Unit-2 Advanced PERL

Solutions

2.1 Finer Points of Looping

Q1. Discuss about finer points of looping.

Answer :

"continue" block

A while loop can have an optional continue block. This block is executed as long as the block is continued i.e., after each normal iteration and before the control goes to retest condition.

Syntax

```
while (condition)
        {
        Statement 1
        Statement 2
            ⋮     ⋮
        Statement n
        }
continue
        {
        Statement 1
        Statement 2
            ⋮     ⋮
        Statement n
        }
```

The 'while' loop with a continue block is equivalent to the 'for' loop. The third component in a 'for' loop is similar to the continue block.

Example

```
for ($a = 10; $a < 13; $a++)
        {
        print "Hello"
        }
```

The equivalent code of 'while' loop is given below

```
        $a = 10;
        while ($a < 13)
        {
        print "Hello"
        }
        continue
        {
        $a++
        }
```

Multiple Loop Variables

'for' loop can also be used to iterate over multiple loop variables. To iterate through more than one variable just separate the expression using comma operator.

```
for ($a = 2, $b = 3; $a < 10; $a++, $b+ = 1)
        {
        print "Hello"
        }
```

Consider, the scalar context, in which the variables are separated by comma. Firstly the left hand side operand is evaluated and its value is returned then right-hand side argument is evaluated. In the list context, the 'comma' operator is considered as 'list constructor'. In the example, above evaluation is in 'scalar context', which increments the values of $a and $b.

Last, Next and Redo Commands

For answer refer Unit-I, Q16, Topic: Last, Next and Redo commands.

Q2. Discuss subroutine prototypes.

Answer :

The number of arguments and their type in a subroutine is specified using subroutine prototypes.

```
sub demo ($);
sub box ( );
```

Here, 'demo' is a subroutine that can take a scalar variable as an argument. 'box' is another subroutine with no arguments.

In general, at the time of compilation the prototype is used to check the arguments and their types. But the actual usage of subroutine prototype is that, it can be used as built-in function when the arguments are not specified in brackets.

In the subroutine demo ($), if the '$' is not a scalar argument then it is converted into a scalar. Here, type checking does not takes place. At compile-time if a mismatch occurs when checking the number of scalar arguments then the compilation is aborted suppose, box is written as,

```
$ s = box –1;
```

we can omit the brackets around the arguments in a subroutine, if prototype is defined before subroutine is being called. The value returned by box is assigned to the variable $s. If the subroutine is defined without a prototype, the compiler parses the statement as,

```
$ s = box ( ) –1
```

For example, consider a function demo without any prototype

```
sub demo
        {
        return var;
        }
```

and if @a = (demo $S1, $S2, $S3);

Here, function "demo" is defined without any prototype. The value of @a contains a list of one item where $S1, $S2, $S3 are taken as argument but ignored by demo. Hence it contains value equal to $S1; $S2 and $S3.
Suppose, demo is defined with a prototype then the compiler takes only one argument and @a contains a list of three items i.e., demo ($S1), $S2 and $S3.

```
sub demobox ($$);
@a = (demobox $S1, $S2, $S3);
```

Here, the subroutine demo box is supplied more than one argument, which gives a compile time error. A prototype may contain optional argument as,

```
sub demobox ($ ; $)
```

This subroutine can takes either one or two arguments. If the arguments present in the subroutine are not scalars then they are forced to be scalars.

@, % or & are used to represent a prototype which can contain non-scalar arguments. '@' signifies a list, '%' ahash and '&' subroutine. If we write a prototype as:

```
sub demo box (@);
```

Then, it takes more number of arguments. An empty list is also considered as a list by the compiler. Hence, subroutine without any arguments is also valid.

```
sub demo box ($ @)
```

This subroutine takes two arguments. First one signifies a scalar and the second argument a list.

```
sub demo box (%)
```

This subroutine take only one argument i.e., hash.

