Prepared By Dr. N.THENMOZHI M.C.A., M.S., M.Phil., Ph.D.,

Govt, Arts College (Autonomous), Coimbatore-18

Department of Information Technology

OPEN SOURCE TOOLS (18MIT33C) -----II MSc

UNIT-IV: PERL: Introduction – advantages –working environment of Perl – variables – strings –statements –subroutines – files –packages and modules – Object-Oriented PERL.

TEXT BOOKS

1. M.N. Rao, "Fundamentals of open source software", PHI Learning Private Limited, 2015.

REFERENCE BOOKS

1. Larry Wall, Tom Christiansen, Jon Orwart- O Reilly, "Programming PERL",3rd Edition, 2010.

UNIT-IV: PERL: Introduction – advantages –working environment of Perl – variables – strings –statements –subroutines – files –packages and modules – Object-Oriented PERL.

4.1 INTRODUCTION

PERL stands for Practical Extraction and Report Language. It is an interpreted language which is developed by Larry Wall in the year 1987. This language is mainly used for extracting information from arbitrary text files, scanning the text files and printing reports based on the resultant information. PERL was originally developed to manipulate text, but now because of its advantages PERL is used in a wide range of tasks including system administration, GUI development, network programming, etc.

4.2 Advantages and working environment of PERL

PERL is an efficient programming language used to solve problems in a wide spectrum of environments including system programming, bio-informatics, and server-side web programming. It is very easy to develop web applications using PERL and requires less effort than any other web language. PERL is written in the C programming language, which is extensively used to develop much system software, but a C/C++ program statement on UNIX is different when coded for Windows because the libraries on different platforms are different.

4.2.1 Advantages of PERL

- PERL is easy to learn and use. PERL is portable and freely available.
- PERL can be used as a Common Gateway Interface (CGI) for web development.
- Both procedural and object-oriented programming is supported by PERL.
- PERL is extensive and also efficient in text and string manipulation.
- PERL helps in coding programs very quickly.
- Code written using PERL is very dense.
- PERL is supported to work with markup languages like HTML, XML, etc.
- Dynamic memory allocation is very easy in PERL.
- PERL is designed in such a way that it is possible to embed the PERL interpreter with other systems.

4.2.2. PERL Working Environment

Working with the Terminal Window in Windows

We have three different options when we prefer windows.

1. If Cygwin is installed, double-clicking on the Cygwin desktop icon will automatically open command-line prompt where one can proceed further.
2. Run the dialog box which will be appeared on the screen by pressing the ‘windows key’ and ‘r’ key at the same time where one can type cmd (command) and then click ok thereby popping up terminal window.
3. Run the dialog box which can also be appeared by clicking on ‘Start’ thereby ‘Run’ can be seen as one of the menu items. Click on Run and type cmd in the dialog box and click OK.

In case of an alternative option for the standard terminal on windows, one can prefer console as it is available for free.

Download at http://sourceforge.net/projects/console/

Working with the UNIX-style Systems

The users who use Unix style systems may opt for System PERL and PERLbrew. The users who prefer Windows can opt for Cygwin, ActivePERL and Strawberry PERL.

PERL has an elaborate explanation in ‘ PERLdoc ’ , where one can learn the basic structure of the docs and how to look up basic information in PERLdoc. One has to use a terminal when programming PERL. “Using a terminal” will explain how to launch a terminal and run a program from the command line in the documentation.

PERL Basics

Example 4.1 My first PERL program using terminal:

Open your terminal and enter

PERL -e ‘print “Hello, how are you!\n”’

For windows operating system, enter

PERL -e “print \“Hello, how are you!\n\”” (observe the difference in quotations)

Executing PERL programs from the command line.

Program in Normal Mode (Editor)

To run a basic PERL program, type the program, save the program and then type PERL <program_name>. A PERL file should be saved with ‘.pl’ extension. The file name must be any combination of numbers, symbols and letters, but should not contain a space. The symbol underscore (_) can be used in the place of spaces. PERL is case sensitive. C language is structured, its syntax is not as flexible as scripting languages, such as PERL.

Example 4.2 Creation of a simple PERL program:

```
#!PERL
use strict;
use warnings;
use diagnostics;
# this is a comment
print “Hello, World!\n”; # this is the output
```

How it works :

In the command prompt, type PERL followed by the filename, which contains a PERL program. PERL reads the file, parses the code, and executes the file. A comment begins with a sharp (#). It can be on its own line or given in a line after some code.

Strict, warnings, and diagnostics:

The above three statements can be very helpful for executing the program. They are useful in declaring variables and subroutines properly which will warn against doing wrong

things. And if the use of diagnostics line in program is included, they actually give an extensive description of anything wrong and suggestions to overcome it.
Note: The statements "use strict; use warnings; use diagnostics;" are considered for all the programs.

4.3 VARIABLES

Unlike many other languages that concentrate on things, like strings, integers, floats, and so on, PERL does not focus on what the data is but mainly focuses on how you organize the data, rather than what the data is. It has its own powerful approach for data types. A PERL program is a file, which includes many files of text instructions guiding the computer what to do with the given data. The beauty of PERL is that it enables to assign any kind of data to a variable. The 'my' function in PERL is used in the declaration of variables and in making the variation visible in the current scope.

Example 4.3 A few variables:

```
my $name = 'Rao';
my @list_names = ( 'Vedavathi', 'Aruna' );
my $last_name = $name;
print $last_name;
```

Example 4.3 assigns values to some variables. The statement in the third line copies the string Rao from the variable $name to the variable $last_name. The print statement prints the word Rao to the console. The variables with 'my' function in front protects part of program from been accidentally changed by another part where the same variable name was used.

4.3.1 Identifiers

Variables are usually represented by names. The names of the things in PERL are known as identifiers. These are things, such as subroutines, filehandles, packages, and a few others. PERL follows fairly simple naming rules. PERL names or identifiers must start with an alphabet or underscore and optionally follow with one or more alphabets, numbers, or underscores.

Examples 3.4 Valid variables:

```
my $v;
my $xyz;
my $_add_2;
my $Salary;
my $roll_number_5;
```

The following are not valid variables:

```
my $salary-for-month;
my $~release;
my $2nd_variable;
```

4.3.2. Working with Data in Variable Types in PERL

There are three built-in variable types in PERL.

