Perl subroutine from array or scalar variables that may have the same name, and from pre-defined Perl functions. For example, consider this piece of code, in which we've defined a subroutine called &print, which also happens to be the name of the in-built Perl print () function:

```
#!/usr/bin/perl

# define a subroutine
sub print
{
print "Ross\n";
}

# main program begins here
print "Question: which Friend once had a pet monkey?\n";

# call the subroutine
print;
```

In this case, when you run the program, you'll get the following:

```
Question: which Friend once had a pet monkey?
```

Here, Perl assumes that the line containing the print statement is a call to the in-built Perl print() function, rather than the user-defined Perl &print subroutine. To avoid this kind of error, you should either avoid naming your subroutines after reserved words, or precede the subroutine call with an ampersand. In the example above, if you changed the last line from

```
print;
```

to

```
&print;
```

Perl would understand the subroutine call correctly, and display the desired output.

```
Question: which Friend once had a pet monkey?
Ross
```

Perl 101 (Part 5) - Sub-Zero Code

By Vikram Vaswani and Harish Kamath
Melonfire
July 21, 2000

Popping The Question

Usually, when a subroutine is invoked in Perl, it generates a "return value". This return value is either the value of the last expression evaluated within the subroutine, or a value explicitly returned via the "return" statement. We'll examine both these a little further down - but first, here's a quick example of how a return value works.

```
#!/usr/bin/perl

# define a subroutine
sub change_temp
{
$celsius = 35;
$fahrenheit = ($celsius * 1.8) + 32;
}

# assign return value to variable
$result = &change_temp;

print "35 Celsius is $result Fahrenheit\n";
```

In this case, the value of the last expression evaluated within the subroutine serves as its return value - this value is then assigned to the variable $result when the subroutine is invoked from within the program.

Of course, it's also possible to explicitly specify a return value - use the "return" statement, as we've done in the next example:

```
#!/usr/bin/perl

# define a subroutine
sub do_you
{
if ($tall == 1 && $dark == 1 && $handsome == 1)
        {
        return "Yes!\n";
        }
else
        {
        return "Nope, afraid I don't feel the same way about you!\n";
        }
}

$tall = 1;
$dark = 1;
$handsome = 1;

# pop the question
print "Will you marry me?\n";

# assign return value to variable
$answer = &do_you;

# print the answer
print $answer;
```

Perl 101 (Part 5) - Sub-Zero Code

By Vikram Vaswani and Harish Kamath
Melonfire
July 21, 2000

Jumping Cows And Extra-Large Pumpkins

Return values from a subroutine can even be substituted for variables anywhere in a program. For example, you could modify the last two lines of the example above to read:

```
#!/usr/bin/perl

# define a subroutine
sub do_you
{
if ($tall == 1 && $dark == 1 && $handsome == 1)
        {
        return "Yes!\n";
        }
else
        {
        return "Nope, afraid I don't feel the same way about you!\n";
        }
}

$tall = 1;
$dark = 1;
$handsome = 1;

# pop the question
print "Will you marry me?\n";

# assign return value to variable and print
print(&do_you);
```

And, of course, return values need not be scalar variables alone - a subroutine can just as easily return an array variable, as we've demonstrated in the following example:

```
#!/usr/bin/perl

# define a subroutine
sub split_me
{
split(" ", $string);
}

# define string
$string = "The cow jumped over the moon and turned into a gigantic pumpkin";

# invoke function and assign result to array
@words = &split_me;

# loop for each element of array
foreach $word (@words)
        {
        print "Word: $word\n";
        $count++;
        }

# print total
print "The number of words in the given string is $count\n";
```

The output is

```
Word: The
Word: cow
Word: jumped
Word: over
Word: the
Word: moon
Word: and
Word: turned
Word: into
Word: a
Word: gigantic
Word: pumpkin
The number of words in the given string is 12
```

Perl 101 (Part 5) - Sub-Zero Code

By Vikram Vaswani and Harish Kamath
Melonfire
July 21, 2000

Turning Up The Heat

Now, if you've been paying attention, you've seen how subroutines can help you segregate blocks of code, and use the same piece of code over and over again, thereby eliminating unnecessary duplication. But this is just the tip of the iceberg...

The subroutines you've seen thus far are largely static, in that the variables they use are already defined. But it's also possible to pass variables to a subroutine from the main program - these variables are called "arguments", and they add a whole new level of power and flexibility to your code.

Consider the following simple example:

```
#!/usr/bin/perl

# define a subroutine
sub add_two_numbers
{
$sum = $_[0] + $_[1];
return $sum;
}

$total = &add_two_numbers(3,5);
print "The sum of the numbers is $total\n";
```

A few words of explanation here:

You're already familiar with the special Perl array @ARGV, which contains parameters passed to the Perl program on the command line. Well, Perl also has a variable named @_, which contains arguments passed to a subroutine, and which is available to the subroutine when it is invoked. The value of each element of the array can be accessed using standard scalar notation - $_[0] for the first element, $_[1] for the second element, and so on.

In the example above, once the &add_two_numbers subroutine is invoked with the numbers 3 and 5, the numbers are transferred to the @_ variable, and are then accessed using standard scalar notation within the subroutine. Once the addition has been performed, the result is returned to the main program, and displayed on the screen via the print() statement.

Note the manner in which arguments are passed to the subroutine, enclosed within a pair of parentheses.

How about something a little more useful? Let's go back a couple of pages, and consider the &change_temp subroutine we've defined:

```
#!/usr/bin/perl

# define a subroutine
sub change_temp
{
$celsius = 35;
$fahrenheit = ($celsius * 1.8) + 32;
}

# assign return value to variable
$result = &change_temp;

print "35 Celsius is $result Fahrenheit\n";
```

Now, suppose we alter this to accept the temperature in Celsius from the main program, and return the temperature in Fahrenheit.

```
#!/usr/bin/perl

# define a subroutine
sub change_temp
{
$fahrenheit = ($_[0] * 1.8) + 32;
}
```

```
print "Enter temperature in Celsius\n";
$temperature = ;
chomp ($temperature);

$result = &change_temp($temperature);

print "$temperature Celsius is $result Fahrenheit\n";
```

And here's what it would look like:

```
Enter temperature in Celsius
45
45 Celsius is 113 Fahrenheit
```

Take it one step further - how about allowing the user to specify the temperature to be converted on the command line itself?

```
#!/usr/bin/perl

# define a subroutine
sub change_temp
{
$fahrenheit = ($_[0] * 1.8) + 32;
}

# get the command-line parameters
# and pass them to the subroutine
# and assign the result
$result = &change_temp(@ARGV);

# print the result
print "$ARGV[0] Celsius is $result Fahrenheit\n";
```

If you saved this program as "convert_temp.pl", and ran it like this

```
$ convert_temp.pl 35
```

you'd see

```
35 Celsius is 95 Fahrenheit
```

The above example also neatly demonstrates the relationship between @ARGV and @_ - the temperature entered on the command line first goes into the @ARGV variable, and is then passed to the subroutine via the @_ variable. Remember that the @_ variable is only available within the scope of a specific subroutine.