Q3. What is the purpose of "caller" function?

Answer :

The 'caller' function returns the stack information of the current subroutine call. The syntax of 'caller' function is,

```
caller expression
```

The argument caller "expression" is a non-negative integer. This argument is evaluated as the member of stack frames to go back from the current one. For example, if expression = 0 then it is the current stack frame. If expression = 1 then it is the caller. If the expression = 2 then it is the caller's caller and so on. The following code generates a simple stack track,

```
$ x = 0;
while (@ stacktrace = caller ($x++))
        {
                print "$ stack trace [3]\n";
        }
```

Here, the caller function has an expression as argument. If the caller function is used without an expression, it returns the package name, file name and line number of the caller that called the currently executing subroutine. Example: ($package, $filename, $line number) = caller; When this is used outside the named package, it returns 'main' as package name.

Another example construct that returns additional information about the caller that called currently executing subroutine is as follows,

($package, $filename, $linenumber, $subname, $numofargs, $context) = caller 0;

2.2 Pack and Unpack

Q4. What is the purpose of pack and unpack functions? Explain how these functions can be use to simulate C structs.

Answer :

'pack' Function

The syntax of pack function is as follows,

```
pack templ ate, list
```

The purpose of this function is to convert the given list into a string of bytes according to the specified template. Finally, it returns the string.

'unpack' Function

The syntax of unpack function is as follows,

```
unpack tem plate, string
```

This function work opposite to 'pack' function. The purpose of this function is to expand the given string into a list of values according to the specified template. Finally, it returns the list of values.

Some of the template characters for pack and unpack are listed in table below,

Template Character	Description
c	Represents a (8-bit) signed character value
C	Represents a (8-bit) unsigned character value
d	Represents a double-precision float value
f	Represents a single-precision float value
i	Represents a signed integer value
I	Represents a unsigned integer value
l	Represents a (32-bit) signed long value

L	Represents a (32-bit) unsigned long value
s	Represents a signed short value
S	Represents a unsigned short value
x	Represents a null byte
N	Represents a 32-bit long in 'Network' order
V	Represents a 32-bit long in 'VAX' order

Table: Template Characters

Example: If @values is a list of five integer values then

pack "ccccc", @values;

returns a string containing five corresponding ASCII characters. Since @values is a list of length 5, it can also be written as

pack "C5", @values;

For a list of unknown length, it is written as follows,

$len = @listname;

pack "c$len", @listname;

In other words, we can say that pack function packs a list of values into a binary structure. Let us illustrate this with an example: Consider an application that deals with chunks of data that is made up of a type identifier, descriptive identifies and a payload. Suppose that the type identifier is a character. The descriptive identifier is a string of 32 characters. The payload is a long integer of 32 bits. In C language, this data item would have been represented as a structure with three fields which occupies 69 bytes. Whereas in perl, the three components are packed into 69-byte string as shown below:

@palyoad = (……); # 16 long integers

$x = pack "ca32N16", (chr(14), "Urgent message for 19 Oct", @payload);

The 'c' in the above template denotes character. The 'a32' denotes 32-character long ASCII string. The 'N16' denotes 16 long integers in network order i.e., big endian representation, wherein most significant byte would be first.

To reconstruct the list of values from the packet string, then unpack function is used.

For example: @list = unpack "ca32N16", $x;

Suppose if the list is sent to VAX machine, where the integers are stored in least significant byte first i.e., little endian order. The template in unpack function must be changed to ca32V16 so that the integers are unpacked in VAX order.

In another method, the pack function can be called as an argument of unpack.

For example,

$num = unpack "B32", (pack "N", $y);

Here the unpack function will convert the Perl integer to a 32 bit string. The result of the above unpack function is substituted by a regular expression shown below to strip off the leading zeroes.

$num = ~ x/^0+//;

2.3 File System

Q5. What are filehandles? Discuss the functions to deal with filehandles.

