UNIT 8

8. perl - The Master Manipulator   **7 Hours**

Text Book

8. **"UNIX – Concepts and Applications",** Sumitabha Das, 4th Edition, Tata McGraw Hill, 2006.

   (Chapters 1.2, 2, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19).

Reference Books

UNIX and Shell Programming, Behrouz A. Forouzan and Richard F. Gilberg, Thomson, 2005.

Unix & Shell Programming, M.G. Venkateshmurthy, Pearson Education, 2005.

.
Perl – The Mater Manipulator

Introduciton
The following sections tell you what Perl is, the variables and operators in perl, the string handling functions. The chapter also discusses file handling in perl as also the lists, arrays and associative arrays (hashes) that have made perl a popular scripting language. One or two lines of code in perl accomplish many lines of code in a high level language. We finally discuss writing subroutines in perl.

Objectives
- perl preliminaries
- The chop function
- Variables and Operators
- String handling functions
- Specifying filenames in a command line
- $_(Default Variable)
- $. (Current Line Number) and .. (The Range Operator)
- Lists and Arrays
- ARGV[]: Command Line Arguments
- foreach: Looping Through a List
- split: Splitting into a List or Array
- join: Joining a List
- dec2bin.pl: Converting a Decimal Number to Binary
- grep: Searching an Array for a Pattern
- Associative Arrays
- Regular Expressions and Substitution
- File Handling
- Subroutines
- Conclusion

1. Perl preliminaries
Perl: Perl stands for Practical Extraction and Reporting Language. The language was developed by Larry Wall. Perl is a popular programming language because of its powerful pattern matching capabilities, rich library of functions for arrays, lists and file handling. Perl is also a popular choice for developing CGI (Common Gateway Interface) scripts on the www (World Wide Web).

Perl is a simple yet useful programming language that provides the convenience of shell scripts and the power and flexibility of high-level programming languages. Perl programs are interpreted and executed directly, just as shell scripts are; however, they also contain control structures and operators similar to those found in the C programming language. This gives you the ability to write useful programs in a very
short time.

Perl is a freeware and can be obtained from http://www.perl.com or http://www.activestate.com (Perl interpreter for Windows).

A perl program runs in a special interpretive model; the entire script is compiled internally in memory before being executed. Script errors, if any, are generated before execution. Unlike awk, printing isn't perl's default action. Like C, all perl statements end with a semicolon. Perl statements can either be executed on command line with the –e option or placed in .pl files. In Perl, anytime a # character is recognized, the rest of the line is treated as a comment.

The following is a sample perl script.
```
#!/usr/bin/perl
# Script: sample.pl – Shows the use of variables
#
print("Enter your name: ");
$name=<STDIN>;
Print("Enter a temperature in Centigrade: ");
$centigrade=<STDIN>;
$fahr=$centigrade*9/5 + 32;
print "The temperature in Fahrenheit is $fahr\n";
print "Thank you $name for using this program."
```

There are two ways of running a perl script. One is to assign execute (x) permission on the script file and run it by specifying script filename (chmod +x filename). Other is to use perl interpreter at the command line followed by the script name. In the second case, we don't have to use the interpreter line viz., #!/usr/bin/perl.

2. The chop function
The chop function is used to remove the last character of a line or string. In the above program, the variable $name will contain the input entered as well as the newline character that was entered by the user. In order to remove the \n from the input variable, we use chop($name).
Example: chop($var); will remove the last character contained in the string specified by the variable *var*.
Note that you should use chop function whenever you read a line from the keyboard or a file unless you deliberately want to retain the newline character.

3. Variables and Operators
Perl variables have no type and need no initialization. However we need to precede the variable name with a $ for both variable initialization as well as evaluation.
Example: $var=10;
                print $var;
Some important points related to variables in perl are:
   1. When a string is used for numeric computation or comparison, perl converts it into a number.

2. If a variable is undefined, it is assumed to be a null string and a null string is numerically zero. Incrementing an uninitialized variable returns 1.
3. If the first character of a string is not numeric, the entire string becomes numerically equivalent to zero.
4. When Perl sees a string in the middle of an expression, it converts the string to an integer. To do this, it starts at the left of the string and continues until it sees a letter that is not a digit. Example: "12O34" is converted to the integer 12, not 12034.

Comparison Operators
Perl supports operators similar to C for performing numeric comparison. It also provides operators for performing string comparison, unlike C where we have to use either strcmp() or strcmpi() for string comparison. The are listed next.