- Scalar
- Arrays
- Hash

Each type of variable existing in PERL is indicated by its own sigil character. Variable names start with a special character known as **sigil**. Some of the sigils used in PERL are '$', '@', '%'. The type of the variable is predefined by these characters in PERL. Dealing with the variables in PERL is a straightforward way, since it is not mandatory to define and allocate some memory to them. So, PERL does not include any technique for memory reallocation.

Scalar Variables

Scalar variable starts with sigil '$'. Scalar variables are those which contain only one element, it may be a string or a number or a reference. PERL will automatically convert them as required. Strings are defined as sequences of characters including letters, numbers or symbols. The numbers consist of integers, decimals or exponent values. The memory address of a scalar, array or hash variable is contained in the reference which is a scalar value. In PERL, a scalar is merely a single value. Here are some scalars:

```
my $name = 'Rao';
my $today_temp = 42;
```

A scalar can be a string, a number, or a reference. In PERL, its meaning is a single value. If anything is not assigned to the variable, it will be as a value called undef.

```
my $xyz_var; # here its value is undef
```

If the value of the variable is undef it often displays 'uninitialized' warnings in the program. We can declare several scalars at once by keeping parentheses around them, as shown below:

```
my ( $today_temp, $city_name );
```

After declaring them by putting parentheses at right side, values can be assigned to them as shown below:

```
my ( $today_temp, $city_name ) = ( 45, 'delhi' );
```

The above statements assigns 45 to $today_temp and "delhi" to $last_name. As we have mentioned above that scalars can be numbers, or a string.

Numbers:

Numbers include data types, like integer, floating point and non-decimal integers. Number data types are explained as follows:

(i) Integers:

All the whole numbers along with negative numbers, except fractions are called integers. Integers are often expressed as decimal integers with base 10. Integer literals are straightforward.

Example 4.5 Integers:

0
2014
−45
666
81923456222

(ii) Floating point numbers:

They store real numbers. In PERL, the floating-point numbers are represented in two forms. One is the fixed method and another is the scientific method. In fixed form of floating-point numbers, the decimal point is fixed on the number to denote the fractional part. Floating-point numbers consist of significant which contains significant digits of the number, and an exponent which represents the power of 10 that the significant is multiplied by.

Example 4.6 Floating point numbers:

1.48 366.000
366.0
3.25e55 # 3.25 times to 10 to the 55th power
−3.5e46 # negative 3.5 times 10 to the 46th power # (comment–a very big negative number)
−14e−34 # negative 14 times 10 to the −34 th power # (comment–a very small negative number)

(iii) Non-decimal integers:

PERL allows to specify numbers in other ways than decimal. Octal (base 8) number starts with a leading 0, hexadecimal (base 16) number starts with a leading 0x, the hex digits A to F represent the conventional digit values of 10 through 15, and binary (base 2) number starts with a leading 0b.

Example 4.7 Non-decimal integers:

0467 # 467 octal number representation
0x12ff # 12FF hex-decimal number representation
0b1111 # a binary number representation

Array Variables

Array variable contains a list of elements (scalar data) and there is no restriction on the number of elements in the list. The variables which are prefixed with sigil @ are called array variables. The first element in the list will be accessed by index zero [0] and the next element will have the index one [1] and so on.

The element of an array variable starts with the '$' character followed by an array variable name and square brackets which include the element index. This is the way of accessing an element existing in an array in PERL. The '$' character is used because the array variable in PERL includes a list of scalar data.

Example 4.8 Creation of array variables:

@cities = ("Mumbai", "Hyderabad", "Delhi", "Pune");

As per Example 4.8, to access an element in the array variable cities, one should use the notation $cities [index number]. So, to get the first element of the array, the notation is $cities [0], i.e., the string "Mumbai" will be displayed and the second element can be accessed with $cities [1] and so on.

The following point should be remembered while using array variables:

- The index of the last element of an array can be known by using special variable $array.
- In general, arrays are zero indexed, but this general setting can be changed by modifying the default variable $ARRAY_BASE.

Slicing elements of an array:

Slicing an array is a process to create a new array with some of the elements retrieved from an existing array. There exists no slice () function in PERL to do this operation. The following process should be followed for slicing in PERL:

@new array-name = @old array-name [list of valid indices separated by comma]

Example: @capitals = @cities [0, 1, 2, 3, 4, 5];

Instead of specifying all the indices, it is also allowed to use the range operator (..) between the indices as given below:

@capitals = @cities [0..5];

Adding an element to an array:

The element can be added to array in two ways as per requirement. It means that an element can be added at the end of an array or at the beginning of the array. To do these operations PERL provides the following two functions:

- Push(): This function adds an element at the end of the array. It takes two arguments, the first argument should be the array name and the second argument should be the name of the element to be added to the array or list of elements to be added to the array.
- Insight(): It adds an element at the beginning of the array like push() function, it also takes two arguments.

The general form of push() function is:

push array, list;

Example 4.9 Usage of push() function:

push @numberarray, (3..6);

The general form of unshift() function is:

unshift array, list;

Example 4.10 Usage of unshift() function:

unshift @numberarray, 1; #operation is at the beginning of the array

Removing an element from an array:

It is possible to remove an element from the beginning of an array and also from the end of the array. PERL includes two functions to perform these operations. The functions are explained below:

- pop(): It deletes the last element of an array. It includes only one argument which is nothing, but the name of the array.
- shift(): This function helps in removing the first element from the specified array provided as argument.

It is allowed to remove an element from an array using index number. The following function is used to perform this.

delete $array[index]: It deletes an element from the entry of DBMS file, the topmost point shows next top after deletion of top element.

The general form to pop() function is:

pop array;

Example 4.11 Usage of pop() function:

@my lastelement = pop @numberarray;

The general form of shift() function is:

shift array;

Example 4.12 Usage of shift() function:

@my firstelement = shift @numberarray; #operation is performed on the first element of array

The general form of delete() function is:

delete array[index];

Example 4.13 Usage of delete() function:

delete @numberarray[10];

Other Built-in Functions of array variable includes the following:

reverse (@array-name): It reverses the order of the elements stored in an array. The array name should be passed as argument to this function.

Example 4.14 Usage of reverse() function:

@arrayxyz = reverse @arrayxyz;

sort (@array-name): This function sorts the elements of array in two ways, one way is to sort alphabetically in ASCII order and the second way is to do sort process numerically . In different versions of PERL different sorting algorithms were used, like quick sort, merge sort, etc.