Answer :

File Handles

A filehandle is an abstract name given to a file, device, socket or a pipe. Filehandles help in getting input from and sending output to many different places easily.

Filehandles are created and attached to a file by using the open function. The open function should be provided with at least two parameters: the filehandle and the filename to be opened.

Example: open (FHNAME, "file1.c");

In this example, FHNAME is the filehandle associated to "File1.c".

The predefined filehandles used in Perl are STDIN, STDOUT and STDERR. The mode in which the filehandle has to be opened can be explicitly specified as shown below:

Open (FHNAME, "<filename"); #open in read mode

Open (FHNAME, ">filename"); #open in write mode

Open (FHNAME, ">>filename"); #open in append mode

Open (FHNAME, "+>filename"); #open in write mode first and then in read mode

Open (FHNAME, "+<filename"); #open in read mode first and then in write mode.

The two modes +> and +< needs special attention. These modes open file for reading and writing. When opened in "+>" mode, the filehandles are opened in write mode first, which causes overwriting any existing file with the same name. And, when opened in +< mode, the filehandles are opened in read mode first and hence keeps the existing file with same name.

The open function returns an "undef" on failure and silently discards the output. To terminate normally, "die" statement is used as shown below:

Open FHNAME, "filename" or die "cannot open";

(or)

Open (FHNAME, "filename") || die "cannot open";

To include error code in case of failure it is written as follows:

Open FHNAME, "filename or die "cannot open ($!)";

Here, the special variable $! holds the error code.

Note that the filename parameter in open function can be omitted by supplying the filename through the filehandle parameter. That is we firstly initialize a variable with the file name. Then supply it to open function as filehandle parameter.

Example:
```
$FHNAME = “d:/cprogs/prog1.c”;
Open FHNAME or die “Cannot open $FHNAME”;
```

We can perform read and write operation by using the filehandle and the angle operator. The input filehandle can be used in angle brackets and an output filehandle can be used after print or printf.

Example:
```
OPEN IFHNAME “$x” or die “cannot open $x”;
OPEN OFHNAME “$y” or die “cannot open $y”;
    while (< IFHNAME>)
        {
            print OFHNAME
        }
```

After completion, the opened filehandles must be closed using the “close” function.

Example:
```
close (IFHNAME);
close (OFHNAME);
```

Q6. Explain how to test the status of a file using suitable examples.

Answer :

In Unix, “if” statements allow users to test the status of the file before attempting to open it.

Example:

The following construct tests whether file named “prog” in /temp directory has write permission.

if –w “/temp/prog” …

Perl provides collection of unary file test operators that allow users to test the status of the file. These operators take filename or filehandle as arguments. The table below lists few useful operators that are common to both UNIX and Windows NT.

S. No.	Operator	Description
1.	-e	checks if file exists
2.	-r	checks if file posses read permission
3.	-w	checks if file posses write permission
4.	-x	checks if file posses execute permission.
5.	-s	checks if file has non-zero size
6.	-z	checks if file has zero size
7.	-f	checks if file is not a directory
8.	-d	checks if file is a directory
9.	-t	checks if file is a character device

Table: Unary File Test Operators

The return value of these operators is a Boolean value. Thus, they can be used with if statement and also other constructs.

For Example: -w “c:\prog” or die “prog file is not writable”.

Now, let us discuss about few special operators.

The operator –s which checks if the file has non-zero size, does not return true if successful instead it returns the actual size of the file.

Thus, this operator can be used to compare file sizes using the following construct,

– s $file 1 > – s $file2

If @filenames contains list of filenames this operator can be used to list the file sizes corresponding to each file name. The construct used is as follows,

@sizes = map {– s $_} @filenames

Where, @sizes is an array that stores the result i.e. the list of sizes corresponding to filenames.

❖ The operator –t is used to check if the standard input is connected to keyboard –t STDIN or die “STDIN not connected to keyboard”

❖ The operator –M and –A are used to return the age of the file in days since the last modification and last access respectively.