Perl 101 (Part 5) - Sub-Zero Code

By Vikram Vaswani and Harish Kamath
Melonfire
July 21, 2000

My() Hero!

Let's now talk a little bit about the variables used within a subroutine, and their relationship with variables in the main program. Unless you specify otherwise, the variables used within a subroutine are global - that is, the values assigned to them are available throughout the program, and changes made to them during subroutine execution are not restricted to the subroutine space alone.

For a clearer example of what this means, consider this simple example:

```
#!/usr/bin/perl

# define a subroutine
sub change_value
{
$hero = "Wolverine";
}

# define a variable
$hero = "The Incredible Hulk";

# before invoking subroutine
print "Today's superhero is $hero\n";
print "Actually, I've changed my mind...";
&change_value;

# after invoking subroutine
print "...gimme $hero instead.\n";
```

And here's what you'll see:

```
Today's superhero is The Incredible Hulk
Actually, I've changed my mind.....gimme Wolverine instead.
```

Obviously, this is not always what you want - there are numerous situations where you'd prefer the variables within a subroutine to remain "private", and not disturb the variables within the main program. And this is precisely the reason for Perl's my() construct.

The my() construct allows you to define variables whose influence does not extend outside the scope of the subroutine within which they are enclosed. Take a look:

```
#!/usr/bin/perl

# define a subroutine
sub change_value
{
# this statement added
my ($hero);
$hero = "Wolverine";
}

# define a variable
$hero = "The Incredible Hulk";

# before invoking subroutine
print "Today's superhero is $hero\n";
print "Actually, I've changed my mind...";
&change_value;

# after invoking subroutine
print "...gimme $hero instead.\n";
```

And here's what you'll get:

```
Today's superhero is The Incredible Hulk
Actually, I've changed my mind.....gimme The Incredible Hulk instead.
```

What happens here? Well, when you define a variable with the "my" keyword, Perl first checks to see if a variable already exists with the same name. If it does [as in the example above], its value is stored and a

variable already exists with the same name. If it does [as in the example above], its value is stored and a new variable is created for the duration of the subroutine. Once the subroutine has completed its task, this new variable is destroyed and the previous value of the variable is restored.

The my() operator can be used with both scalar and array variables. And - since Perl is all about efficiency - you can assign a value to the variable at the same that that you declare it, like this:

```
sub change_value
{
my ($hero) = "Wolverine";
}
```

Perl 101 (Part 5) - Sub-Zero Code

By Vikram Vaswani and Harish Kamath
Melonfire
July 21, 2000

The Age Gauge

So Perl gives you "public" variables and "private" variables - more than enough for most programmers. But you know what geeks are like...they're never satisfied. And so Perl also provides a useful middle ground - variables which are available between subroutines, but are hidden from the main program.

Why would you want to use something like this? Well, consider the following example, which demonstrates the concept:

```
#!/usr/bin/perl

# define some subroutines
sub display_value
{
print "During the subroutine...you are $age years old.\n";
}

sub change_age
{
local ($age) = $age + $increment;
&display_value($age);
}

# ask for age
print "How old are you?\n";
$age = ;
chomp ($age);

# ask for increment
print "How many years would you like to add?\n";
$increment = ;
chomp ($increment);

# demonstrate local variable
print "Before invoking the subroutine...you are $age years old.\n";
&change_age;
print "After invoking the subroutine...you are $age years old.\n";
```

And here's what it looks like:

```
How old are you?
32
How many years would you like to add?
9
Before invoking the subroutine...you are 32 years old.
During the subroutine...you are 41 years old.
After invoking the subroutine...you are 32 years old.
```

When making calls between subroutines in this manner, it often becomes necessary to store the value of a variable across subroutines - and that's where local() comes in. In the example above, the variable $age is assigned an initial [global] value on the basis of user input. However, once the &change_age subroutine is invoked, this global value is stored and a new value is assigned to $age.

So far so good...we've already seen this with my(). But now, &change_age needs to call &display_value, and pass it the value of the variable $age. By declaring $age to be a "local" variable, Perl makes it possible for the &display_value subroutine to access the new value of $age, and display it.

Once the subroutines finish and return control to the main program, the original value of $age is restored, and displayed. Thus, the example demonstrates how the local() keyword can be used to share variable values between subroutines, without affecting the global value of the variable.

And that's about it for this week. Next time, we'll be taking a close look at some of Perl's built-in string, math and pattern-recognition functions...so make sure you come back!

Perl 101 (part 6) - The Perl Toolbox

By Vikram Vaswani and Harish Kamath
Melonfire
August 30, 2000

Stringing Things Along

With the arcane mysteries of Perl subroutines behind us, it's now time to move on to the many practical uses of Perl. Over the next few pages, we're going to spend some time investigating Perl's many built-in functions, functions that come in very handy when you need to do a little string manipulation.

Among the items on today's agenda: pattern matching and replacement, a close look at some useful string functions, and a quick trip back in time to math class with Perl's math functions. Pay attention, now!

Perl 101 (part 6) - The Perl Toolbox

By Vikram Vaswani and Harish Kamath
Melonfire
August 30, 2000

Expressing Yourself

One of Perl's most powerful features is the ability to do weird and wonderful things with "regular expressions". To the uninitiated, regular expressions, or "regex", are patterns which are built using a set of special characters; these patterns can then be compared with text in a file, or data entered into a Web form. A pattern match can trigger some kind of action...or not, as the case may be.

Though regular expressions can get a little confusing when you're first starting out with them, a little patience will reap rich rewards, as they can save you a tremendous amount of work. Our very first example illustrates a simple pattern-matching operation, and introduces you to Perl's match operator:

```
#!/usr/bin/perl

# get a line of input
print "Gimme a line!\n";
$line = ;
chomp ($line);

# get a search term
print "Gimme the string to match!\n";
$term = ;
chomp ($term);

# check for match
if ($line =~ /$term/)
{
print "Match found!\n";
}
else
{
print "No match found\n";
}
```

A quick explanation is in order here. In Perl, the "pattern" is the sequence of characters to be matched - this pattern is usually enclosed within a pair of slashes. For example,

/xyz/ represents the pattern "xyz".

The example above asks for a line of text and a search term - this search pattern is then used with the match operator =~ to test for a match. The result of the =~ operation is true if the pattern is found in the string, and false if not.

Here's the output of the example above:

```
Gimme a line!
I'll be back
Gimme the string to match!
b
Match found!
```

Perl also has the !~ operator, which does the reverse of the =~ operator - it returns true if a match is NOT found.

Perl 101 (part 6) - The Perl Toolbox

By Vikram Vaswani and Harish Kamath
Melonfire
August 30, 2000

Engry Young Men

In addition to simple matching, Perl also allows you to perform substitutions.

Take a look at our next example, which prompts you to enter a line of text, and then replaces all occurrences of the letter "a" with the letter "e".

```
#!/usr/bin/perl

# get a line of input
print "Gimme a line!\n";
$line = ;
chomp ($line);

# substitute with the substitution operator
$line =~ s/a/e/;

# and print
print $line;
```

In this case, we've used Perl's substitution operator - it looks like this:

```
s/search-pattern/replacement-pattern/
```

In the example above, the line

```
$line =~ s/a/e/;
```

simply means "substitute a with e in the scalar variable $line".

