



Perl, Getting Started with

Short Course

Information Technology Services - Training

The University of Texas at Austin

<http://www.utexas.edu/its/training>

e-mail: training@its.utexas.edu

© 2002

Instructional Technology Services

The University of Texas at Austin

October 1, 2002

Table of Contents

Simple Perl programs	5
Data Structures	6
Scalar Variables	6
Lists	7
Manipulating lists	7
Scalar vs. list context	8
Associative Arrays	8
Input	10
Control Structures	11
Conditionals	11
Looping	12
while and until	12
for and foreach statements	13
next and last statements	14
Subroutines	14
Defining a subroutine	14
Calling subroutines	15
Local variables	16
Using other subroutines and modules	17
File Handles	17
Writing to files	18
Reading from files	18
File operators	20
File management functions	21
CGI Scripts	22
HTTP Headers	23
CGI Wrap	24
Debugging	24
Sources for Additional Information	25

Perl, Getting Started with

This course explains how to write simple Perl programs. Program techniques will include basic input and output and control structures and techniques for using Perl to write CGI scripts.

Objectives

After completing the course, you should be able to:

- Write a Perl program to display “Hello World” on the screen.
- Prompt user for input and get input from a file.
- Perform error checking with conditionals.
- Construct simple loops.
- Write a basic CGI script using the a Perl CGI library.

Workbook Conventions

The following conventions are used in this workbook:

Names of files and directories appear in bold letters with the first letter capitalized (e.g. **Myfile.doc**).

Keyboard shortcuts are represented as **<Ctrl C>**. This means hold down the Ctrl key and press the letter C.

All text to be input by student appears in bold type. (e.g. **The University of Texas at Austin.**)

A series of commands is typed in as follows:

Select File > New

Perl, Getting Started with

Perl is a useful programming language common on many UNIX systems. However, Perl is not unique to the UNIX world. It has grown in popularity because of the ease and flexibility it offers in performing data manipulation, reporting, and system administration functions. More recently, Perl has become the language of choice for writing CGI scripts on the World Wide Web. Several attractive features of perl include:

Freely available and portable – You can download the latest version of perl from <http://www.perl.com/pace/pub/perldocs/latest.html>. It is available for Windows, Macintosh, and UNIX. If you write code on a PC running Windows NT, it is easily ported to a Sun running Solaris.

Similarity to C, but easier – Perl is an interpreted language that shares many syntactical similarities to C. If you are a C programmer, you will be able to run with perl quickly. If you are not a C programmer don't panic, perl's learning curve is not very steep and you can write useful programs in a short period of time.

Text handling constructs – CGI scripts most frequently process text and generate HTML responses on the Web. Perl's text handling capabilities and support for regular expressions make it an excellent choice for CGI development.

Simple Perl programs

Perl is an interpreted language; the first line of a perl program specifies the path to the perl interpreter on the system. On most UNIX systems, perl is found in `/usr/bin/perl` or `/usr/local/bin/perl`. Type `which perl` on your system to determine the path to the perl interpreter.

Perl programs can have any valid filename, but most programs have a suffix of `.perl` or `.cgi` if they are cgi scripts. The `.pl` and `.pm` suffixes are generally reserved for Perl libraries and modules.

The classic Hello World program, which displays the words "Hello World" on the screen, appears below

Example 1

```
#!/usr/local/bin/perl
# Hello World program
print "Hello World\n";
```

The first line in Example 1 identifies this file to the shell as a perl program and specifies the path to the perl interpreter. The next line is a comment. Perl uses the `#` character to indicate comments. The `print` statement sends its arguments to standard output, normally the screen. Note the semi-colon (`;`) at the end of the `print` statement all perl statements end with a semi-colon.

Data Structures

Perl has three built in data structures, scalars, lists, and associative arrays.