Q7. Write short notes on,

(a) Pipes

(b) Binary data.

Answer :

(a) Pipes

Pipes are used to establish interprocess communication. They provide unidirectional flow of data from one process to another process. Perl supports the concept of pipes through the pipe operator ‘|’ (vertical bar).

The open function in Perl is used to attach standard input or output of a command to a filehandle by appending or prepending pipe symbol (vertical bar) to its second parameter.

Case 1

If the pipe symbol is appended (i.e., placed at the ending) to its second argument then, it treats the rest of the argument as a command and is executed in new process.

Example:
```
open (WHOHANDLE, “who|”);
        @users = <who>;
```

In this example, the open function attaches the standard output of command to the filehandle “WHOHANDLE”. And the array @users stores the list of all the users who are logged-in.

Case 2

If the pipe symbol is prepended (i.e., placed at the starting) to its second argument then, the filehandle becomes input to the specified command (i.e., lpr).

Example: open (LPRHANDLE, "/lpr –P Callahan");

print LPRHANDLE @symbol;

In the example above, the filehandle LPRHANDLE acts as input to the UNIX command "lpr". This command runs as long as close (LPRHANDLE) is encountered.

(b) Binary Data

In Perl, a string byte is treated as 8 bits without bothering about the contents of the byte. This makes very convenient for the binary data to be represented as a string.

The build-in file handling functions such as carriage return (\r) and line feed (\n) have different meaning on different platforms. In Unix, a line ends with '\n'. In DOS and Windows, the line ends with '\r\n'. Whereas in Macintosh, it ends with '\r'. Perl adopts UNIX conventions. In addition, the input output system of Perl translates these conventions according to the platform it is running. The bin mode function can be used to stop this translation.

Example: Binmode TEMPFILE

Here, while processing a file with filehandle TEMPFILE, the bytes that contain code for \n or \r are not translated but are left as it is.

The binmode function must be used after open function and before accessing the file.

It is a good practice to use binmode to guarantee portability. The "print" function and "read" function are used to write and read the binary data to and from a file respectively.

Example

$bytesread = read TEMPFILE, $data, $num

Here, $num of bytes are read from the file specified by the filehandle TEMPFILE. These bytes are stored as a string in $data variable and finally the actual number of bytes read are returned, which are stored in $bytesread. It returns 0 in case it attempts to read at EOF. To check if read is successful we write (read TEMPFILE, $data, $num) = = $num or die "Read at EOF".

The read command can be used to specify the position where data has to be placed in the current (target) string. Suppose if we wanted to place data at the end of the current string $data. The optional offset parameter of read function is used. Here, the "offset" will be equal to length of the current string. It is written as follows,

read TEMPFILE, $data, $num, length $data;

After placing the binary data into the current string, the unpack function can be used to decompose it. The template used with unpack function will define the structure of data.

Example

Consider reading and decoding a UDP packet header of 8 bytes that contains source and destination port number, checksum and length of the packet. The file handle must be set up as follows,

read UDPPACKET, $headerinfo, 8;

($sport, $deport, $len, $checksum) = unpack "S3", $header info;

After knowing the length, the packet payload can be read as follows,

read $UDPPACKET, $info, $len;

Q8. Explain sysread and syswrite functions.

Answer :

'sysread' Function

The syntax of 'sysread' function is,

sysread Fhandle, scalar, len, offset

This function reads the block of data (len bytes) from the given Filehandle (Fhandle) into the variable scalar.

It returns number of bytes read on success, return '0' if EoF is reached and returns undef on failure. The size of the scalar variable increases and decreases depending on the length of data actually read.

The offset parameter is optional. It specifies from where the string has to read. By default, the string in read from the beginning of the File. An exception is raised when "len" is negative or when offset is out of string boundary.