Numeric comparison	String comparison
==	eq
!=	ne
>	gt
<	lt
>=	ge
<=	le

Concatenating and Repeating Strings
Perl provides three operators that operate on strings:
- The . operator, which joins two strings together;
- The x operator, which repeats a string; and
- The .= operator, which joins and then assigns.

The . operator joins the second operand to the first operand:
Example:
$a = "Info" . "sys";   # $a is now "Infosys"
$x="microsoft"; $y=".com"; $x=$x . $y;   # $x is now "microsoft.com"
This join operation is also known as string concatenation.

The x operator (the letter x) makes *n* copies of a string, where *n* is the value of the right operand:
Example:
$a = "R" x 5;   # $a is now "RRRRR"
The .= operator combines the operations of string concatenation and assignment:
Example:
$a = "VTU";
$a .= " Belgaum";   # $a is now "VTU Belgaum"

4. String Handling Functions
Perl has all the string handling functions that you can think of. We list some of the frequently used functions are:

length determines the length of its argument.
index(s1, s2) determines the position of a string **s2** within string **s1**.
substr(str,m,n) extracts a substring from a string **str**, **m** represents the starting point of extraction and **n** indicates the number of characters to be extracted.
uc(str) converts all the letters of str into uppercase.
ucfirst(str) converts first letter of all leading words into uppercase.
reverse(str) reverses the characters contained in string str.

5. Specifying Filenames in Command Line
Unlike awk, perl provides specific functions to open a file and perform I/O operations on it. We will look at them in a subsequent section. However, perl also supports special symbols that perform the same functionality. The diamond operator, <> is used for reading lines from a file. When you specify STDIN within the <>, a line is read from the standard input.
Example:
1. perl –e 'print while (<>)' sample.txt
2. perl –e 'print <>' sample.txt
In the first case, the file opening is implied and <> is used in scalar context (reading one line).
In the second case, the loop is also implied but <> is interpreted in list context (reading all lines).

The following script will print all Gupta's and Agarwal/Aggarwal's contained in a file (specified using an ERE) that is specified as a command line parameter along with the script name.

```
#!/usr/bin/perl
printf(%30s", "LIST OF EMPLOYEES\n");
while(<>) {
   print if /\bGupta|Ag+[ar][ar]wal/ ;
}
```

6. $_ : The Default Variable
perl assigns the line read from input to a special variable, $_, often called the default variable. chop, <> and pattern matching operate on $_ be default. It represents the last line read or the last pattern matched.

By default, any function that accepts a scalar variable can have its argument omitted. In this case, Perl uses $_, which is the default scalar variable. chop, <> and pattern matching operate on $_ by default, the reason why we did not specify it explicitly in the print statement in the previous script. The $_ is an important variable, which makes the perl script compact.

For example, instead of writing
        $var = <STDIN>;
        chop($var);
you can write,

chop(<STDIN>);

In this case, a line is read from standard input and assigned to default variable $_, of which the last character (in this case a \n) will be removed by the chop() function.

Note that you can reassign the value of $_, so that you can use the functions of perl without specifying either $_ or any variable name as argument.

7. $. (Current Line number) And .. (The range operator)

$. is the current line number. It is used to represent a line address and to select lines from anywhere.

Example:

perl –ne 'print if ($. < 4)' in.dat   # is similar ro head –n 3 in.dat
perl –ne 'print if ($. > 7 && $. < 11)' in.dat   # is similar to sed –n '8,10p'

.. is the range operator.

Example:

perl –ne 'print if (1..3)' in.dat   # Prints lines 1 to 3 from in.dat
perl –ne 'print if (8..10)' in.dat   # Prints lines 8 to 10 from in.dat

You can also use compound conditions for selecting multiple segments from a file.

Example: if ((1..2) || (13..15)) { print ;}   # Prints lines 1 to 2 and 13 to 15

8. Lists and Arrays

Perl allows us to manipulate groups of values, known as lists or arrays. These lists can be assigned to special variables known as array variables, which can be processed in a variety of ways.

A list is a collection of scalar values enclosed in parentheses. The following is a simple example of a list:

            (1, 5.3, "hello", 2)

This list contains four elements, each of which is a scalar value: the numbers 1 and 5.3, the string "hello", and the number 2.

To indicate a list with no elements, just specify the parentheses: ()

You can use different ways to form a list. Some of them are listed next.

- Lists can also contain scalar variables:
  (17, $var, "a string")
- A list element can also be an expression:
  (17, $var1 + $var2, 26 << 2)
- Scalar variables can also be replaced in strings:
  (17, "the answer is $var1")
- The following is a list created using the list range operator:
  (1..10)➔ same as (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
- The list range operator can be used to define part of a list:
  (2, 5..7, 11)
  The above list consists of five elements: the numbers 2, 5, 6, 7 and 11

Arrays

Perl allows you to store lists in special variables designed for that purpose. These variables are called array variables. Note that arrays in perl need not contain similar type of data. Also arrays in perl can dynamically grow or shrink at run time.