Scalar Variables

Scalar variables store strings or numeric values. Scalar variables begin with a `$` followed by characters, digits, and underscores. Use the `=` operator to assign values to scalar variables. Several scalar variable assignments follow:

```
# assigns the value 100 to the scalar variable $n
$n = 100;

# assign string value "John" to scalar variable $name
$name = "John";

# adds $n and 50 and stores results in $bigger_num
$bigger_num = $n + 50;
```

Scalar variables do not have to be declared as integer or string because perl correctly interprets the scalar type depending on the context in which it is used.

In expressions and assignments, Perl distinguishes between single and double-quoted strings. Scalar variables and escape characters like newline `\n` and tab `\t` are interpreted when they appear within double quotes.

Example 2

```
#!/usr/local/bin/perl
# Scalar and quoting examples

# assigns fails to the $foo scalar variable
$foo = ' fails ';

# $s1 equals test followed by newline character
$s1 = "test\n";

# $s2 equals test followed by \ followed by n
$s2 = 'test\n';

# $s3 equals test fails (the value of $foo)
$s3 = "test $foo\n";

# $s4 equals the characters $s2, not the value of $s2
$s4 = '$s2';
print "$s3\n$s4";
```

In Example 2 when `\n` appears within single quotes it is not interpreted as a newline character. Also the value of `$foo` is interpreted inside the double quotes so `$s3` equals the word test followed by the value of `$foo`. However, when a scalar appears within single quotes, the scalar is not interpreted and the value of the resulting assignment is the literal string.

Lists

An array, or list, is an ordered collection of scalar values. Lists begin with the @ symbol. The statement:

```
@cities = ('Dallas','Houston','Austin','San Antonio');
```

creates a list called @cities and assigns it four elements. Each scalar value in a list is referenced individually using its position in the list. The first element of an array is in position 0, the second element is in position 1, and so on. For example, to access the second element of the @cities list use \$cities[1].

Although lists begin with the @ character the \$ symbol is used to reference specific elements of a list. This makes sense since an individual element of a list is a scalar value. To return the length of a list, use \$#listname. This actually returns the index of the last element of the list. Because the index of the first element is 0, the actual length of the array is \$#listname + 1.

It is not necessary in perl to declare a list prior to its creation. You can dynamically add items to a list by assigning additional elements. For example, \$cities[4]= "El Paso" adds a fifth element to the @cities list, or creates the @cities list if it does not already exist.

Manipulating lists

Two functions used frequently with lists are split and join. The join function takes a specified glue character and glues together all elements of an array to create a single scalar value. The split function does exactly the opposite; it splits a scalar on a specific character and creates a list. Perl's split function is very useful for parsing lines of delimited data into fields. Both the split and join functions take two arguments. The first argument is a string or regular expression that specifies the glue expression or split expression, and the second argument specifies the target of the operation. For example, @records = split(\n,\$lines) splits the scalar variable \$lines on the \n character and stores the result in @records.

Example 3 illustrates the use of join and split with lists.

Example 3

```
#!/usr/local/bin/perl
# Program example with split and join
@colors = ('blue','green','pink','yellow');

# Create a $shade scalar that contains blue\tgreen\tpink\tyellow
$shade = join("\t",@colors);
print "$shade";

# Split the $shade variable on \t
@newcolors = split("\t",$shade);
```

Scalar vs. list context

Many perl functions operate with both scalars and lists. Perl decides which interpretation to use depending on the context of the assignment or expression. Consider the `split` function. When used in a list context, the `split` function creates a list consisting of each field of an input record. When used in a scalar context however, the `split` function returns the number of elements in the line.

```
$in = Joe:Smith:34:CGI Wizard;
@input_field = split(":",$in);
$num_fields = split(":",$in);
# split $in but store the split pieces in four scalars
($first,$last,$id,$comment) = split(":",$in);
```

`@input_field` will be a four element array with Joe being the first element, but `$num_fields` will have a value of 4 because the `split` function was used in a scalar context.