'syswrite' Function

The syntax of 'syswrite' function is,

sysread Fhandle, scalar, len, offset

This function writes "len" bytes of data from scalar variable to the given Filehandle (Fhandle). It returns number of bytes written on success and returns undef on failure. The offset parameter is optional. It specifies the starting point for writing the string. Generally, offset is 0, when scalar is empty. If the offset is negative then writing starts that many bytes backward from the end of the string.

An exception is raised when "len" or when "offset" is out of string boundary.

Q9. Explain random access files.

Answer :

Files can be accessed randomly using 'seek' and 'tell' functions.

'seek' Function

The syntax of seek function is as follows,

seek Fhandle, offset, position

This function is used to position (or move) the file pointer explicitly.

The "Fhandle" parameter specifies the Filehandle of the file that has to be opened.

The "offset" parameter specifies the byte position.

The "position" parameter specifies the position that will be used as starting point by the "offset" parameters. The starting point of offset will be,

(a) When position = 0, it is beginning of the file. For example: seek TEMPFILE, $num, 0; This construct tells that the file pointer is set to an offset of $num from the beginning of the file.

(b) When position = 1, it is current position in the file. For example:- seek TEMPFILE, $num, 1;

This construct tells that the file pointer is set to an offset of $num (can be positive or negative) relative to current value of the file pointer.

(c) When position = 2, it is end of the file. For example: seek TEMPFILE, $num, 2;

This construct tells that the file pointer is set to an offset of $num (can be positive or negative) relative to end of file.

The return value of seek function is '1' if successful and 0 if it fails.

'tell' Function

The syntax of tell function is as follows,

tell Fhandle

This function is quite opposite to seek function. The "Fhandle" parameter is optional. If specified, it returns the current file pointer offset for the specified filehandle. Otherwise, it returns the file pointer offset of the file recently used.

Example: $pos = tell TEMPFILE;

Here, the tell function returns the current position (offset) of file pointer for TEMPFILE and then stores it in $pos.

Q10. Explain the following,
(a) Searching directories
(b) Filename globbing.

Answer :

(a) Searching Directories

Similar to the open function which provides filehandle to access an openfile, the "opendir" function provides handle on a directory.

The syntax of opendir function is as follows,

opendir Dhandle, Dname

This function opens the directory specified by "Dname". The processing is done using "readdir", "telldir", "seekdir", "rewinddir" and "closedir" functions.

Example

(i) In Unix, opendir (TEMPDIR, "/temp");

opens a directory named temp and stores its handle in variable TEMPDIR.

(ii) Similarly, in Windows, it is written as opendir (TEMPDIR, "d:\temp");

It is a good practice to track error messages and protect against failure. The following code does it:

Example $directory = "C:\temp";

opendir (OURDIR, $directory) or die "$directory cannot be opened";

After acquiring handle to the directory, the contents of the directory can be read using "readdir" function a shown below:

@files = readdir TEMPDIR;

Here, the "readdir" function returns a list of entries (files) in the directory and stores in @files.

Let us look at a simple example which shows processing of directory.

```
$directory = "d:\temp";
opendir (OURDIR, $directory) or die "$directory cannot be opened";
foreach $entry (readdir OURDIR)
    {
        ......# processing each entry
    }
closedir OURDIR
```

In the above example, the readdir returns the next entry in the list. This is because $entry is a scalar variable.

(b) Filename Globbing

The term globbing refers to 'pattern matching'. File name globbing means, the process of matching a pattern which contains wild-card characters against the file names in the current directory.

In Perl, file name globbing can be performed by using the angle operator with wild-card pattern

Example: @files = <*.c>;

It returns all the files in the current directory whose filename ends with ".c" and stores them in @files variable.

To avoid overloading of the angle operator, "glob" function can be used. For the example given above, the alternative way of globbing is using "glob" function as follows:

@files = glob ("*.c");

Now, let us see how the filename globbing works when used inside while loop.

```
while (<*.cpp>)
        {
            chmod 0400, $–;
        }
```

The above code changes the permissions of all the cpp files to 'read by owner only'. Each time <*.cpp> is called it returns the next file name and assigns it to $-. This finally results in, changed permission of all the cpp files.