And here's the output:

```
Gimme a line!
angry young man

engry young man
```

Ummm...didn't work quite as advertised, did it? All it did was replace the first occurrence of the letter "a". How about adding the "g" operator, which does a global search-and-replace?

```
#!/usr/bin/perl

# get a line of input
print "Gimme a line!\n";
$line = ;
chomp ($line);

# substitute with the substitution operator
$line =~ s/a/e/g;

# and print
print $line;
```

And this time,

```
angry young man
```

is replaced with

```
engry young men
```

Much better! But what if your sentence contains an upper-case "a", also known as "A". Well, that's why Perl also has the case-insensitivity operator "i", which takes care of that last niggling problem:

```
#!/usr/bin/perl

# get a line of input
print "Gimme a line!\n";
$line = ;
chomp ($line);

# substitute with the substitution operator
$line =~ s/a/e/gi;

# and print
print $line;
```

And the output:

```
Gimme a line!
Angry young man
Engry young men
```

Of course, we're just scratching the tip of the regex iceberg here. Things get even more interesting when you start using patterns and metacharacters instead of actual words...

Perl 101 (part 6) - The Perl Toolbox

By Vikram Vaswani and Harish Kamath
Melonfire
August 30, 2000

Aardvark, Anyone?

The example you just saw was pretty cut-and-dried - decide search term, decide replacement term, shove both into Perl script, bang bang and Bob's your uncle. But what happens if you need to replace a class of words, rather than a single word? If, for example, you need to replace not just the letter "a", but words beginning with one or more "a" and ending with a "k"...like "aardvark", for example?

The special characters you'll use to modify your pattern are called "metacharacters", which is another of those words that sounds impressive but means absolutely nothing useful. We'll explain some of them here, and point you to a resource for more information a little further down.

Two of the more useful metacharacters in Perl are the "+" and "*" characters, which match "one or more" instances and "zero or more" instances of the preceding pattern respectively.

For example,

```
/boo+/
```

would match "book", and "booo" but not "bottle", while

```
/bo*/
```

would match all of "bottle", "bog", "book", and "booo". One common use of Perl pattern-matching is to remove unnecessary blank spaces from a block of text. This is accomplished using the "\s" metacharacter, which matches whitespace, tab stops and newline characters. The simple substitution

```
/\s+/ /
```

would "substitute zero or more occurrences of whitespace with nothing"

Perl also has two important "anchor" characters, which allow you to build patterns which specify which characters come at the beginning or end of a string. The ^ character is used to match the beginning of a string, while the $ character is used to match the end. So,

```
/^h/
```

would match "hello" and "house", but not "shop"

while

```
/g$/
```

would match "dig" and "bog", but not "gold" or "eagle"

And you can even specify more than one pattern to match with the | operator.

```
/(the|a)/
```

would return true if the string contained either "the" or "a".

Obviously, regular expressions is a topic in itself - if you're interested, we've put together a well-written, comprehensive guide to the topic at http://www.devshed.com/Server_Side/Administration/RegExp/

Perl 101 (part 6) - The Perl Toolbox

By Vikram Vaswani and Harish Kamath
Melonfire
August 30, 2000

Needles In Haystacks

Perl comes with a wide variety of functions that come in handy when manipulating strings. The first one on our list is the length() function, which returns the length of a specified string.

Here's an example of how the length() function can be used to restrict the length of a login name to between six and ten characters:

```
#!/usr/bin/perl

do
{

# ask for a name
      print ("Please enter a username:");
$username = ;
chomp($username);

# and repeat until the username is between 6 and 10 characters long
} while ((length($username) < 6) ||  (length($username) > 10));


print "A new star is born...and its name is $username!\n";
```

In this case, each time a username is entered, we use the length() function to count the number of characters in it. If this number is less than 6, or greater than 10, the loop is repeated until a username of the correct length is entered.

Here's what it looks like:

```
Please enter a username:me
Please enter a username:galapaloozy
Please enter a username:godzilla
A new star is born...and its name is godzilla!
```

The next string function that we're going to unravel is the index() function. This function is typically used to find out if a particular pattern exists within a string. Here's what it looks like:

```
$var = index(string, pattern)
```

where "string" is the string to be searched for "pattern". If the pattern is found within the string, the position of the first character of the matched pattern will be assigned to the variable $var; if not, $var will be assigned the value -1.

Here's a quick example:

```
#!/usr/bin/perl

# how to find a needle in a haystack
# Perl-style

print "THE HAYSTACK\n";
print "------------\n";

# set up the string
$haystack = "211643831 923465971315874643 13729247620352625
9235923595232305232095 8284529 2347392901847
32393482562502925352395327202358";
print $haystack . "\n";

# ask for a search term
print "Gimme a needle: ";
$needle = ;
chomp ($needle);

# use index() to look for the string
$location = index($haystack, $needle);

# print appropriate message
if ($location >= 0)
```

```
if ($location >= 0)
{
print "Needle located $location characters deep in the haystack\n";
}
else
{
print "Sorry, this haystack contains no needle\n";
}
```

And here's the output:

```
THE HAYSTACK
------------
211643831 923465971315874643 13729247620352625 9235923595232305232095
8284529
2347392901847 32393482562502925352395327202358
Gimme a needle: 1267
Sorry, this haystack contains no needle

THE HAYSTACK
------------
211643831 923465971315874643 13729247620352625 9235923595232305232095
8284529
2347392901847 32393482562502925352395327202358
Gimme a needle: 6250
Needle located 101 characters deep in the haystack
```

In this case, we've set up a string containing a set of random numbers. The user is then asked to enter a number of his own choice, and the index() function is used to scan the string for the number. Depending on the result, an appropriate message is printed.

Similar to the index() function is the rindex() function, which also searches for a pattern within the specified string, but starts from the end.

Perl 101 (part 6) - The Perl Toolbox

By Vikram Vaswani and Harish Kamath
Melonfire
August 30, 2000

Slice And Dice

Next up, the substr() function. As the name implies, this is the function that allows you to slice and dice strings into smaller strings. Here's what it looks like:

```
substr(string, start, length)
```

where "string" is a string or a scalar variable containing a string, "start" is the position to begin slicing at, and "length" is the number of characters to return from "start".

Here's a Perl script that demonstrates the substr() operator:

```
#!/usr/bin/perl

# get a string
print "Gimme a line!\n";
$line = ;
chomp ($line);

# get a chunk size
print "How many characters per slice?\n";
$num_slices = ;
chomp ($num_slices);

$length = length($line);
$count = 0;

print "Slicing...\n";

# slice the string into sections
while (($num_slices*$count) < $length)
{
$temp = substr($line, ($num_slices*$count), $num_slices);
$count++;
print $temp . "\n";
}
```

Here, after getting a string and a block size, we've used a "while" loop and a counter to keep slicing off pieces of the string and displaying them on separate lines.

