Perl - The master maniupulator

Perl scripting

- 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.

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.

Example program

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

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.

```
#!/usr/bin/perl
print("Enter your name:") ;
$name = <STDIN>;
chop($name);
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."
```

```
Enter your name:prathi
Enter a temperature in Centigrade: 32
The temperature in Fahrenheit is 89.6
Thank you prathi for using this program.
```

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.

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"

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.

Specifying Filenames in Command Line

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 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.

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.

$. And .. 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

List 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.

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 runtime.

@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.

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 elements

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