@array = (1, 2, 3);   # Here, the list (1, 2, 3) is assigned to the array variable @array.

Perl uses @ and $ to distinguish array variables from scalar variables, the same name can be used in an array variable and in a scalar variable:

$var = 1;

@var = (11, 27.1, "a string");

Here, the name var is used in both the scalar variable $var and the array variable @var. These are two completely separate variables. You retrieve value of the scalar variable by specifying $var, and of that of array at index 1 as $var[1] respectively.

Following are some of the examples of arrays with their description.

x = 27;   # list containing one element

@y = @x;   # assign one array variable to another

@x = (2, 3, 4);

@y = (1, @x, 5);   # the list (2, 3, 4) is substituted for @x, and the resulting list
                             # (1, 2, 3, 4,5) is assigned to @y.

$len = @y;   # When used as an *rvalue* of an assignment, @y evaluates to the
                           # length of the array.

$last_index = $#y;   # $# prefix to an array signifies the last index of the array.

9. ARGV[]: Command Line Arguments

The special array variable @ARGV is automatically defined to contain the strings entered on the command line when a Perl program is invoked. For example, if the program (test.pl):

#!/usr/bin/perl

print("The first argument is $ARGV[0]\n");

Then, entering the command

$ test.pl 1 2 3

produces the following output:

The first argument is 1

Note that $ARGV[0], the first element of the @ARGV array variable, does not contain the name of the program. This is a difference between Perl and C.

Modifying Array Contents

For deleting elements at the beginning or end of an array, perl uses the shift and pop functions. In that sense, array can be thought of both as a stack or a queue.

Example:

@list = (3..5, 9);

shift(@list);   # The 3 goes away, becomes 4 5 9

pop(@list);   # Removes last element, becomes 4 5

The unshift and push functions add elements to an array.
unshift(@list, 1..3);   # Adds 1, 2 and 3 –- 1 2 3 4 5
push(@list,9); # Pushes 9 at end –- 1 2 3 4 5 9

The splice function can do everything that shift, pop, unshift and push can do. It uses upto four arguments to add or remove elements at any location in the array. The second argument is the offset from where the insertion or removal should begin. The third argument represents the number of elements to be removed. If it is 0, elements have to be added. The new replaced list is specified by the fourth argument (if present).
splice(@list, 5, 0, 6..8);   # Adds at 6th location, list becomes 1 2 3 4 5 6 7 8 9
splice(@list, 0, 2); # Removes from beginning, list becomes 3 4 5 6 7 8 9

10. foreach: Looping Through a List
foreach construct is used to loop through a list. Its general form is,
```
        foreach $var in (@arr) {
                statements
        }
```

Example: To iterate through the command line arguments (that are specified as numbers) and find their square roots,
```
foreach $number (@ARGV) {
        print("The square root of $number is " .
        sqrt($number) .  "\n");
}
```
You can even use the following code segment for performing the same task. Here note the use of $_ as a default variable.
```
foreach (@ARGV) {
  print("The square root of $_ is " . sqrt() . "\");
}
```

Another Example
```
#!/usr/bin/perl
@list = ("This", "is", "a", "list", "of", "words");
print("Here are the words in the list: \n");
foreach $temp (@list) {
print("$temp ");
}
print("\n");
```
Here, the loop defined by the foreach statement executes once for each element in the list @list. The resulting output is
Here are the words in the list:
This is a list of words

The current element of the list being used as the counter is stored in a special scalar variable, which in this case is $temp. This variable is special because it is only defined for the statements inside the foreach loop.

perl has a for loop as well whose syntax similar to C.
Example:
          for($i=0 ; $i < 3 ; $i++) { . . .

11. split: Splitting into a List or Array
There are two important array handling functions in perl that are very useful in CGI programming, viz., split and join.

split breaks up a line or expression into fields. These fields are assigned either to variables or an array.
Syntax:
($var1, $var2, $var3 ….…) = split(/sep/, str);
@arr = split(/sep/, str);
It splits the string **str** on the pattern **sep**. Here **sep** can be a regular expression or a literal string. **str** is optional, and if absent, **$_** is used as default. The fields resulting from the split are assigned to a set of variables , or to an array.