Associative Arrays

An associative array, or hash, stores one or more scalar values each associated with a unique key. Unlike lists that use a numeric index, associative arrays can use any string. Associative arrays begin with the `%` symbol. However, when referencing individual elements of an associative array, use the `$` symbol and `{}`. For example, `$foo{'name'}` returns the value of the `%foo` array associated with the name key.

The following code creates an associative array called `%test` with keys browser and protocol. The value of the browser key is Lynx. Additional key value pairs can be added by specifying a key name and a value for that key.

```
# define %test associative array
%test = (
    'browser', 'Lynx',
    'protocol', 'http'
);
# print the browser element of the %test associative array
print "$test{browser}\n";
%test{"network"} = "AppleTalk";    # add new element to %test
array
```

Associative arrays are well-suited to storing Web form input. Each form input element has a name. Those names will be the keys in an associative array. The values of those keys are the data entered by the user. The keys operator is also useful when working with associative arrays. The perl statement `keys %array_name` returns a list of all the keys in the associative array `%array_name`. Note however that the keys are not returned in alphabetical order. To list the keys of an associative array in alphabetical use the expression `sort keys %array_name`.

The table below summarizes the different perl data types and how they are referenced.

Variable type	Representation	Referencing individual elements
Scalar	<code>\$var_name</code>	<code>\$var_name</code>
Array	<code>@array_name</code>	<code>\$array_name[1]</code>
Associative array	<code>%array_name</code>	<code>\$array_name{"key"}</code>

Example 4

```
#!/usr/local/bin/perl
# Example showing scalar variables, arrays and associative arrays

$index1 = 3;           # assigns value 3 to scalar variable
$index1
$index2 = 4;           # assigns value 4 to scalar variable
$index2
$prod = $index1 * $index2; # Assigns 12 to $prod

# assign John, Susan, Bill, Karen to the @names array
# $names[0] will equal John and $name[1] will equal Susan
@names = ('John', 'Susan', 'Bill', 'Karen');

# define "red" as a key and "blue" as the corresponding value in
# the associative array %foo, similarly "world" and 7 are keys
# with web and elections the corresponding values
%foo = (
    "red", "blue",
    "world", "web",
    7, "elections"
);

# prints Karen loves the web
print "$names[$index1] loves the $foo{world}\n";
%input = (
    'lastname', 'Cook'
    'quest', 'CGI Wizard'
    'age', '35'
);

# create a list containing all keys in %input array
@pair = keys %input;
```

Input

Use the `<STDIN>` operator to prompt a user for input during execution. For example, to ask a user to enter a number or a name, prompt them to type that information using a print statement and then use a scalar variable to hold the contents of the `<STDIN>`. The following example asks a user to enter a number between 1 and 10 and assigns the response from `<STDIN>` to `$num`. The last statement, `chop`, removes the last character (usually `\n`) from the input.

```
# prompt user to enter number
print "Please enter a number between 1 and 10\n";
$num = <STDIN>;      # store input in $num variable
chop($num);          # Removes newline character from end of input
```

You can also use the `<STDIN>` input to assign values to an array. In the example below, the `@names` array reads each line of standard input until the end of file is reached (or the user presses `^D` to signify end of input). Also notice that the assignment and the removal of the newline character with the `chop` operator are performed in the same line.

```
print "Enter names of people you want to track\n";
chop(@names = <STDIN>);
```

Perl can also accept input from command line arguments. The `@ARGV` list contains all of the command line parameters specified when perl is invoked. For example, if you type `perl example 10 5`, the `@ARGV` list contains (10, 5), the two parameters after the name of the file being executed. Example 5 uses the `@ARGV` list to multiply the two numbers specified on the command line after the filename.

Example 5

```
#!/usr/local/bin/perl
# Assign command line arguments to scalar variables
$x = $ARGV[0];
$y = $ARGV[1];

# Calculate and print answer to standard output
$answer = $x * $y;
print "The product of $x and $y is $answer\n";
```

Control Structures

All programming languages have control structures used to perform conditional execution and repeat statements. Both conditionals and loops use the concept of statement blocks. A statement block is a collection of one or more Perl statements included between { } characters.