And here's what it looks like:

```
Gimme a line!
The cow jumped over the moon, giggling madly as a purple pumpkin with fat
ears exploded into confetti
How many characters per slice?
11
Slicing...
The cow jum
ped over th
e moon, gig
gling madly
 as a purpl
e pumpkin w
ith fat ear
s exploded
into confet
ti
```

You've already used the print() function extensively to sent output to the console. However, the print() function doesn't allow you to format output in any significant manner - for example, you can't write 1000 as 1,000 or 1 as 00001. And so clever Perl programmers came up with the printf() function, which allows you to define the format in which data is printed to the console.

Consider a simple example - printing decimals:

```
#!/usr/bin/perl

print (5/3);
```

And here's the output:

```
1.66666666666667
```

As you might imagine, that's not very friendly. Ideally, you'd like to display just the "significant digits" of the result. And so, you'd use the printf() function:

```
#!/usr/bin/perl

printf "%1.2f", (5/3);
```

which returns

```
1.67
```

A quick word of explanation here: the Perl printf() function is very similar to the printf() function that C programmers are used to. In order to format the output, you need to use "field templates", templates which represent the format you'd like to display.

Some common field templates are:

```
%s string
%c character
%d decimal number
%x hexadecimal number
%o octal number
%f float number
```

You can also combine these field templates with numbers which indicate the number of digits to display - for example, %1.2f implies that Perl should only display two digits after the decimal point.

Here are a few more examples of printf() in action:

```
printf("%05d", 3); # returns 00003
printf("$%2.2f", 25.99); # returns $25.99
printf("%2d%", 56); # returns 56%
```

And here's a calculator which uses the printf() function to display numbers in various numerical bases like hexadecimal and octal.

```
#!/usr/bin/perl

print "Enter a number: ";
chomp($number = );

printf("In decimal format: %d\n",$number);
printf("In hexadecimal format: %x\n",$number);
printf("In octal format: %o\n",$number);
```

Perl also comes with a sprintf() function, which is used to send the formatted output to a variable instead of standard output.

Perl 101 (part 6) - The Perl Toolbox

By Vikram Vaswani and Harish Kamath
Melonfire
August 30, 2000

Going Backwards

The next few string functions come in very handy when adjusting the case of a text string from lower- to upper-case, or vice-versa:

```
lc($string) - convert $string to lower case
uc($string) - convert $string to upper case
lcfirst($string) - convert the first character of $string to lower case
ucfirst($string) - convert the first character of $string to upper case
```

Here's an example:

```
#!/usr/bin/perl

# get a string
print "Say something: ";
chomp($string = );

# convert case
$output= lc($string);
print "All lower case: $output\n";

$output= uc($string);
print "All upper case: $output\n";

$output= lcfirst($string);
print "Look at the first character: $output\n";

$output =  ucfirst($string);
print "Look at the first character: $output\n";
```

And here's the output:

```
Say something: Something's rotten in the state of Denmark
All lower case: something's rotten in the state of denmark
All upper case: SOMETHING'S ROTTEN IN THE STATE OF DENMARK
Look at the first character: something's rotten in the state of Denmark
Look at the first character: Something's rotten in the state of Denmark
```

The reverse() function is used to reverse the contents of a particular string.

```
#!/usr/bin/perl

# ask for input
print "Say something: ";
$something = ;
chomp ($something);

# reverse and print
$gnithemos = reverse($something);
print "Sorry, you seem to be talking backwards - what does $gnithemos
mean?";
```

Here's the output:

```
Say something: God, I'm good
Sorry, you seem to be talking backwards - what does doog m'I ,doG mean?
```

And the chr() and ord() functions come in handy when converting from ASCII codes to characters and vice-versa. For example,

```
print chr(65);

returns

A

while
```

```
print ord("a");
```

returns

```
97
```

Perl 101 (part 6) - The Perl Toolbox

By Vikram Vaswani and Harish Kamath
Melonfire
August 30, 2000

Math Class

And finally, Perl also comes with a set of math functions that allow you to carry out complex mathematical operations. You probably won't need these, but you should at least know of their existence.

Sine of a angle: sin($radians)

Cosine of a angle: cos($radians)

Square root of a number: sqrt($variable)

Exponent of a number: exp($variable)

Natural logarithm of a number: log($variable)

Absolute value of a number: abs($variable)

Decimal value of a number from hexadecimal: hex($variable)

Decimal value of a number from octal: oct($variable)

Integer portion of a number: int($variable)

And here's an example that demonstrates all these:

```
#!/usr/bin/perl

# set up the choices
print "Pick from the choices below:\n";
print "Sine of an angle[1]\n";
print "Cosine of an angle[2]\n";
print "Square root of a number[3]\n";
print "Exponent of a number[4]\n";
print "Natural logarithm of a number[5]\n";
print "Absolute value of a number[6]\n";
print "Decimal value of a number from hexadecimal[7]\n";
print "Decimal value of a number from octal[8]\n";
print "Integer value[9]\n";

chomp($choice = );

# and process them
if ($choice == 1 || $choice == 2)
        {
        print "Enter the angle in radians: ";
        chomp($angle = );

        if($choice == 1)
                {
                $value = sin($angle);
                print("Sine of $angle is $value\n");
                }
        else
                {
                $value = cos($angle);
                print("Cosine of $angle is $value\n");
                }
        }
elsif($choice == 3)
        {
        print "Enter a positive number: ";
        chomp($number = );
        $value = sqrt($number);
        print("The square root of $number is $value\n");
        }
elsif($choice == 4)
        {
        print "Enter a number: ";
        chomp($number = );
        $value = exp($number);
        print("e ** $number = $value\n");
        }
elsif($choice == 5)
        {
        print "Enter a number: ";
        chomp($number = );
        $value = log($number);
```

```
        $value = log($number);
        print("The natural log of $number is  $value\n");
        }
elsif($choice == 6)
        {
        print "Enter a number: ";
        chomp($number = );
        $value = abs($number);
        print("The absolute value of $number is $value\n");
        }
 elsif($choice == 7)
        {
        print "Enter a number: ";
        chomp($number = );
        $value = hex($number);
        print("The decimal value of $number is $value\n");
        }
elsif($choice == 8)
        {
        print "Enter a number: ";
        chomp($number = );
        $value = oct($number);
        print("The decimal value of $number is $value\n");
        }
elsif($choice == 9)
        {
        print "Enter a number: ";
        chomp($number = );
        $value = int($number);
        print("The integer value of $number is $value\n");
        }
else
        {
        print("Invalid choice\n");
        }
```

And finally, if you need to use Perl to generate random numbers, you should know about the rand() function. The rand() function takes a number as parameter, and generates a random number between 0 and that number. Here's an example:

```
#!/usr/bin/perl
print rand(9);
```

And this could return

```
7.06539493566379
```

If you omit the parameter, you'll get a random number between 0 and 1. And here's a script that asks you for a numerical range, and then returns a random number within that range:

```
#!/usr/bin/perl

# get the limits
print "Enter the lower limit of the range: ";
$lower = ;
chomp ($lower);

print "Enter the upper limit of the range: ";
$upper = ;
chomp ($upper);

# keep generating until number falls within range
while ($random < $lower)
        {
        $random = int(rand($upper));
        }

# then print
print $random;
```

And that's about all we have time for today. We'll be back with more in a couple of weeks - so keep coming back!