Example 4.15 Usage of the sort() function:

@arrayxyz = sort @files; # this command sort lexically the files and stores in arrayxyz

splice (@array-name, offset, length, new-elements):

The first argument should be the array name in which some of the array elements are to be replaced. The second argument is the offset which indicates the starting array element to be removed, if this offset is negative then it will start from the end of the list. The third argument is the length representing the number of elements to be removed from offset. The fourth argument is the new elements which indicate the list of elements to be replaced with the removed elements from the array. The size of the array can be increased or decreased accordingly.

Example 4.16 Usage of splice() function:
splice(@array, -1); # it removes the last element of array.

Hash Variables

Hash variables are prefixed with a sigil '%'. A hash is a special kind of an array where each element consists of a key and a value associated with it. Hashes are very complex data types. While accessing the elements of an array, the index must be a number, whereas in the case of accessing an element of a hash variable, the hash index could be a number, or a string. All the keys within a hash must be unique. A hash allows a programmer to create one-to-one associations between a variable, or 'key', and its value. The most well-known special hash variable is %ENV which contains environment variables. Hashes are one of the most important features of PERL.

Example 4.17 Creation of hash variables:

```
my %students = ( "sarat", 1, "ramesh", 2, "sangam", "excellent", );
print $students{'sarat'}; # the above code is equal to declaring @array=( 1,2,'excellent')
```

The output generated by the program is:

1

```
my $name = $student{ 'Ramesh' }; # output for this is 2
$students{rakesh} = 'raju'; # this how we add data–a new value raju to hash named students

%students = (%students, rakesh => 'raju', krishna => 'sanjay'); # adding multiple values in hash
```

Arithmetic Operations

The arithmetic operators, like +, −, * and / are used for addition, subtraction, multiplication and division, respectively. Initially precedence, multiplication and division are calculated. Then addition and subtraction are calculated. The associativity is from left to right in both the cases.

Example 4.18 Prints 12 as result:

```
my $result = 8 + 4 / 2 * 2;
print $result;
```

To prevent visual confusion, the authors generally avoid parentheses, but they are strongly recommended. To avoid confusion we can also write Example 4.18 as:

```
my $result = 8 + ( ( 4 / 2 ) * 2 );
print $result;
```

If one wants the addition first, followed by the multiplication and then division, just use parentheses to group things logically.

```
my $result = ( 8 + 4 ) / ( 2 * 2 );
print $result;
```

Now, we obtained 3 as the answer instead of 12. To handle exponentiation use ** operator. To calculate the cube of 15, the following code is to be considered.
print 15 ** 3;

The result is 3375.

4.4 STRINGS

Literal strings can be represented in two different ways: single-quoted string and double quoted string . A single-quoted string literal is a sequence of characters, enclosed in single quotes (‘ ’). The single quotes are used in strings to identify the starting and the ending of the string, but they are not part of the string. Here the backslash character (\) does not have full energy. The limitations with single quote usage is in the newline character ‘\n’ is not considered as newline with single quotes as generally done in case of double quotes.

Example 4.19 Single quoted strings:
‘welcome’ .... #interpreted as w,e,l,c,o,m,e
‘welcome\n’ .. #interpreted as w,e,l,c,o,m,e,\,n
‘welcome to home’ ..... #interpreted as w,e,l,c,o,m,e newline t,o, , h,o,m,e
‘\’’ ..... #interpreted as a single quote, as single quote given after ..... the backslash
‘\\’ ..... #interpreted as backslash\, as two backslash are kept in a row
‘’ ..... #interpreted as a null string, which is a space character

The substr() function considers a string and displays as given by the parameters for it. The general form of substr() function is:

substr offset , length

Example 4.20 Usage of substr() function:

```
my $string = ‘welcome’, 0, 3;
my $substring = substr $string, 0, 3;
print $substring;
```

The output generated by the program is:

wel # as mentioned in the substring, from 0 th element three characters(3) are displayed

The double quoted string literal is a sequence of characters, enclosed in double quotes (“ ”). The double quotes and backslash are used for control characters and also for representing octal and hexadecimal numbers.

Example 4.21 Double quoted strings:
“welcome” ..... # same as single quotes no special effect
“welcome\n” ..... # now backslash character works and takes as a newline
“welcome \t mam” .. # tab special character works here

In the same way \a for bell (alarm), \f form feed, \r return, \e for escape, \\ for backslash can be used.

String concatenation is done by "." dot operator. The general form of string concatenation is:

$string = $string . $string ;

Example 4.22 How to concatenate the strings:

```
my $string1 = "ravi";
my $string2 = "revanth";
my $stringconcat = $string1 . $string2;
print $stringconcat;
```

The output generated by the program is:

ravirevanth

Example 4.23 Numbers as strings:

```
my $string1 = "10";
my $string2 = "5";
my $stringconcat = $string1 + $string2;
my $stringconcat = $string1 - $string2;
print $stringconcat;
```

The output generated by the program is:

105

5

Another magic operations to strings are auto increment (++) and auto decrement (--), the increment will take to the next letter, and decrement brings the previous letter.

Example 4.24 How to increment the value of a variable:

```
my $stringinc = "b";
$stringinc++;
print $stringinc;
```

The output generated by the program is:

c

4.5 STATEMENTS

Statements can be defined as a complete unit of instruction for the computer to process. The semicolon (;) at the end of each statement is very important. It indicates that the PERL statement is complete. Statements are used in a language in order to process or evaluate the expressions. The statements which are kept between an opening brace ({) and closing brace (}) can be called a block. A block may include one or more PERL statements. Every block statement will be treated as an individual statement by PERL. There are two kinds of statements, namely, conditional statements and control statements.

4.5.1 Conditional Statements

The conditional statements existing in PERL are 'if', 'unless' and 'switch'. The conditional statements are used to test whether the specified condition in the statement is true or

not, and thereby, deciding either to continue the execution of scripts or jump to a particular statement.

if statement

This statement is the primary conditional structure in PERL which is used to execute a block of statements if the given condition is true. The general form of 'if' statement is:

```
if(Boolean expression)
{
        Statement 1;
        Statement 2;
        . . .
        Statement N;
}
```

If the result of Boolean expression is true then the statements in the block will undergo execution.