Conditionals

To execute a block of Perl statements if a certain condition is true use Perl's `if` statement. In the following example Perl tests the condition (`$age > 29`). If that condition is true perl executes the line in the statement block and prints the words "Getting older". If the condition is false, perl executes the code immediately after the statement block.

```
if ($age > 29) {  
    print "Getting older\n";  
}
```

The `unless` statement is similar to `if` except it executes the statement block only if the specified condition is false. The following example prints "Still a youngster" unless `$age` is greater than 29.

```
unless ($age > 29) {  
    print "Still a youngster\n";  
}
```

When performing tests within conditionals use the comparison operators shown in the table below.

Comparison	Numeric	String
Equal	==	eq
Not equal	!=	ne
Less than	<	lt
Greater than	>	gt
Less than or equal to	<=	le
Greater than or equal to	>=	ge
Logical AND	&&	&&
Logical OR		

To execute one statement block if a condition is true but another block if the condition is false, use the `else` statement in conjunction with a `if` statement as shown in Example 6 below.

Example 6

```
#!/usr/local/bin/perl
# Conditional example
# Ask user to enter their name
print "Please enter your name\n";
chop($name = <STDIN>);
if ($name eq "Bill" || $name eq "Sue") {
    print "Hi $name\n";
}
else {
    print "Who are you?\n";
}
```

In Example 6, if `$name` equals "Bill" or "Sue", perl prints "Hi" otherwise it prints "Who are you".

The `elsif` statement, shown below is useful for handling multiple outcomes. Note that perl does not care where you type the beginning and ending `{ }` characters. However, you should adopt a style of placement and indenting to make your programs easy to read.

```
if ($age == 30) {
    statement 1;
    statement 2;
}
elsif (condition) {
    statement 1;
}
else {
    statement 1;
    statement 2;
}
```

Looping

Perl has several different looping constructs for repeating statement blocks.

while and until

Use the `while` or `until` statements to execute statements based on a certain condition. The statement block following the `while` statement executes as long as the condition is true. If the condition is not true to begin with, the loop does not execute at all and the program continues with the first statement after the `while` loop.

```
while (condition) {
    statement to repeat;
    statement to repeat;
}
```

For example, the following code prints the value of `$age` and increments it by 1 while `$age` is less than 30.

```
$age = 20;
while ($age < 30) {
  print $age;
  $age++;
}
```

Similarly the **until** statement repeats statements until a specified condition is true.

```
until (condition) {
  statement to repeat;
  statement to repeat;
}
```

for and foreach statements

To execute a loop a fixed number of times use Perl's **for** statement. The **for** statement uses initial and ending values and a increment expression to control how many times a loop executes.

```
for (starting value; ending value; increment) {
  statement;
  statement;
}
```

The following example successively assigns the variable `$number` values from 1 to 10 and then calculates and prints the square of those numbers.

Example 7

```
#!/usr/local/bin/perl
# For loop example

# Print common HTTP header used in CGI scripts
print "Content-type: text/html\n\n";
for ($number = 1; $number <= 10; $number++) {
  $square = $number * $number;
  print "The square of $number is $square<br>\n";
}
```

The **foreach** statement takes a list and assigns values from that list to a temporary scalar variable. This is useful when you want to iterate over all of the elements of a list. The first time through the loop, the scalar equals the first element of the list; the second time through the loop the scalar equals the second element.

The following example takes the keys of the `%in` associative array and assigns them one at a time to the `$control_variable`. It then prints the `$control_variable` and the corresponding value from the array.

```
foreach $control_variable (keys %in) {
    print "$control_variable = ${in{$control_variable}}\n";
}
```

Example 8 illustrates a `foreach` loop that displays all of the CGI environment variables and their values.