Q11. What are typeglobs? Explain its usage for aliases and filehandles.

Answer : **Model Paper-I, Q2(a)**

Typeglobs

Perl provides a special type called "typeglob", which can represent any type of variable like hashes, array, scalar variable, etc. It is generic variable name, when evaluated it produces a scalar value that represents all the data structures assigned to same variable name. Typeglob variables are preceded by the character *. A typeglob variable can represent the following types of variables,

Type	Character
scalar	$
array	@
hash	%
subroutine	&

Example for a typeglob

For example, the typeglob variable * demo can represent $ demo (or) @ demo (or) %demo (or) & demo.

This typeglob variable can be used as any ordinary data type. We can perform the following operations,

1. Assign the value to a variable.
2. Store in an array.
3. Pass it as parameter to a subroutine.

We can even declare a typeglob variable as local in an subroutine by using local declaration as,

local * demo;

Aliases

An "alias" refers to another name given to a variable of any type. Typeglobs are very useful in creating aliases of symbol table entries. If a typeglob 'T' is assigned the value of another typeglob, then 'T' becomes an alias of that typeglob. That is both the typeglobs will now refer to the same symbol table entry.

Example *table = *demo;

The assignment statement above, makes "table", an alias of "demo". As we know typeglob *demo hold pointers to $demo, @demo, %demo etc. And hence its alias i.e., *table will also point to same entries. Moreover,

$table becomes an alias for $demo,

@table becomes an alias for @demo,

and so on.

This declaration will replace the copies of pointer in *table with "undef" and the typeglob assignment in the next line makes *table an alias for "demo".

However, at the end of the block, the *table will be restored with its original pointer. Aliasing typeglob can be also used for passing arrays as reference to subroutine. Consider that we have a subroutine "demosub" that takes an array as parameter. We express it as,

&demosub (@array);

This will copy all the contents of the carry into the argument of array @_.

For a large array, coping the whole contents of arrays to the argument is inefficient. So, a typeglob is passed as an argument to the subroutine.

```
sub demosub
        {
        local * dup = shift;
        # @dup contain all element of array,
        }
```

Now, if the subroutine can be called in the same way as before,

& demosub (*array);

By this, it makes dup as the alias to array.

As, dup is declared locally, it was not declared before with any other value, at the end of the block dup will be discarded. Perl 5 users the concept of references for passing arrays as parameters to subroutine. Special form of alias can be used by typeglobs like strings. When a string value is assigned to typeglob, then perl will creates a variable name same as the string.

FileHandles

Filehandles cannot be assigned as the values of variables nor they can be stored in any data structure. Moreover they cannot be passed as parameter to subroutine. Filehandles have there own namespaces. Suppose INFILE is a file-handle and we have another scalar variable with the same name i.e., typeglob * INFILE.

$INFILE = "/temp /demo";

open INFILE;

which has the same effect as,

open INFILE, "/temp /demo";

If the built in function open will be called by passing an argument as filehandle. And typeglob is used to find the file name set as a value to the scalar variable with the same name. Now a typeglob will be passed as argument to the subroutine

```
sub demo
{
local *FILE1 = shift;
while (<FILE1>)
{
}
}
```

and this function will be called as shown below,

```
open SOMEFILE, "/temp /some.txt" or
      die "file cannot be opened";
demo (* SOMEFILE);
```

Typeglob can also be used to redirect output,

```
open FILE "log.txt"
or die "cannot open logfile";
*STDOUT = *FILE;
print ("$logmessages \n");
```

In the line above, the print statement by default send the printing on STDOUT and redirection is done through typeglob assignment. Above lines can also be written as,

```
open FILE "log.text" or
die "file can't be opened";
print FILE "$logmessage in";
```

2.4 eval

Q12. What is the purpose of eval operator? Give example.