Note: The braces related to the block are mandatory even if the block contains only a single statement.

Example 4.25 Usage of 'if' statement:

```
my $rajesh = "kavin";
if ($rajesh gt 'avinash')
{
        print "'$rajesh' comes after 'avinash' in sorted order.\n";
 } # the above condition is true
```

In addition to the above syntax, it is also allowed to write the 'if' statement as follows if only single statement should be executed provided the Boolean expression evaluates to be true.

The alternative general forms of 'if' statement is: "statement" if (Boolean expression); In some situations, it is necessary to execute another block of statements if the Boolean expression specified in 'if' statement evaluates to be false. This block of statements is called as 'else' block is: The general form of using else block is:

```
if (Boolean expression)
{
        Statement 1;
        Statement 2;
        . .
        Statement N;
}
else
{
        Statement 1;
        Statement 2;
        .
        .
        Statement N;
}
```

In a simple way, the general form of using block code can also be written as follows:

```
if (Boolean expression)
        BLOCK
else
        BLOCK
```

Example 4.26 Usage of "ifelse" statement:

```
if ($rajesh gt 'avinash')
{
        print "'$rajesh' comes after 'avinash' in sorted order.\n";
}
else
{
        print "'$rajesh' does not come after 'avinash'.\n";
        print "this is the second statement.\n";
}
```

elsif statement

This statement is used to test multiple conditions and executing unique block of statements for each condition is satisfied. The general form of 'elsif' statement is:

```
if (expression or condition)
{
        Statements;
}
elsif (condition)
{
        Statements;
}
elsif (condition)
{
        Statements;
}
```

In a simple way, the general form of 'elsif' statement can also be written as:

```
if (condition)
        BLOCK
elsif (condition)
        BLOCK
elsif (condition)
        BLOCK
.....
```

The block of statements will be executed only if the Boolean expression or the condition related to that block results in true. In this type of checking multiple conditions using elsif statement, it is possible to add else block to the existing construct to make them execute if no condition is satisfied. The general form of adding else block to elsif statement is:

```
if (condition)
{
        Statements;
}
```

```
elsif (condition)
{
        Statements;
}
elsif (condition)
{
Statements;
}
. .
else
{
Statements;
}
```

The else block will be executed if the conditions specified in the 'if' statement and 'elsif' statement evaluate to false. Alternatively, the general form of adding else block to elsif statement can also be written as follows:

```
if (condition)
        BLOCK
elsif (condition)
        BLOCK
elsif (condition)
        BLOCK
        .....
else
BLOCK
```

```
#! /usr/ bin/PERL
print "Enter the number";
my $num= <STDIN>;
 if ($num <=25)
{
        print "The number entered is less than 25";
}
elsif ($num <=50)
{
        print "The number you have entered is greater than 25 and less than 50";
}
elsif ($num<=75)
{
        print "The number you have entered is greater than 50 and less than 75";
}
else
{
        print "The number entered is greater than 75";
}
```

unless statement

It is opposite to 'if' statement. The block following the 'if' statement will be executed if the condition specified is true, whereas in the case of 'unless' statement, the block gets executed if the condition specified is false. The general form of 'unless' statement is:

```
unless (Boolean expression)
{
        Statements;
}

(or)
unless (condition)
        BLOCK
```

If the Boolean expression results in false then only the control takes up the block for execution, otherwise it will continue the execution process from the immediate statement after the block. If single statement needs to be executed then the general form of 'unless' statement is: The alternative general forms of 'unless' statement is:

"statement" unless (condition)

Note: The else and elsif clauses can also be used with this 'unless' statement. The syntax of else and elsif clauses with 'unless' statement is similar to the 'if' statement except, the only difference is that 'if' should be replaced with 'unless'.

Example 4.27 Usage of 'unless' statement:

```
unless ($age< 20; $count++)
{
        a = b + count;
        print "this is example for count : $count\n";
}
```

4.5.2 Control Statements

The control statements are an important category of the Perl statements. The main purpose of Perl control statements is to change the sequential flow of a Perl program.

Loops

In some cases, It is necessary to execute a particular block by several times depending on the specified condition evaluates to be true. The repeated execution of a block can be called a loop. There are four loops available in PERL which are as follows:

- for loop
- while loop
- until loop
- foreach loop

for loop:

This loop works similar to C and other languages. It executes a block of statements iteratively till the condition mentioned in the statement results to be false. The general form of 'for' loop is:

LABEL for (initialization; condition; increment or decrement) BLOCK

Initialization assigns some specific value to the variable specified, and executes only once at the beginning of for statement. Then the condition will be checked, if it evaluates to be true then the block will be executed. After that the increment or decrement of the variable mentioned will take place and again the condition will be checked. If the condition results in 'true' then again the block will be executed. This process goes on until the condition is not met.

Example 4.28 Usage of for loop:

```
for ($count= 1; $count< 20; $count++)
{
a = b + count;
print "this is example for count : $count\n";
}
```

Note: The LABEL before the 'for' in the above syntax is an optional identifier followed by a colon. LABEL is very useful when there is a need to alter the normal flow within the block by using the loop controls (next, last and redo). The braces related to block should not be omitted even if a single statement exists.

while loop:

This executes the block repeatedly until the specified condition fails. The general form of while loop is:

```
while (condition)
{
        Statements;
}
```

If the condition evaluates to be true then the control executes the statements inside the block. The condition will be checked every time for each iteration of the loop. Loop termination takes place when the condition results to be false.