Example 8

```
#!/usr/local/bin/perl
# Program to Display CGI environment variables and their
# values
print "Content-type: text/html\n\n";
foreach $j (keys %ENV) {
    print "$j = $ENV{$j}<br>\n";
}
```

next and last statements

To handle a special condition in a loop, use the `next` or `last` statements to control the loop flow. The `last` statement terminates the loop and execution continues with the first statement after the loop. The `next` statement returns flow back to the top of the loop and continues execution from there.

Subroutines

Subroutines, or functions as they are known in perl, make your code more modular and easier to read. If you perform the same task in many programs, define a subroutine for that task so you do not have to type the same code over and over again.

Defining a subroutine

Define a subroutine using the `sub` statement. Following the `sub` statement, specify the subroutine name and begin a statement block that includes the code you need the subroutine to perform. Don't forget to end the subroutine with the closing `}` character.

```
sub subroutine_name {
    statements;
    ...
    statement;
}
```

Calling subroutines

To call a subroutine, type the subroutine name preceded by the `&` symbol. The subroutine returns the result of the last calculation performed. Or you can use the `return` statement to be more explicit about the subroutine's return value(s). To pass arguments to the subroutine, include those between parentheses immediately after the function call. The function receives those arguments in the `@_` array.

In Example 9 below, the `@calculate_interest` function takes three arguments and determines the monthly payment for a loan. Notice that the arguments are referenced using `$_[0]`. This is the first element of the `@_` array.

Example 9

```
#!/usr/local/bin/perl
#Program to calculate loan payment

$interest = .0825;
$term = 30;
$principal = 80000;

# Call subroutine and print the returned value
print &calculate_interest($interest,$term, $principal);

# Define calculate_interest subroutine
sub calculate_interest {

    # Convert annual interest rate to a monthly rate
    $monthly_interest = $_[0]/12;

    # Calculate term of loan in months
    $term_months = $_[1]*12;

    # Calculate numerator and denominator of fraction
    $numerator = $monthly_interest * (1 +
        $monthly_interest)**$term_months;
    $denominator = (1 + $monthly_interest)**$term_months - 1;

    # Calculate actual payment
    $payment = $_[2]*($numerator/$denominator);
}
```

Local variables

To define variables that are local to the subroutine, use the `my()` function. For example, the code below defines `$num` and `$name` as local variables in the `calculate` function.

```
$name = "bar";
$num = 0;
print "$name $num\n";
&calculate; # call function calculate
print "$name $num \n"; # last statement of main code

#now define the function called above
sub calculate {
    my($name) = "foo";
    my($num) = 10;
    print "$name $num\n";
    additional_statements;
} # end of sub calculate
```

From this example you can see that the first `print` statement prints "bar 0". The second `print` statement prints "foo 10" because the value of `$name` and `$num` changed within the subroutine. However, the final `print` statement again prints "bar 0" because the values assigned to `$name` and `$num` were only local to the subroutine `calculate` and when that subroutine exited, the values were restored to what they were on entering the subroutine.

Example 10 contains a subroutine to convert arguments to upper case using perl's `tr` function.

Example 10

```
sub Upper {
    my($string) = $_[0];

    # translate all lower case characters in $string to
    # upper case
    $string =~ tr/a-z/A-Z/;
    return $string;
}
```

In CGI scripts you often need to return a properly formatted HTML document. The `print_html_top` function is useful because it contains the output necessary for all HTML pages. When you pass this function a title, it prints the beginning of an HTML document.

Example 11

```
#!/usr/local/bin/perl
# Subroutine to return initial HTML formatting and
# background color

$title = 'Test Page';
$color = 'green';
print "Content-type: text/html\n\n";
&print_html_top($title,$color);

# print_HTML_top subroutine
sub print_html_top {
    my($t,$c) = @_
    print "<HTML>\n";
    print "<HEAD>\n";
    print "<TITLE>$t</TITLE>\n";
    print "</HEAD>\n";
    print "<BODY bgcolor=\"\$c\">\n";
}
```

Using other subroutines and modules

Perl has a large collection of subroutine libraries and modules that you can include in your own programs. Many of these modules are available at the Comprehensive Perl Archive Network (CPAN) located at <http://www.perl.com/CPAN/>. After you download and install the appropriate modules on your system, you can use the subroutines in your programs by including a use statement. For example use CGI; includes all of the functions in the CGI module.