Perl 101 (part 7) - CGI Basics

By Vikram Vaswani and Harish Kamath
Melonfire
September 25, 2000

Moving On

If you've been paying attention these last few weeks, you should now know enough to write basic Perl programs. And with that task accomplished, Perl 101 now turns its attention to teaching you how to use Perl to generate HTML pages dynamically through server-side CGI scripts.

Since CGI scripts typically involve passing a number of name-value pairs of variables from one page to another, we're first going to introduce you to a new type of variable, and then use that knowledge to build some simple CGI scripts. Keep reading!

Perl 101 (part 7) - CGI Basics

By Vikram Vaswani and Harish Kamath
Melonfire
September 25, 2000

Meet Donald Duck

So far, you've used two different types of Perl variables - the scalar and the array. However, unbeknownst to you, Perl also comes with a third type of variable - the hash.

One of the significant features of an array is that array values can only be accessed via a numerical index. This implies that if you need to access an element of the array, you need to first know its exact location in the array. Since this can get complicated with large arrays, Perl offers you a simpler way to access array values, using easy-to-remember "keywords" or "keys".

Thus, a hashes is a species of Perl variable which allows you to define an array of key-value pairs. Take a look:

```
%myhero = ("fname" => "Donald", "lname" => "Duck");
```

The Perl statement above will create a hash named "myhero", which consists of two name-value pairs. The first key is "fname", and it points to the value "Donald", while the second is "lname" and it points to "Duck".

You can also write the statement above like this:

```
%myhero = ("fname", "Donald", "lname", "Duck");
```

Accessing the elements of a hash is equally simple - in the example above, the notation

```
$myhero{"fname"}
```

will return the first value of the hash ("Donald"), while

```
$myhero{"lname"}
```

will return the second value ("Duck").

The rules following hash names are the same as those for scalar and array variables - a hash name begins with a % symbol, followed by an alphabetic character, which may be followed by one or more numbers or letters. Hashes also have their own "space" in Perl.so the variables $duck, @duck and %duck are not treated as one and the same.

Perl 101 (part 7) - CGI Basics

By Vikram Vaswani and Harish Kamath
Melonfire
September 25, 2000

Heroes Of The Silver Screen

Here's a simple program that demonstrates how hashes can be used:

```
#!/usr/bin/perl

# define a hash
%director = ("1995" => "Mel Gibson", "1996" => "Anthony Minghella", "1997"
=> "James Cameron", "1998" => "Steven Spielberg", "1999" => "Sam Mendes");

# print
print "The Best Director Oscar in 1995 went to $director{1995}\n";
print "The Best Director Oscar in 1996 went to $director{1996}\n";
print "The Best Director Oscar in 1997 went to $director{1997}\n";
print "The Best Director Oscar in 1998 went to $director{1998}\n";
print "The Best Director Oscar in 1999 went to $director{1999}\n";
```

And here's the output:

```
The Best Director Oscar in 1995 went to Mel Gibson
The Best Director Oscar in 1996 went to Anthony Minghella
The Best Director Oscar in 1997 went to James Cameron
The Best Director Oscar in 1998 went to Steven Spielberg
The Best Director Oscar in 1999 went to Sam Mendes
```

In the example above, a hash has been used to store a bunch of name-value pairs, and a print() function has been used to display them.

You can also use the alternate hash notation if you prefer.

```
# define a hash
%director = ("1995", "Mel Gibson", "1996", "Anthony Minghella", "1997",
"James Cameron", "1998", "Steven Spielberg", "1999", "Sam Mendes");
```

Perl 101 (part 7) - CGI Basics

By Vikram Vaswani and Harish Kamath
Melonfire
September 25, 2000

Open Sesame

The example above demonstrates how hashes can speed up access to specific values in the array. However, if you'd like to access each and every element of the hash in a sequential fashion, things can get hairy.which is why Perl has a few functions designed to simplify that task.

The first of these is the keys() function, which returns a list of all the keys in the specified hash as an array. Here's an example:

```
#!/usr/bin/perl

# define a hash
%director = ("1995" => "Mel Gibson", "1996" => "Anthony Minghella", "1997"
=> "James Cameron", "1998" => "Steven Spielberg", "1999" => "Sam Mendes");

# get the names
@year = keys(%director);

# and use them in a loop
foreach $year(@year)
        {
        print "And the Oscar for Best Director($year) goes to
$director{$year}\n";
        }
```

And the output is:

```
And the Oscar for Best Director (1995) goes to Mel Gibson
And the Oscar for Best Director (1996) goes to Anthony Minghella
And the Oscar for Best Director (1997) goes to James Cameron
And the Oscar for Best Director (1998) goes to Steven Spielberg
And the Oscar for Best Director (1999) goes to Sam Mendes
```

In this case, the keys() function returns an array named @year, which looks like this:

```
@year = ("1995", "1996", "1997", "1998", 1999");
```

Once this array has been generated, the "foreach" loop is used to iterate through it and print each name-value pair. And there's a corresponding values() function, which returns.yup, you guessed it, the values from a hash.

```
#!/usr/bin/perl

# define a hash
%director = ("1995" => "Mel Gibson", "1996" => "Anthony Minghella", "1997"
=> "James Cameron", "1998" => "Steven Spielberg", "1999" => "Sam Mendes");

# get the names
@names = values(%director);

print "The Best Directors on the planet are: \n";

# and use them in a loop
foreach $name (@names)
        {
        print "$name \n";
        }
```

And the output is:

```
The Best Directors on the planet are:
Mel Gibson
Anthony Minghella
James Cameron
Steven Spielberg
Sam Mendes
```

Perl 101 (part 7) - CGI Basics

By Vikram Vaswani and Harish Kamath
Melonfire
September 25, 2000

JUMP TO:

> Moving On
> Meet Donald Duck
> Heroes Of The Silver Screen
> Open Sesame
> Each() Time The Lights Go Out.
> Perl And CGI
> A Cure For Low Self-Esteem
> GETting Your Form To Work

Each() Time The Lights Go Out.

Perl also has the each() function, designed to return a two-element array for each key-value pair in the hash. This comes in particularly useful when iterating through the hash, and is conceptually similar to the foreach() loop. Take a look:

```
#!/usr/bin/perl

# define movie hash
%film = (1995 => 'Braveheart', 1996 => 'The English Patient', 1997 =>
'Titanic', 1998 => 'Saving Private Ryan', 1999 => 'American Beauty');

# define director hash
%director = ('Braveheart' => 'Mel Gibson', 'The English Patient' =>
'Anthony Minghella', 'Titanic' => 'James Cameron', 'Saving Private Ryan' =>
'Steven Spielberg', 'American Beauty' => 'Sam Mendes');

# use loop to iterate through hash
while (($year, $filmname) = each %film)
        {
        print "The Oscar for Best Director($year) goes to
$director{$filmname} for $film{$year}\n";
        }
```

In the example above, we've created two hashes, one for the movies and years, and the other linking the movies with their directors. Next, we've used the each() function to assign the name-value pairs from the first hash to two variables, $year and $filmname, which are then used to obtain the corresponding director names from the second hash.