Example 4.29 Usage of while loop:

```
$count = 0;
while ($count < 15)
{
        $count += 3;
        print "count value is now $count\n"; # output values 3 6 9 12
}
```

The general form of while loop when continue block is included:

LABEL while (condition) BLOCK continue BLOCK

In this case, LABEL is an optional identifier followed by a colon (:), if it is included it must be in uppercase. It includes an optional block continue which will be executed after each current iteration. In simple words, it can also be said that the continue block will undergo execution before the successive re-evaluations of the test condition. In case of use of the three looping controls, namely, next, last or redo then the label can use as a support for jumping to or from within the block.

until loop:

It is just the reverse of while loop. This until loop executes the block as long as the condition remains false. Initially, if the condition is true the entire block is skipped. The block will undergo repeated execution until the condition becomes true. The general form of 'until' loop is:

```
until (condition)
{
        Statements;
}
```

Example 4.30 Usage of 'until' loop:

```
my $factnum= 1;
my $i= 1;
until ( $i > 10 )
{
        $factnum *= $i++;
}
print $factnum; # this script prints factorial of 10
```

The general form of until loop when continue block included is:

```
LABEL until (condition) block continue block
```

Note: Either LABEL or continue block are optional.

foreach loop:

This loop has its applications in areas of lists and arrays. If one wants to iterate over every element of a list or an array to execute some particular code for unique element, then this loop will be very useful. The general form of 'foreach' loop is:

```
LABEL foreach VAR (LIST) BLOCK
```

LABEL is an optional identifier in uppercase followed by a semicolon. VAR is a variable which helps to iterate all the elements in the specified list or an array foreach iteration. LIST represents a list of scalars or an array that points to a list. A BLOCK must be enclosed by curly braces ({}) even if single statement exists.

Example 4.31 Usage of 'foreach' loop:

```
$find = "cherry";
$message = "none\n";
@elements = ("rajesh", "shanthi", "cherry", "laxmi", "kishore");
```

```
foreach $name (@elements)
{
        if ($name eq $find)
        {
          $message = "$find name is found!\n"; # assignment is done if given string is
                        found in the array last;
        }
}
print $message; #prints message if it is found or not
```

The general form of foreach loop when continue block included is:

LABEL foreach VAR (LIST) BLOCK continue BLOCK

Note: This continue block is optional. If it is present, it will be executed after each iteration.

Loop Controls

PERL provides three loop controls, namely, next, last, and redo.

next:

Placing this loop control inside the loop will stop the current iteration and go on to the next one. The general form of 'next' loop control is:

Label: while (condition) { .... next Label; }

last:

This control immediately terminates execution of the loop by skipping the remaining statements in the block. This is similar to the break keyword in the C language. 'Last label expression' can also be given in the script. The general form of 'last' loop control is:

Label: while (condition) { last Label ; .... }

redo:

It will execute the same iteration again without re-evaluating the conditional statement for the loop. The general form of 'redo' loop control is: Label:

while (condition) { .... } redo Label;

4.6 SUBROUTINES

PERL allows the user to define their own functions named as subroutines . To perform a particular task a separate code is defined and can be called anywhere across programs. A subroutine can be reusable and can be defined anywhere in the program. PERL program executes subroutine in the program by calling or invoking it, and this act of invoking a subroutine is called subroutine invocation. If the subroutines had defined in another file, it is possible to load them in any program by using use, do or require statement.

4.6.1 Uses of Subroutine

- It breaks the program into smaller parts which is very easy to read and understand.
- The same piece of code which performs the same task can be used multiple times.
- PERL subroutines are flexible and powerful.

A subroutine can be defined by using the keyword 'sub' followed by the subroutine name. The 'sub' keyword is case-sensitive and should be in lower case. The subroutine name starts with a letter followed by one or more letters, numbers, or underscores. A subroutine name must be chosen in such a way that it should not belong to PERL built-in functions. The general form of subroutine is:

sub subroutine-name { ….. }

Prototypes Attributes Block 'Prototypes' indicates PERL what parameters the subroutine expects. 'Attributes' gives a subroutine additional explanation. Both attributes and prototypes are optional. 'Block' consists of the code to be executed whenever subroutine is called.

Example 4.32 Subroutine:

```
sub subroutine-name
{
        Code to be executed
}
```

A subroutine can be called anywhere in the program by two ways. One way to call a subroutine by prefixing ampersand (&) to the name of subroutine. Another way of calling subroutine is by specifying the subroutine name with parenthesis. The general form for calling a subroutine in the above mentioned two ways are as:

&subroutine-name, and subroutine-name ().

4.6.2 Return Statement in Subroutine

There are two methods to return a value from a subroutine. One method is called an implicit method which makes the value, or any expression, or a list, or a single scalar to be returned from subroutine as the last statement in it. This implicit method is very easy. The second method is to return a value from a subroutine is by using the return statement explicitly. This method can be called explicit method. The return value can either be a scalar value or a list. The general form of 'return' statement is:

```
return (return-value);
```

The return-value is the value that the subroutine needs to return. If the subroutine does not include return statement, it will return the last value calculated. A subroutine may consist of multiple return statements, the first one that is executed will exit the subroutine.

my variables

Variables which are used in a subroutine can be made private to itself with the help of my operator. It helps to avoid overwriting similarly-named variables in the main program. Since, the subroutine body is considered as a block, therefore, the variables only exist within the body of the subroutine.

Example 4.33 The subroutine defined before the script:

```
Sub name #defining subroutine name
{
        print "hi this is subroutine test\n";
}
for ($n= 1; $n<5;$n++)
{
```

```
        name;
    } # calling subroutine 4 times using a for loop
```

The output generated by the program is:

```
hi this is subroutine test
hi this is subroutine test
hi this is subroutine test
hi this is subroutine test
```

In Example 4.33, there is no return value sending back, it displays the value from the subroutine. In some cases the return value may be sent to the calling function by return keyword. The subroutine defined in Example 4.33 shows how the return keyword is used in subroutine. It is also possible to create anonymous subroutines in PERL. Subroutines without a name can be known as anonymous subroutines.

Example 4.34 Usage of subroutines:

```
    sub total (@)
    {
        my $total = 0;
        for my $temp (@_)
        {
            $total += $temp;
        }
        return $total;
    }
#this subroutine named total prints the total of numbers, by taking through temp variable.
```

Example 4.35 Usage of subroutines:

```
    sub square_root
    {
        my $element = shift;
        $element **= 1/2;
        return $element;
    }
    $element = “stanley”;
    $n = square_root(16);
    print “$element says the root is $n\n”;
```

The output generated by the program is:

```
Stanley says the root is 4 #this subroutine named square_root output is 4
```

Handling arguments in the subroutine are like this:

```
  sub students_count
  {
    my $number_of_students = shift; #shift for pointing first value in an array my
    ($number_of_students, $number_of_faculty) = @_; # handling multiple arguments ....
```

```
}
```

Shift() function is used to move the entire array towards the left hand side. Therefore, one can start reading the first element of an array from left. We can declare shift() function explicitly.

```
my $number_of_students = shift @_;
```

The subroutine for making private in PERL use underscore (_) prefixed to the name of the subroutine. This is useful when the subroutine is called by other codes, then they will realize it as private.