File Handles

Perl uses file handles to communicate with the outside world. A file handle is perl's method of keeping track of files you use. You can open files for reading, writing, or appending or even open files handles to other processes. This is useful if you want to write the output of a program to sendmail on a UNIX system for example.

Use perl's open statement to open a file handle. The general syntax is `open(FILE_HANDLE_NAME,"filename")` where "FILE_HANDLE_NAME" is the name you assign the file handle and filename is the actual file you are reading or writing. By convention, file handle names are capitalized, but this is not a requirement.

```
# opens file appointments for reading under file handle
# name APPOINT
open (APPOINT,"<appointments");

# opens file test.out for writing under file handle name RESULTS
open (RESULTS,">test.out");

# opens file list.txt for appending under file handle
#name RESULTS
open (DB,">>list.txt");
```

Writing to files

Writing to a file handle is accomplished with perl's `print` statement. Specify the file handle after the `print` statement.

For example, `print OUTPUT "lastname\tfirstname\tID";` writes the specified string to the `OUTPUT` file handle. Similarly, Example 12 writes "Hello World" to a file rather than to standard output.

Example 12

```
#!/usr/local/bin/perl
# write hello world to text file

# open file handle
open(HELLO, ">output.txt");

# write string to HELLO
print HELLO "Hello File World\n";

# close HELLO file handle
close(HELLO);
```

Reading from files

Reading from files is a little more involved. Perl has a `read` function that reads a specified number of characters from a specified file handle, and stores the results in a specified scalar. The syntax of the `read` function is:

```
read(FILEHANDLE, $var, characters);
```

where `FILEHANDLE` is the file perl reads from, `$var` is the scalar where the data is stored and `characters` is the number of characters to read. The following example reads 9 characters from the `ID` handle and stores in the `$id` scalar.

```
open(ID, "<people");
read(ID, $id, 9);
print $id;
```

Example 13 uses the `stat` function to determine the length of a file and uses those results to read an entire file into a scalar variable. Perl's `stat` function returns an entire list of information about a file. Element 7 of that list is the file size. Example 13 is equivalent to the UNIX `cat` command.

Example 13

```
#!/usr/local/bin/perl
# Get command line parameter, the name of the file to
# read and display
$file = $ARGV[0];

#Get the size of the file
$file_length = (stat("$file"))[7];

# Open file for reading and read bytes into $content
# variable
open(INPUT,"<$file");
read(INPUT,$content,$file_length);

close(INPUT);
print $content;
```

The `read` function is most frequently used to read binary or fixed length data. To read delimited records from a file one line at a time use perl's `< >` operator. Imagine you have a list of states in a file called `states`. A sample listing appears below.

```
Alabama<td>Mobile<td>Central
Alaska<td>Juneau<td>Pacific
```

Example 14 illustrates how to read one line at a time from this file and display each line in a Web browser.

Example 14

```
#!/usr/local/perl
# Example to read from file and write output to Web
# browser

# print http header
print "Content-type: text/html\n\n";

# begin printing of HTML table
print "<BODY><table border=1>\n";

# open a file handle to the states files for reading
open(STATES,"<states.txt");

# the while loop reads a line at a time and stores each
# line in the scalar $states
while ($state = <STATES>) {
print "<tr><td>\n"; # print beginning of table row
    print "$state\n";
    print "</tr>\n";
}
print "</table></body>\n";
```

The `while ($states = <STATES>)` translates roughly to “while there are lines in the “STATES” file, read them and store the result in the scalar variable `$state`”. After the first time through the loop `$state` equals Alabama<td>Mobile<td>Central. If the file contains 50 states the loop will execute 50 times.