Here's the output:

```
The Oscar for Best Director(1995) goes to Mel Gibson for Braveheart
The Oscar for Best Director(1996) goes to Anthony Minghella for The English
Patient
The Oscar for Best Director(1997) goes to James Cameron for Titanic
The Oscar for Best Director(1998) goes to Steven Spielberg for Saving
Private Ryan
The Oscar for Best Director(1999) goes to Sam Mendes for American Beauty
```

Note that Perl allows you to use scalars within the hash notation as well - $director{$filmname} above is an example of this.

And finally, the delete() function allows you to delete a pair of elements from the hash. For example, if you have the hash

```
%film = (1995 => 'Braveheart', 1996 => 'The English Patient', 1997 =>
'Titanic', 1998 => 'Saving Private Ryan', 1999 => 'American Beauty');
```

you can delete the second entry(1996) like this:

```
delete $film{1996};
```

And your hash will then look like this:

```
%film = (1995 => 'Braveheart', 1997 => 'Titanic', 1998 => 'Saving Private
Ryan', 1999 => 'American Beauty');
```

Perl 101 (part 7) - CGI Basics

By Vikram Vaswani and Harish Kamath
Melonfire
September 25, 2000

Perl And CGI

You're probably wondering what all this has to do with anything, and why exactly we're torturing you with it at this stage in this series. Well, hashes come in very useful when writing CGI scripts that run off your Web server.and coincidentally, that's just what we're going to be talking about next.

Over the last couple of months, you've spent a great deal of effort understanding variables, loops, file input and output, and string manipulation. The goal of all this: to give you the confidence to use Perl to develop dynamic Web pages. And we're now at the point where you begin applying your hard-won knowledge to some real-life applications.

In non-geek terms, CGI, or the Common Gateway Interface, is a programming environment which allows you to communicate between Web pages and a Web server. So, when you submit a form on a Web site (for example), a CGI program on the server receives your information, saves it to a database or flat file, and dynamically generates an acknowledgment page. The form data is usually submitted is a series of name-value pairs (think hashes!) which is then interpreted and used by the CGI program.

Of course, CGI programs can do much more than this - you can run CGI programs to communicate with a database, read and write flat files, or run processes on the server.

Perl is by far the most common language used for CGI programs. If you're using Apache, you should have the ability to execute CGI scripts out of the box - although you need to keep the following points in mind:

* Your CGI script usually needs to be stored in a particular directory for it to work correctly. This directory is usually the /cgi-bin/ directory off your Web server, although you should check with your Webmaster or system administrator to make sure.

* CGI scripts need to be "mode-executable" under *NIX systems. Simply use the "chmod" command to give your scripts, and the directory they reside in, 0755 permission.

* Many Web servers require that your CGI script end in the file extension .cgi. Again, you'll need to check this with your system administrator.

* Since CGI scripts can run commands on the server, they pose an inherent security risk. You'd be well advised to check that your scripts do not open security holes on your system by running them past an experienced Webmaster before using them in a production environment. The examples we're going to be using are meant for demonstration purposes *only*.

* If you're going to process form data via a CGI script, make sure that your server supports both GET and POST methods of transferring data.

'nuff said. Let's actually write one of these babies.

Perl 101 (part 7) - CGI Basics

By Vikram Vaswani and Harish Kamath
Melonfire
September 25, 2000

A Cure For Low Self-Esteem

```
#!/usr/bin/perl

print "Content-Type: text/html\n\n";
print "<html><body><h1>God, I'm good!</h1></body></html>";
```

And when you view this page in your browser (assuming you've placed it in the appropriate place with the appropriate permissions, and that the file is called "good.cgi") by surfing to http://localhost/cgi-bin/good.cgi, you'll see this:

```
God, I'm good!
```

Let's dissect this a bit. The first line of the script is one you'll have to get used to seeing in all your CGI scripts, as it tells the browser how to render the data that follows. In this case, we're telling the browser to render it as HTML text.

Note the two line breaks following the Content-Type statement - forgetting these is one of the most common mistakes Perl newbies make, and it's the cause of a whole slew of error messages.

Our next line is a print() statement which simply outputs some HTML-formatted text - this is what the browser will see.

You can also use variables when generating your page - as the next example demonstrates:

```
#!/usr/bin/perl

print "Content-Type: text/html\n\n";
print "<html><body><h1>Tables</h1>";
print "<table border=2>";

for ($x=1; $x<=5; $x++)
{
print "<tr><td>Row $x</td></tr>";
}

print "</table></body></html>";
```

Perl 101 (part 7) - CGI Basics

By Vikram Vaswani and Harish Kamath
Melonfire
September 25, 2000

GETting Your Form To Work

The CGI environment comes with a set of pre-defined variables that can assist in the task of developing server-side scripts. Here's a list of the important ones:

$ENV{REMOTE_ADDR} The address of the client machine
$ENV{REMOTE_HOST} The host name of the client machine
$ENV{REQUEST_METHOD} The method used by a form to request data
$ENV{HTTP_USER_AGENT} The client browser
$ENV{QUERY_STRING} The query string passed if the GET method is used

Our next few examples will give you some idea of how these, and similar variables, can be used within your scripts.

```
#! /usr/bin/perl

$ip = $ENV{REMOTE_ADDR};
$browser = $ENV{HTTP_USER_AGENT};
print "Content-Type: text/html\n\n";
print "<html><body><font face=Arial>Your IP address is $ip and your browser
is $browser</font></body></html>";
```

And now, when you browse to this page, you'll see some information on your IP address and browser version.

Our next example demonstrates how a Perl/CGI script can be used to process data submitted via a form. Here's the form...

```
<html>
<head>
<basefont face=Arial>
</head>
<body>
<form action=http://localhost/readform.cgi method=GET>
Enter your first name: <input type=text length=20 name=name><br>
<input type=submit value=Submit>
</form>
</body>
</html>
```

...and here's the CGI script that receives and processes the data.

```
#!/usr/bin/perl

# readform.cgi
# reads form data and generates a page

# for GET data
if ($ENV{'REQUEST_METHOD'} eq "GET")
{
$yourname=$ENV{'QUERY_STRING'};
}
# for POST data
 else
{
$yourname = <STDIN>;
}

# Remove spaces if any
$yourname =~ s/\+/ /g;

# split form data and store in hash
%details = split (/=/, $yourname);

# generate page
print "Content-Type: text/html\n\n";
print "<html><body>";

while (($name, $value) = each %details)
{
print "Thank you for your submission, $value!\n";
}

print "</body></html>";
```

The script above will accept data from the HTML form and store it in a variable called $yourname. The manner in which data is submitted by the form(GET or POST) decides the manner in which it is assigned to the variable; the QUERY_STRING environment variable is used to make this decision.

If the text entered into the form contains spaces, the spaces are replaced with + characters when the form is submitted - this needs to be reversed via a regex. For example, if you enter the name "Luke Skywalker" into the form, the URL string will look like this:

http://localhost/cgi-bin/readform.cgi?name=Luke+Skywalker

Next, the name-value pairs are split apart on the basis of the = sign, and are assigned to their respective places in the hash %details. This hash is then used to print the name in the result page, which is also generated by the same script.