4.6.3 Types of Subroutines

There are two types of subroutines, namely, nested subroutines and recursive subroutines. Nested Subroutines It is possible to call a subroutine from other subroutines known as nested subroutines since one subroutine is nested inside another subroutine.

Example 4.36 Usage of nested subroutines:

```
sub first { ...
}
sub second{
 ... }
sub third{
        my ($arg) = @ARGV;
        if ($arg == 1)
        {
                first();
        }
        elsif ($arg == 2)
        {
                second();
        }
}
```

Recursive Subroutines

In addition to nested subroutines, there exists another category of subroutines which can actually call themselves. The subroutine which calls itself can be defined as recursive subroutines. A subroutine can be used as a recursive subroutine if the following conditions are true. All the variables used by the subroutine are local. The subroutine should contain code that determines when it should stop calling itself.

Example 4.37 Usage of recursive subroutines:

```
        my $recfunc = sub
        {
                my ($recfunc, $paramt1, $paramt2) = @_;
                do { -----lines of code here---};
```

```
        $recfunc->($recfunc, $paramt1, $paramt2);
}
```

4.7 FILES

When the PERL program connects to the external data file, PERL names the connection as a Filehandle. In simple words, a file handle can also be understood as a dummy name for the files one wants to use in PERL scripts. A handle is nothing, but a temporary name assigned to a file. All input and output to files are achieved with the help of Filehandling. Filehandles provide a means for one program to communicate with another program. A variety of operators and functions are used to read and update the data associated with the Filehandle. The three main file handles are STDIN, STDOUT and STDERR.

4.7.1 Opening a File

Files in PERL are opened by using open and sysopen function. Both the functions may take up to four arguments. The first argument is the Filehandle, second is the filename or the file path, third argument is the mode and the last argument includes the permissions to be granted to a particular file.

Open function

In general, the open function includes two arguments. The first argument is the file handle which is returned by open function and the second argument is EXP which indicates the expression having file name and the mode in which the file is opened. There exist mainly three modes in which the file can be opened, namely, 'read' mode where the file is to be opened for reading purposes, 'write' mode where the file is opened for writing purposes, and 'append' mode where the file is opened to add the required contents to the end of file. The general form to open() function is:

open (FILEHANDLE, "<filename"); here '<' sign indicates file is opened for read mode.
open (FILEHANDLE, ">filename"); here '>' sign indicates file is opened for writing mode.
open (FILEHANDLE, ">>filename"); here '>>' sign indicates file is opened in append mode.

Example 4.38 To open a file in reading mode:

```
#!/usr/bin/PERL open(DATA, "<my file.txt")
while(<DATA>)
{
        print "$_";
}
```

sysopen function

Sysopen function is similar to the open function except that it uses the arguments supplied to it as the arguments to the system function. The sysopen() function takes three arguments, such as, filehandle, filename and mode.

The different modes available regarding to sysopen() function and their description are as follows:

O_RDONLY ..... − This is read only mode
O_WRONLY ..... − This is write only mode

O_RDWR ..... − This is read and write mode
O_CREAT ..... − This mode helps to create the file
O_TRUNC ..... − This mode helps to truncate the file
O_APPEND ..... − This mode helps to append the file
O_NONBLOCK ... − This is non block mode
O_EXCL ..... − This mode stops if the specified file already exists

Example 4.39 Program for reading operation:

```
#! /usr/bin/PERL
sysopen (FH, "/tmp/text", O_RDONLY);
$lines=<FH>;
print $lines;
```

4.7.2 Reading From a File

Whatever the operation be–whether it is an input or output operation that is to be performed by the file will go through Filehandle. It is allowed to read lines from files and input them by using the input operator '< >'.

File permissions

Permissions are needed to assign to files in order to implement security and authorized access to it. A PERL file (. pl) to function on the web server has executable file permissions. The following list of things helps to pass to the open function as per the required operation.

Entity	**Meaning**
< or r	-This provides read only access permission to the file.
+< or r+	-This provides reads and writes permissions to the file.
> or w	-This provides creates, writes and truncates permissions to the file.
+> or w+	-This provides reads, writes, creates and truncates permissions to the file.
>> or a	-This provides writes, appends and creates permissions to the file.
+>> or a+	-This provides reads, writes, appends and creates permissions to the file.

close function

To close a Filehandle related to a corresponding file, this close function is used. The close function indicates PERL to finish with that file. The general form of close() function is:

```
close FILEHANDLE;
```

It is also allowed to specify this function as simply close without specifying Filehandle. If no Filehandle is mentioned, the currently selected Filehandle is closed. If the buffer flushes correctly and closes the file then it returns true.

getc function

The getc function is used to return a single character from the given Filehandle. The Filehandle does not return whatever it reads from STDIN. The function returns some undefined value if there is any error, or if the Filehandle is at the end of the file. The general form of getc() function is:

```
getc Filehandle;
```

read function

To read the information from the buffered Filehandle as a binary data one can use the read function. The general form of read() function is:

```
read FILEHANDLE, SCALAR, LENGTH, OFFSET;
read FILEHANDLE, SCALAR, LENGTH;
```

It will read specified LENGTH data characters into the variable named SCALAR from the given FILEHANDLE. It returns the number of characters actually read on success, returns zero at the end of file and finally an undefined value if an error occurs.