If you need to further parse each line after it has been read, use the `split` function discussed earlier. For example, `@fields = split("<td>", $state)` creates a list called `@fields`. The first element of the list is the state name, the second element is the capital, and the third element is the time zone.

File operators

When working with file handles and these file operations, it is often a good idea to make sure the operation performed successfully. You might try to open a file handle for a non-existent file.

The `die` statement forces the termination of the program and prints a specified string to Standard Error. A more common form of the `die` function is used in conjunction with the `open` function. If you try to open a file handle and fail, perhaps because the file does not exist or is not readable, then the program will exit. The following code attempts to open a file called `appointments`. If the `open` function fails, the program will end and the specified error message is written to Standard Error. In the case of a CGI script, Standard Error is the server’s error log.

```
open (APPOINT, "<appointments") || die "Sorry I could not read from
appointments: $!\n";
```

If the opening of `appointments` fails, Perl writes "Sorry I could not read from `appointments`" to Standard Error and terminates the program. The `$!` variable will print the error message from the operating system.

You can avoid some problems with opening or writing to files by testing the existence of files and checking their permissions. Perl has a series of file operators that test whether a file exists and if it is readable or writable. The following table lists the most common file operators and several examples.

Operator	Comments	Example
<code>-e filename</code>	Tests for existence of filename; returns true if file exists and false otherwise	<code>if(-e '/etc/passwd')</code>
<code>-r filename</code>	Returns true if file is readable	<code>if(-r '/usr/local/ foo')</code>
<code>-w filename</code>	Returns true if file is writable	<code>if (-w '/etc/passwd')</code>
<code>-d filename</code>	Returns true if filename is a directory	<code>if (-d 'foo')</code>

Example 15 is similar to Example 13 except it checks if the file exists before trying to read from it. If the file does not exist, it displays an error message.

Example 15

```
#!/usr/local/bin/perl
# Get command line parameter, the name of the file to
# read and display
$file = $ARGV[0];

# Check to see if $file exists, if no print error
# message
if (! -e $file) {
    print "File does not exist in this directory or is
misspelled\n";
    exit;
} else {
    #Get the size of the file
    $file_length = (stat($file))[7];

    # Open file for reading and read bytes into
    # $content variable
    open(INPUT,"<$file");
    read(INPUT,$content,$file_length);

    close(INPUT);
    print $content;
}
```

File management functions

Perl can read from and write to files using the file handles described above. It also enables you to perform common file operations like renaming and deleting files, creating directories and navigating the directory tree. The following table summarizes how to perform each of these operations in perl.

Task	Perl function	Example and Comments
Creating a directory	mkdir(directoryname,permissions)	mkdir('daily_data',0755)
Removing a directory	rmdir(directoryname)	rmdir('daily_data')
Changing directories	chdir(directoryname)	chdir('output') chdir('/tmp')
Renaming a file	rename(old-file-name, new-file-name)	rename('foo', 'newfoo')
Deleting a file	unlink(filename)	unlink('old_output')

Example 16 uses the `rename` function to rename all files that begin with uppercase letters to all lower case.

Example 16

```
#!/usr/local/bin/perl
#program to rename files with upper case to all lower case

#Create a list of all files that begin with two upper
#case letters in the current directory
@files = <[A-Z][A-Z]*>;

#iterate through the list of files, translate upper case
#characters to lower case and rename them
foreach $upperfile (@files) {
    $newfile = $upperfile;
    $newfile =~ tr/A-Z/a-z/;
    if (-e $newfile) {
        print "$newfile already exists. Cannot complete
        rename\n";
        exit;
    } else {
        rename($upperfile,$newfile);
        print "I just renamed $upperfile to $newfile\n";
    }
}
```

CGI Scripts

Perl is probably the most popular programming language for writing CGI scripts on the World Wide Web. CGI scripts must be able to read information from the server and send information back to the server in an appropriate form. Fortunately, there is a module of perl functions and subroutines to make this process easier.