Thus, the CGI environment can be used to accept data from one Web page, process it or transfer it to another, and generate a new page containing dynamically-generated output. This is the basis of using CGI to create dynamic Web sites.

In the next issue of Perl 101, we'll delve deeper into CGI, with a look at some simple CGI programs that can be used to track hits on your Web site, or store visitor comments. Don't miss it!

Perl 101 (part 8): Putting It To The Test

By Vikram Vaswani and Harish Kamath
Melonfire
October 02, 2000

Reach Out And Touch Someone

Last time out, we discussed how Perl can be used to write simple CGI applications that reside on the server and do useful things like receiving form data. This time, we're going to take it a little further, by showing you how to make Perl do some really useful things.

Over the next few pages, we're going to show you how to track the number of visitors your page receives with a simple counter, store their comments on your Web development efforts in a basic guestbook, and write a simple mailer that can email the contents of a feedback form to the site webmaster. So keep reading - this is your chance to get to know your site visitors better!

Perl 101 (part 8): Putting It To The Test

By Vikram Vaswani and Harish Kamath
Melonfire
October 02, 2000

Adding Things Up

The first tool that we're going to discuss today is the counter. The basic purpose of a Web site counter is to record the number of people who have visited a particular page.

There are four simple steps involved in building a counter:

Step One: Create a text file on the server, which will store the number of visitors.

Step Two: When someone visits the site, this file is accessed, the number stored within it is obtained and incremented by one, and then displayed to the visitor.

Step Three: The new number is written back to the file.

Step Four: Go back to Step Two.

Here's the code:

```
#!/usr/bin/perl

# counter.cgi - count visitors to page

# open a file handle to read the contents of the file "counter.dat"
# this file contains the number of visitors before the
# current user.

# make sure that you have permission to write to this file

open (COUNT, "counter.dat");

# assign the value that is stored in the file handle in a temporary variable
$counter = <COUNT>;
close (COUNT);

# open the file handle to write the new number back to the file
open (COUNT, "> counter.dat");

# increment the counter variable by one
$counter += 1;

# write the new number
print COUNT "$counter";

close (COUNT);

# now display the HTML code
print "Content-Type: text/html\n\n";

# print the new counter value
print "<html>\n<body>\n<h1>Your are visitor number
$counter.</h1>\n</body>\n</html>";
```

And here's the output:

```
Content-Type: text/html

<html>
<body>
<h1>Your are visitor number 30.</h1>
</body>
</html>
```

You can easily extend the functionality of this counter to store the IP address, the browser type, the referrer URL and other visitor statistics by making use of the environment variables we showed you last time.but we're going to leave that to you.

Perl 101 (part 8): Putting It To The Test

By Vikram Vaswani and Harish Kamath
Melonfire
October 02, 2000

Visitors Welcome!
Next up, a guest book. We're going to create a simple guest book that asks the user to enter three parameters: name, email address and comment.

In order to make things simple for you, we've included the basic HTML code for the form as well.all you need to do is cut, copy and paste into your favourite HTML editor. Good service, huh?

```
<html>
<head>
<basefont face=Arial>
<title>Guest Book</title>
</head>
<body>
<form action="submit.cgi" method="post">
<center>
<h2>Your Space In My Space</H2>
<table width="600" bgcolor="#D6D6D6" cellpadding="10" cellspacing="5">
    <tr>
        <td width="300" align="right">Name</td>
        <td width="300" align="left"><input type="text" name="name"
size="25" maxlength="25"></td>
    </tr>
    <tr align="center">
        <td width="300" align="right">Email</td>
        <td width="300" align="left"><input type="text" name="email"
size="25" ></td>
    </tr>
    <tr>
        <td align="right" width="300">Comments</td>
        <td align="left" width="300"><textarea name="comments" cols="25"
rows="3" wrap="virtual"></textarea></td>
    </tr>
    <tr>
        <td colspan=2 align="center" width="600"><input type="submit"
value="Sign My Guestbook"></td>
    </tr>
</table>
</div>
</form>
</body>
</html>
```

As you can see, this form asks the user to enter three values: Name, Email address and Comments. The ACTION of the form is a CGI script that will accept the values entered into the form and store them in a text file.

Perl 101 (part 8): Putting It To The Test

By Vikram Vaswani and Harish Kamath
Melonfire
October 02, 2000

The Code...
Here's the other half of the puzzle - the CGI script that takes care of the actual storage of form data.

```
#! /usr/bin/perl

# submit.cgi - accepts guestbook data and writes to file

# define a variable that accepts a value from the form
$in;

# assign values to the variable depending on form METHOD

if ($ENV{'REQUEST_METHOD'} eq "GET") {

$in = $ENV{'QUERY_STRING'};
} else {

$in = <STDIN>;
}

# fix URL-encoded strings
$in =~ s/\+/ /g;


# all variables are passed to the script as name-value pairs separated by &
# split the input string on the basis of &
@detail = split (/&/, $in);

# display data entered by user again
print "Content-Type: text/html\n\n";
print "<html><body>";

print "<center>";
print "<table cellspacing=5 cellpadding=5 width=600 bgcolor=#D6D6D6>";

print "<tr><td align=center colspan=2 width=600><font face=Verdana
size=2>Thank you for entering the following details in the
guestbook.</font></td></tr>\n";

# each name-value pair is stored as an element of an array
# now take each element of the array and split to form a hash
# on the basis of the = symbol

# using the "foreach" loop, we split each element of the array into a
"temporary" hash
foreach $details(@detail)
{
    %details = split (/=/, $details);

# now extract the name-value pair from the temporary hash to display to the
user

# some forward thinking here:
# since values need to be stored in a text file, create a variable called
$entry
# and differentiate the different elements of each guestbook entry by the #
symbol
# this comes in useful later, wait and see!

    while (($name, $value) = each %details)
    {

        print "<tr><td align=right width=300><font face=Verdana size=2>Your
$name:</font></td><td align=left width=300><font face=Verdana size=2>
$value</font></td></tr>\n";
        $entry .= $value . "#";
    }
}

# end the display with a link that allows the user to view other guestbook
entries
print "<tr><td align=center colspan=2 width=600><font face=Verdana
size=1>Click <a href= view.cgi>here</a> to view other
entries.</font></td></tr></table>";

print "</body></html>";

# make things simple by ensuring that every entry in the file is on a
single line
# by terminating the entry with a newline
$entry = $entry."\n";

# this is where the file write actually happens
# remember to open the file in "append" mode to avoid losing previous data
```

```
# make sure that you have permission to write to this file

open (GBOOK, ">> guestbook.txt");
print GBOOK "$entry";
close (GBOOK);

# whew!
```

Now, how about an explanation?

Perl 101 (part 8): Putting It To The Test

By Vikram Vaswani and Harish Kamath
Melonfire
October 02, 2000

...And The Explanation

Let's start from the top. We've first defined a variable, $in, that accepts a value from the form. As explained in our previous example, all the form variables are passed to the script as name-value pairs. However, in this case, we are passing more than one name-value pair, separated from each other with an ampersand.