12. join: Joining a List
It acts in an opposite manner to split. It combines all array elements in to a single string. It uses the delimiter as the first argument. The remaining arguments could be either an array name or a list of variables or strings to be joined.
$x = join(" ", "this", "is", "a", "sentence");   # $x becomes "this is a sentence".
@x = ("words","separated","by");
$y = join("::",@x,"colons");   #$y becomes "words::separated::by::colons".

To undo the effects of join(), call the function split():
          $y = "words::separated::by::colons";
          @x = split(/::/, $y);

13. dec2bin: Converting a Decimal Number to Binary
Here we summarize our understanding of array handling functions with an illustrative script that converts a input decimal number into its binary equivalent. The script logic is to repeatedly divide the number by two and collecting the remainders and finally printing the reverse of all the collected remainders. The script is as follows:

```perl
#!/usr/bin/perl
foreach $num (@ARGV) {
  $temp = $num;
  until ($num == 0) {
    $bit = $num % 2;
    unshift(@bit_arr, $bit);
    $num = int($num/2);
}
```

```
 $binary_num = join(“”,@bit_arr);
 print (“Binary form of $temp is $binary_num\n”);
 splice(@bit_arr, 0, $#bit_arr+1);
}
```
The output of the above script (assuming script name is dec2bin.pl) is,

```
$ dec2bin.pl 10
Binary form of 10 is 1010

$ dec2bin.pl 8 12 15 10
Binary form of 8 is 1000
Binary form of 12 is 1100
Binary form of 15 is 1111
Binary form of 10 is 1010

$
```

14. grep: Searching an array for pattern
grep function of perl searches an array for a pattern and returns an array which stores the array elements found in the other array.
Example:
$found_arr = grep(/^$code/, @dept_arr); # will search for the specified $code at the beginning of the element in the array @dept_arr.

15. Associative Arrays
In ordinary arrays, you access an array element by specifying an integer as the index:
@fruits = (9, 23, 11);
$count = $fruits[0];   # $count is now 9
In associative arrays, you do not have to use numbers such as 0, 1, and 2 to access array elements. When you define an associative array, you specify the scalar values you want to use to access the elements of the array. For example, here is a definition of a simple associative array:
%fruits=("apple", 9, "banana", 23, "cherry", 11);
It alternates the array subscripts and values in a comma separated strings. i.e., it is basically a key-value pair, where you can refer to a value by specifying the key.
$fruits{“apple”} will retrieve 9. $fruits{“banana”} will retrieve 23 and so on.
Note the use of {} instead of [] here.

There are two associative array functions, keys and values.
keys: Holds the list of subscripts in a separate array.
values: Holds the value of each element in another array.

Normally, keys returns the key strings in a random sequence. To order the list alphabetically, use sort function with keys.
1. foreach $key (sort(keys %region)) {   # sorts on keys in the associative array, region
2. @key_list = reverse sort keys %region; # reverse sorts on keys in assoc. array, region

16. Regular Expressions and Substitution

perl supports different forms of regular expressions we have studied so far. It makes use of the functions s and tr to perform substitution and translation respectively.

The s function: Substitution

You can use the =~ operator to substitute one string for another:

```
$val =~ s/abc/def/;      # replace abc with def
$val =~ s/a+/xyz/;       # replace a, aa, aaa, etc., with xyz
$val =~ s/a/b/g;         # replace all a's with b's;It also uses the g flag for global
                         # substitution
```

Here, the s prefix indicates that the pattern between the first / and the second is to be replaced by the string between the second / and the third.

The tr function: Translation

You can also translate characters using the tr prefix:

```
$val =~ tr/a-z/A-Z/;     # translate lower case to upper
```

Here, any character matched by the first pattern is replaced by the corresponding character in the second pattern.

Using Special Characters in Patterns

The following examples demonstrate the use of special characters in a pattern.

1. The * character matches zero or more of the character it follows:
   /jk*l/   # This matches jl, jkl, jkkl, jkkkl, and so on.
2. The + character matches one or more of the preceding character:
   /jk+l/   # This matches jkl, jkkl, jkkkl, and so on.
3. The ? character matches zero or one copies of the preceding character:
   /jk?l/   # This matches jl or jkl.
4. If a set of characters is enclosed in square brackets, any character in the set is an acceptable match:
   /j[kK]l/   # matches jkl or jKl
5. Consecutive alphanumeric characters in the set can be represented by a dash (-):
   /j[k1-3K]l/   # matches jkl, j1l, j2l, j3l or jKl
6. You can specify that a match must be at the start or end of a line by using ^ or $:
   /^jkl/   # matches jkl at start of line
   /jkl$/   # matches jkl at end of line
7. Some sets are so common that special characters exist to represent them:
   \d matches any digit, and is equivalent to [0-9].
   \D doesn't match a digit, same as [^0-9].
   \w matches any character that can appear in a variable name; it is equivalent to

[A-Za-z0-9_].
\W doesn't match a word character, same as [^a-zA-Z0-9_]
\s matches any whitespace (any character not visible on the screen); it is
equivalent to [\r\t\n\f].