Generally, an OFFSET is used to read when the data is stored in between other than the beginning. The specified OFFSET can be either negative or positive. A positive OFFSET which is greater than the SCALAR length with a size of "\0" bytes is padded before the result is appended. A negative OFFSET represents characters backwards from the end of the string.

print function

This function is used to print a string or a list of strings. This will return true upon success. The general form of print() function is:

```
print FILEHANDLE LIST
print FILEHANDLE
print LIST
print
```

The Filehandle may be a scalar variable name, where the variable may contain the name or reference to the Filehandle, thereby introducing one level of indirection. The main thing which needs to remember here is, if the Filehandle is a scalar variable. The following token is a term, it may be misunderstood as an operator unless one specify a + or keep parentheses around the arguments. If Filehandle is not specified, by default the print function prints to standard output or to the output channel selected at last. If LIST is also not specified, it prints $_ to the currently selected output channel. One can use the select operation to select the default output channel to something else, but not to STDOUT. This function takes a LIST, in which anything in it is evaluated in list context, including any subroutines whose return lists all passed to print. One should remember, that print keyword should not be followed by left parentheses except when one wants the right parenthesis to end the arguments for printing.

tell function

The general form of tell() function is:

```
tell FILEHANDLE
tell
```

This function returns the current position in bytes for Filehandle. If an error occurs it returns −1. The Filehandle may be any expression. The value of the expression is the actual name of the Filehandle. If Filehandle is not specified, it will assume the last file read. It is

recommended not to use the tell function on a Filehandle that has been manipulated by sysread(), syswrite() or sysseek(). Those functions ignore the buffering, whereas tell() does not.

seek function

This function will place the file pointer to the specified number of bytes within a file. The general form of seek() function is:

seek FILEHANDLE, POSITION, WHENCE

The Filehandle may be any expression whose value represents the name of the Filehandle. The WHENCE can take any of the three values, namely, 0, 1, 2. If the value of the whence is zero (0), set the new position in bytes to POSITION. If it contains the value one (1) it sets the new position to the current position plus POSITION, and finally the new position is set to EOF plus POSITION for the value two (2). It is also allowed to use the constants SEEK_SET (represents the start of the file), SEEK_CUR (represents the current position) and SEEK_END (represents the end of the file) for WHENCE.

Example 4.40 Usage of seek() function:

```
seek DATA, 250, 0;
```

Zero (0) indicates the position relative to the start of the file. The line in the file set the file pointer of the file to the 250th byte.

4.7.3 Copying Files

File duplication can be done by using the copy function. This function takes two arguments, the first argument is the URL of the file to be copied and the second argument is the URL of the new file. If the same file name or the same URL is used, PERL will rewrite the file if permissions allow.

Example 4.41 Copying of files:

```
#!/usr/bin/PERL
open (DATA1, "<my file1.txt");
open (DATA2, ">my file2.txt");

while (<DATA1>)
{
        print DATA2 $_;
}
close (DATA1);
close (DATA2);
```

4.7.4 Moving Files

Move function will help in moving files. The difference between copying and moving files is while copying files two copies will exist, whereas in the case of moving files only one copy will exist and only the location of the file will change. While moving a file from one

location to another location, the file in the old location has been completely removed and placed in the new location. The general form to move the files is:

move (old location, new location)

Example 4.42 Moving of files from one location to another location:

```
#!/usr/bin/PERL
$oldlocation= “file1.html”;
$newlocation= “folder1/file1.html”;
move($oldlocation, $newlocation);
```

4.7.5 Renaming Files

The file name can be changed to another name using a rename function. It takes two parameters, one is the old file name, second is the new file name. The general form to rename a file:

rename (old file name, new file name)

Example 4.43 Renaming of files:

```
#! /usr/bin/PERL
rename (“/usr/test/filename1.txt”, “/usr/test/filename2.txt”);
```

4.7.6 Deleting Files

An existing file in PERL can be deleted by using unlink function. The best option to delete a file is to set a variable name equal to the URL of the file to be deleted. It is also possible to remove multiple files at once by first creating an array of files to be deleted and then looping through each one. The general form to delete a file is:

unlink (file-name or filepath).

Example 4.44 Deletion of files:

```
#! /usr/bin/PERL
unlink (“/usr/test/myfile1.txt”);
```

4.8 PACKAGES AND MODULES

Packages and modules are related to each other, but they are independent concepts.

4.8.1 Packages

It is essential to know about namespace before package. The collection of unique variable names can be called a namespace. It will prevent variable name collisions between packages. A package defines a namespace where a module’s functionality or data resides. A namespace can also be called symbol tables. PERL files with .pm extension can be called packages and can also be considered as a separate namespace. It is possible to have many packages in a single file and

also a single package that spans several files. In simple, we can also say that packages are namespaces where the functions and variables can be placed.

Package Declaration

To declare that a code belongs to a particular namespace, the package directive should be used. Package statement helps in switching the current context to the mentioned namespace. If the specified namespace does not exist, a new namespace is first created. The package statement will be in effect until either another package statement is invoked or until the end of the current block or file. The variables can be referred explicitly within a package using the :: package qualifier.

4.8.2 Modules

A module is a file which contains related functions and variables. Other modules and scripts can re-use these modules. At Run time we can load these modules and can call their functions also. A module can contain several packages. A module can be loaded by calling any one of the two functions, namely, use and require.

use function

The use function is called to load a module. The syntax is to write the 'use' function followed by the name of the module. The use statement will be evaluated at compile time. The general form of 'use' function is:

use MODULE
use MODULE LIST
use VERSION

This use function will import all the functions exported by specifying MODULE or only those referred to by giving LIST into the namespace of the current package. The VERSION argument can also be specified in between the MODULE and LIST. If so, then the VERSION method by the use functions in class MODULE with the specified version as an argument. If simply VERSION is specified as an argument, VERSION may be either a positive decimal fraction or a v-string of the form v5.6.1. An exception may be raised if the specified VERSION is greater than the version of the current PERL interpreter, and then PERL will not try to parse the rest of the file.

require function

In addition to use function, a module can also be loaded over required scripts by using require function. The subroutine names with respect to a module which is mentioned using require function must be fully qualified. The require statement will be evaluated at execution time. The general form of 'require' function is:

require
require EXPR

If the EXPR is specified as numeric, it represents that the script requires the given version of PERL in order to execute the script. If both EXPR or $_ are not numeric, then it includes the given name of a library file to process the script. It is not allowed to include the same file with this function twice. Whatever the file is included, it must return a true value as the last statement.

Table 4.1 shows the difference between use and require function. Even though there exist differences and they behave differently, but they achieve the same goal.