Therefore, it becomes necessary to split the input string against the & to get the individual name-value pairs. Each name-value pair is stored as an element of an array. When we begin printing the data entered (for verification), it becomes necessary to again split each element of the array into individual "names" and "values" against the = symbol. Splitting things up this way also makes it simple to write the different items to a text file.

Using the "foreach" loop, we split each element of the array, as described above, into a temporary hash variable. We can then extract the name-value pair from the temporary hash and display it to the user.

Since we have to also store the values in a text file, we've created a variable named $entry, formatted it in such a way that it contains all three values entered by the user (separated by a # symbol), and dumped it into a text file. Remember to open the file in "append" mode!

Perl 101 (part 8): Putting It To The Test

By Vikram Vaswani and Harish Kamath
Melonfire
October 02, 2000

Going Backwards

So that takes care of data entry. But how about the link that allows the user to read previous guestbook entries? Well, here's the code for the CGI script that reads the file and displays all previous entries to the user.

```
#! /usr/bin/perl

# view.cgi - display guestbook entries

# open the file and read contents into an array

open (GBOOK, "guestbook.txt");
@entries = <GBOOK>;
close (GBOOK);

print "Content-Type: text/html\n\n";
print "<html><body>";

print "<center>";
print "<table cellspacing=5 cellpadding=5 width=600 >";

print "<tr><td align=center width=600><font face=Verdana size=2>View other
comments.</font></td></tr>\n";

# at this point, each entry in the file is stored as a single element of
the array

foreach $entry(@entries)
{

# use chomp() to delete the newline character from the end of each line

    chomp $entry;

# split each array element against the # delimiter into a new temporary
array
# now, the first element of the temp array contains the name
# the second, the email address and the third, the comment

    @singles = split (/#/, $entry);

# display it
    print "<tr><td width=600>";

    print "<table cellspacing=5 cellpadding=0 width=600 bgcolor=#D6D6D6><tr
><td align=right width=300 ><font face=Verdana size=2>Name:</font></td><td
align=left width=300><font face=Verdana
size=2>$singles[0]</font></td></tr>\n";

    print "<tr><td align=right width=300><font face=Verdana size=2>Email
address:</font></td><td align=left width=300><font face=Verdana
size=2>$singles[1]</font></td></tr>\n";

    print "<tr><td align=right width=300 valign=top><font face=Verdana
size=2>Comments:</font></td><td align=left width=300><font face=Verdana
size=2>$singles[2]</font></td></tr></table>\n";

    print "</td></tr>";
}

print "</body></html>";
```

Now, when it's time to display the contents of the guestbook, we do things in reverse: first read the file into an array, split each element against the # delimiter we added earlier, and display it in a neatly formatted table. Simple, huh?

Perl 101 (part 8): Putting It To The Test

By Vikram Vaswani and Harish Kamath
Melonfire
October 02, 2000

Fortune Cookies

Perl also allows you to run external commands, and display the output of those commands on your Web page. Consider the following simple fortune cookie generator, which uses the "fortune" program to give you a random quote each time you reload the page.

```
#!/usr/bin/perl

# fortune.cgi - get a random quote

# get a quote - change your path appropriately
$quote = `/usr/games/fortune`;

# print it in a page
print "Content-Type: text/html\n\n";
print <<EOF;
<html>
<head>
<basefont face=Arial>
</head>
<body>
And your quote is:
<br>
$quote
</body>
</html>
EOF
```

In this case, we've executed a command by enclosing it in single quotes, and sent the output to the variable $quote. Next, we've used Perl to output an HTML page containing the quote - this page is the one you'll see when you visit the site through your browser. Each time you refresh it, you'll see a new quote.

Note the slightly different manner in which we've structured the print() statement here. The << marker indicates to Perl that what comes next is a multi-line block of text, and is to be printed as is right up to the marker (the marker in this case is the string "EOF"). In Perl-lingo, this is known as a "here document", and it comes in very handy when you need to output a chunk of HTML code.

Perl 101 (part 8): Putting It To The Test

By Vikram Vaswani and Harish Kamath
Melonfire
October 02, 2000

You Have Mail!

And on to our final example of the day - a simple form mailer. Let's assume that you have a feedback form which looks like this:

```
<html>
<head>
<style type=text/css>
td {font-family: Arial}
</style>
</head>

<body>

<h2>So Who Are You, Anyway?</h2>

<form action="mailform.cgi" method=post>
<table border="0" cellspacing="5" cellpadding="0">
<tr>
<td>Name</td>
<td><input type=text name=who size=30></td>
</tr>
<tr>
<td>Email address</td>
<td><input type=text name=email size=30></td>
</tr>
<tr>
<td valign="top">Address</td>
<td><textarea name="address" cols="30" rows="5"></textarea></td>
</tr>
<tr>
<td>Age</td>
<td><input type=text name=age size=2></td>
</tr>
<tr>
<td align=center colspan=2><input type=submit value=Send!>  <input
type=reset></td>
</tr>
</table>
</form>

</body>
</html>
```

Now, you need to have the contents of this form emailed to you every time someone submits it. Here's the Perl script that you'll need:

```
#! /usr/bin/perl

# mailform.cgi - email form contents to webmaster

# define a variable that accepts a value from the form
$in;

# assign values to the variable depending on form METHOD
if ($ENV{'REQUEST_METHOD'} eq "GET")
{
$in = $ENV{'QUERY_STRING'};
}
else {
$in = <STDIN>;
}

# fix URL-encoded strings
$in =~ s/\+/ /g;

# all variables are passed to the script as name-value pairs separated by &
# split the input string on the basis of &
@detail = split (/&/, $in);

# open mail pipe

open (MAIL,"|/usr/sbin/sendmail -t");
print MAIL "To: <webmaster\@yoursite.com>\n";
print MAIL "From: Feedback Form Mailer\n";
print MAIL "Subject: Feedback on your site\n\n";
print MAIL "Here is the result of your feedback form.\n\n";

foreach $details(@detail)
{
    %details = split (/=/, $details);
    while (($name, $value) = each %details)
```

```
    {
print MAIL "$name: $value\n";
    }
}

close MAIL;

print "Content-Type: text/html\n\n";
print "<html><body><center><font face=Arial size=+1>Thank you for your
feedback!</font></center></body></html>";
```

If you take a close look, you'll see that this script is very similar to the one we used when writing the guestbook. Here too, we've accepted form data as a single string, split it on the basis of the &, and then further split it against the = separator.

Next, we've opened a file handle - except that we haven't actually opened a file, but a UNIX "pipe", which allows you to "pipe" data to a UNIX command. In this case, the command is the sendmail program, which is used to deliver email. sendmail needs a few basic headers - the To:, From: and Subject: fields, which we've provided, followed by each name-value pair in the body of the message. Once all the pairs are exhausted, the handle is closed and the mail is sent out.

And here's the sample mail that you'll see:

```
To: webmaster@yoursite.com
From: Feedback.Form.Mailer
Subject: Feedback on your site

Here is the result of your feedback form.

who: johndoe
email: johndoe@cyberspace.com
address: the web
age: 28
```

And that's about it for this time. We hope you enjoyed this series of tutorials - write in and tell us what you'd like to see next. And till next time, stay healthy!