**perl accepts the IRE and TRE used by grep and sed, except that the curly braces
and parenthesis are not escaped.**
For example, to locate lines longer than 512 characters using IRE:
perl –ne 'print if /.{513,}/' filename # Note that we didn't escape the curly braces

Editing files in-Place
perl allows you to edit and rewrite the input file itself. Unlike sed, you don't have to
redirect output to a temporary file and then rename it back to the original file.

To edit multiple files in-place, use –I option.
        perl –p –I –e "s/<B>/<STRONG>/g" *.html *.htm
The above statement changes all instances of <B> in all HTML files to <STRONG>. The
files themselves are rewritten with the new output. If in-place editing seems a risky thing
to do, oyu can back the files up before undertaking the operation:
        perl –p –I .bak –e "tr/a-z/A-Z" foo[1-4]
This first backs up foo1 to foo1.bak, foo2 to foo2.bak and so on, before converting all
lowercase letters in each file to uppercase.

17. File Handling
To access a file on your UNIX file system from within your Perl program, you must
perform the following steps:
1. First, your program must open the file. This tells the system that your Perl program
wants to access the file.
2. Then, the program can either read from or write to the file, depending on how you
have opened the file.
3. Finally, the program can close the file. This tells the system that your program no
longer needs access to the file.

To open a file we use the open() function.
open(INFILE, "/home/srm/input.dat");
INFILE is the file handle. The second argument is the pathname. If only the filename is
supplied, the file is assumed to be in the current working directory.
open(OUTFILE,">report.dat");   # Opens the file in write mode
open(OUTFILE,">>report.dat");   # Opens the file in append mode

The following script demonstrates file handling in perl. This script copies the first three
lines of one file into another.
#!/usr/bin/perl
open(INFILE, "desig.dat") || die("Cannot open file");
open(OUTFILE, ">desig_out.dat");
while(<INFILE>) {

```
  print OUTFILE if(1..3);
}
close(INFILE);
close(OUTFILE);
```

18. File Tests

perl has an elaborate system of file tests that overshadows the capabilities of Bourne shell and even find command that we have already seen. You can perform tests on filenames to see whether the file is a directory file or an ordinary file, whether the file is readable, executable or writable, and so on. Some of the file tests are listed next, along with a description of what they do.

if -d *filename*	True if file is a directory
if -e *filename*	True *if* this file exists
if -f *filename*	True if it is a file
if -l *filename*	True if file is a symbolic link
if -s *filename*	True if it is a non-empty file
if -w *filename*	True if file writeable by the person running the program
if -x *filename*	True if this file executable by the person running the program
if -z *filename*	True if this file is empty
if -B *filename*	True if this is a binary file
if -T *filename*	True if this is a text file

19. Subroutines

The use of subroutines results in a modular program. We already know the advantages of modular approach. (They are code reuse, ease of debugging and better readability).
Frequently used segments of code can be stored in separate sections, known as subroutines. The general form of defining a subroutine in perl is:

```
        sub procedure_name  {
                # Body of the subroutine
        }
```

Example: The following is a routine to read a line of input from a file and break it into words.

```
sub get_words {
        $inputline = <>;
        @words = split(/\s+/, $inputline);
}
```

Note: The subroutine name must start with a letter, and can then consist of any number of letters, digits, and underscores. The name must not be a keyword.

Precede the name of the subroutine with & to tell perl to call the subroutine.
The following example uses the previous subroutine get_words to count the number of occurrences of the word "the".

```
#!/usr/bin/perl
$thecount = 0;
&get_words;            Call the subroutine
while ($words[0] ne "") {
  for ($index = 0; $words[$index] ne "";
$index += 1) {
   $thecount += 1 if $words[$index] eq "the";
}
&get_words;
}
```

Return Values

In perl subroutines, the last value seen by the subroutine becomes the subroutine's return value. That is the reason why we could refer to the array variable @words in the calling routine.

Conclusion

Perl is a programming language that allows you to write programs that manipulate files, strings, integers, and arrays quickly and easily. perl is a superset of grep, tr, sed, awk and the shell. perl also has functions for inter- process communication. perl helps in developing minimal code for performing complex tasks. The UNIX spirit lives in perl. perl is popularly used as a CGI scripting lan