The CGI Perl module, `CGI.pm`, is normally installed with Perl 5.0. If `CGI.pm` is not available on your system, download it from the Comprehensive Perl Archive Network, <http://www.perl.com/CPAN/>

The following two lines use the `ReadParse` library of the CGI module to read form data and store it in an associative array.

```
use CGI;
CGI::ReadParse(*data);
```

Example 20 uses these two lines to create an associative array called `%in`. It then prints the values of the form data back to the Web browser.

Example 20

```
#!/usr/local/bin/perl
# CGI script to echo form input and variables

# Include CGI perl library
use CGI;

#Get input from form and store it in %in
CGI::ReadParse(*in);

#Print header

# print HTML code
&print_html_top("Echo Form Variables and Input");
foreach $k (keys %in) {
    print "$k = $in{$k}\n";
}
print "</body>\n";
print "</html>\n";

sub print_html_top {
    print "<HTML>\n";
    print "<HEAD>\n";
    print "<TITLE>$title</TITLE>\n";
    print "</HEAD>\n";
    print "<BODY>\n";
}
```

HTTP Headers

CGI scripts must be able to read data from Web browsers, and they must be able to output properly formatted HTML documents. Perl's `print` statement sends information to standard output. When a perl program is executed through the Web as a CGI script, standard output is the Web server. Because of this CGI scripts must send http header information to the Web server before it returns HTML. Two commonly used headers are `Content-type: text/html\n\n` and `Location: URL\n\n`.

The first header, `Content-type`, identifies the output as HTML. The second header, `Location`, redirects the browser to the specified URL.

Example 21 illustrates the use of both these headers in a CGI script.

Example 21

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

# Get the type of browser being used from the
# HTTP_USER_AGENT
# Environment variable
$browser = $ENV{HTTP_USER_AGENT};

# Check to see if $browser contains MSIE
if ($browser =~ /MSIE/) {
    # if yes, send http header and then some formatted
    # HTML
    print "Content-type: text/html\n\n";
    print "<h3>You are using Microsoft Internet
    Explorer</h3>\n";
} else {
    # If no, send location header redirecting browser
    # to specified URL
    print "Location: http://home.netscape.com\n\n";
}
```

CGI Wrap

Normally CGI scripts are stored in a special `cgi-bin` directory under the `httpd` data directory and execute under a privileged user. In order to run your own CGI scripts on the UT Austin central web server you must use a program called CGIwrap. Instructions for using CGIwrap are available at <http://www.utexas.edu/learn/cgi/>

Debugging

Several command line switches provide helpful debugging tools in Perl.

Use the `perl -cw` switch to have Perl check your program for syntax errors without actually executing it.

Perl also has an interactive debugger. Type `perl -d filename` to activate the debugger. The prompt will change to `DB<n>`. Type `h` to get help with the different debugger commands.

Finally, the example below is useful for debugging CGI scripts. When CGI scripts encounter errors, you often get a malformed header message back from the server. This does not provide much useful information. Try executing your CGI script from the command line before testing them on the World Wide Web.

```
# if %in is not defined, assign values to various keys
# so the remainder of the program can execute
if ( ! defined(%in)) {
    $in{'NAME'} = 'defaultvalue or debugging value'
    $in{'OTHERVAR'} = 'defaultvalue or debugging value'
}
```

Sources for Additional Information

There are a number of good additional sources of information on Perl programming. Two books that are commonly used are Learning Perl (the Llama book), and Programming Perl (the Camel book) both by O'Reilly and Associates. There are numerous newsgroups that carry information on Perl. The most popular are comp.lang.perl.misc, comp.lang.perl.announce, and comp.lang.perl.modules. Finally, there are a number of web sites containing perl resources. The best site is the Comprehensive Perl Archive Network at <http://www.perl.com/CPAN/>.