Table 4.1 Differences Between use function and require function

use function	Require function
This is used only for the modules, ie., only to include ,pm type file.	This is used for both libraries and modules
The included objects will be verified at compilation time.	The included objects will be verified at Runtime,
There is no need to give file extension here, ie., only file name is required	There is a need to give file extension.
This function will implicitly call the import method of the module being loaded.	This function does not implicitly call the import method of the module being loaded.
use does not behave like a function	Requires behavior like a function

BEGIN and END Blocks

The BEGIN and END blocks are the two special code blocks for PERL modules. When a module is loaded, the BEGIN block gets executed and when the interpreter of the PERL is about to unload the module, the END block is executed. The BEGIN and END blocks act as constructors and destructors respectively. The BEGIN and END blocks are written as follows:

```
BEGIN
{
. .
}
END
{
. .
}
```

4.9 OBJECT-ORIENTED PERL

In general, Object-Oriented Programming also called OOP is a software design paradigm helps in writing more re-usable and elegant code by declaring classes which define the functionality of their various instances called objects. In procedural programming, the main criteria are on splitting a project into smaller and smaller tasks, using subroutines to define a single task and so on, whereas in the case of object-oriented programming the main focus is on data. Objects which include data will have various properties and can interact with each other in various ways. Object-oriented programming is mainly concerned with groups of actions and interactions between data. The basic unit of operation in object oriented programming is the object.

4.9.1 Objects

An object is that which contains data on its attributes or properties, and can perform actions through methods. The object is a way of accessing data and can be referred as an instance of a class. An object in PERL can simply be called a reference.

4.9.2 Classes

A class in PERL can be treated as an ordinary package which contains methods required for creation and manipulation of objects. Classes define the methods an object can have, and how those methods work. In general, a class can also be understood as a description of the attributes of an object and the manner of accessing and modifying those objects. To create a class in PERL, it is necessary to build a package first. A package is defined as a unit of self-contained user-defined variables and subroutines, which can be re-used. To declare a class in PERL the general form is as follows:

package class-name

4.9.3 Methods

Methods in PERL can be referred as subroutines defined in a package. In PERL, the arrow operator (−>) is used to call a method. The general form to call a method is:

$object->method (@arguments).

Example 4.45 Sample method:

```
package emp;
sub printnames
{
        my ($self)=@_;
        printf("full-name:%s %s\n \n", $self->empfirstname, $self->emplastname);
 }
```

If suppose for any variable $var1 contains an emp(employee) object, this method print names can be called on that object by writing following statement: $var1- >printnames();

4.9.4 Constructors

For creating an object which is an instance of a class, it is necessary to have an object constructor. It is a method which has been defined in a package. The majority of the programmers will name this object constructor method as new, but whereas in PERL no rules exist to name this method. The constructor is that which constructs and returns a new object. So, that is why it is usually called new() . It is allowed to pass arguments to this constructor, which it can then use to do the initial setup of the object.

4.9.5 bless Function

The bless function tells the entity referenced by REF which is an object in the CLASSNAME package, if CLASSNAME is omitted. Usage of the two-argument form of bless is recommended. The general form of 'bless' function is:

bless REF, CLASS-NAME
bless REF

This function specifies the entity referenced by REF, and specifies an object in the CLASSNAME if the CLASSNAME is given. If the CLASSNAME is omitted it uses the current package.

Example 4.46 shows the constructor creation for sample employee class using PERL hash reference. While creating an object, a constructor is supplied that returns an object reference. The creation of object reference provides a reference to the class in the package.

Example 4.46 Constructor creation for sample employee class using PERL hash reference:

```
package emp;
#emp1.pm
use warnings;
use strict;
sub new
{
        my $self={};
        bless($self, "emp");
        return $self;
}
```

The emp(employee) class can be used to create an object:

```
 #/usr/bin/PERL
use warnings;
use strict;
use emp1;
my $emp=emp->new();
```

4.9.6 Inheritance

Inheritance allows reusability of code across various classes. The methods and attributes defined in one class can be used in another class through this property without rewriting the code again. The class from which the methods and attributes are inherited can be called a parent class and the class which inherits and uses those methods can be called a child class . PERL includes a special variable @ISA, which represents the relationship between classes. Classes which have just one item in their @ISA array comes under the category of single inheritance. Classes which have more than one element in @ISA indicate multiple inheritance.

Example 4.47 Usage of inheritance:

```
package dept1;
#dept.pm
use warnings;
use strict;
our @ISA=qw(emp);
```

4.9.7 Polymorphism

Polymorphism is a situation in which a single object exhibits many forms. Polymorphism is defined as the ability for objects to respond differently to the same message depending on their class. Polymorphism in object-oriented programming allows a single method to do different tasks depending on the class of the object that calls it. Generally, there are mainly two types of

polymorphism, namely, inheritance polymorphism and interface polymorphism. In inheritance polymorphism, all the objects share a common set of methods, whereas in interface polymorphism polymorphic behavior exists with objects that do not share anything in common. In this objects are not related, but they share some common interface which allows them to be treated in a polymorphic way.

4.9.8 Destructor

Destructor deallocates the memory allocated to the object of the class when one stops using it. PERL automatically destroys a variable when an object is no longer in use. There exists a method known as DESTROY which can be called on the object before de-allocating the memory allocated to the object by PERL. In general, a destructor method can be viewed as a subroutine named DESTROY called automatically under any of the four circumstances, such as, when the object reference variable is undefined, when the variables of object reference's goes out of scope, when the script terminates or when the PERL interpreter terminates.

Example 4.48 Usage of destructor:

```
package class1;
.
.
sub DESTROY
{
        print "class1: :DESTROY called now";
}
```

4.9.9 Method Overriding

If the implementation of the inherited method in the child class is modified, the process is known as method overriding. Using the SUPER constructor override allows you to call an overridden superclass method without actually knowing where that method is defined. The SUPER: construct is meaningful only within the class.

REVIEW QUESTIONS

1. Define parsing with examples using PERL.
2. Explain PERL parsing rules and advantages.
3. Define variable and explain about different variable types.
4. Write a program for swapping of two numbers using two variables.
5. Explain about local scope and global scope of variables with examples.
6. Explain about different variables (i.e., Integer variable, float variables, etc.) with syntax in PERL.
7. Explain about array operations in PERL (i.e., Adding elements, deleting elements).
8. Explain about statements and write a program to tell younger, middle and old depending on a age.
9. Write a PERL program by using different conditional statements.
10. What are the different types of iterative statements in PERL? Explain each of them with an example.
11. Write a PERL program to print Fibonacci series.
12. Explain about various object-oriented concepts in PERL.

13. Explain different object methods in PERL.
14. Explain different concepts of polymorphism in PERL.
15. Explain the concept of modularization in PERL.
16. Explain about modules and packages.
17. Explain about subroutines with suitable examples.
18. Explain different operations on files with an example.