Answer :

'eval' operator is categorized into two forms,

(i) eval with block

(ii) eval with expression.

(i) eval with Block

This form of eval take its arguments as a block. This block will be compiled only once. If an error occurs at the runtime, eval returns "undef" and puts it in $@.

This structure is same as try and catch blocks in C++.

Here, instead of try and catch we use "eval" and "if ($@)" condiction.

Example

```
eval

{

# code that may contain errors

}

if ($@ '   ')

{ # code that is to be executed

  # in case of error

}
```

It is more efficient because syntax checking is done at compile time.

eval BLOCK is itself a loop, so next, last, loop controls are not used to leave and restart the block.

And, BLOCK should contain the valid perl code that is validated before the compile phase.

(ii) eval with Expression

This form of eval takes an arbitrary string as operand and executes it at runtime.

This form runs very slower than the other because it has to parse the string all the time. The value of the expression that is evaluated last, is the return value of eval. However we can return in the middle of an eval using return operator. The return value provided by the expression is evaluated as void, scalar or list context. In case of syntax errors eval returns "Undef".

Expr does not get validated until the runtme. So, it does not show any error until the runtime.

Examples

```
$cmd = '...';

----

----

$val = eval "\$$cmd";
```

2.5 Data Structures

Q13. What is meant by referencing and dereferencing? Explain about references to arrays and hashes.

Answer :

Referencing

Reference are nothing but pointer to another object. They allow indirect access at data objects.

Like pointers in 'C' language, perl uses references.

There are two types of references,

(i) Symbolic

(ii) Hard.

(i) Symbolic Reference

A symbolic reference is a variable that contains name of another variable. In other words, the value present in a symbolic reference variable is a string which is the name of another variable.

Example

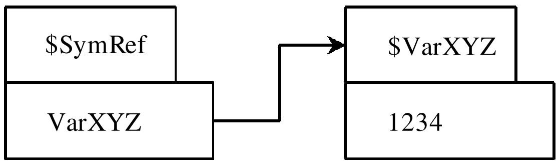

Figure: Symbolic Reference Variable

In the example above $SymRef is a symbolic reference to the variable $VarXYZ. Note that the contents of $SymRef are nothing but the name of the variable which it is referring to (i.e., VarXYZ).

A symbolic references can be created by just assigning name of the variable as follows,

$SymRef = "VarXYZ"

(ii) Hard Reference

A hard reference does not store the name (string) of another variable, instead it locates the actual value in memory to which the reference is pointing.

Example

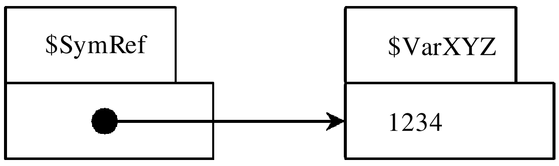

Figure: Hard Reference

The example above shows that $SymRef is a hard reference variable that points to the location in the memory where $VarXYZ is present. A hard reference is created by a backslash operator as follows,

$SymRef = \$VarXYZ

Example

$ref = \$scalar;//scalar variable

$ref = \@list;//list

$ref = \%hash;//hashes

$ref = *FILE;//Files

Dereferencing

Dereferencing is a reference variable means extracting the value to which it is pointing to. If the reference of a scalar variable is given as,

$ref2demo = \$demo;

Then its dereference is

$$ref2demo;

This will result the same value as $demo would have returned. For, other data structures the dereference are given as.

If $ref = \@list is a reference, then @ {$ref} will be the dereference.

If $ref = \%hash is a reference, then % {$ref} will be the dereference.

If $ref = *FILE is a reference, then {$ref} or scalar <$ref> will be the deference.

References to Arrays and Hashes

There are two main contexts where references to arrays and hashes is used.

(i) While constructing data structures

(ii) While passing arrays or hashes as arguments to subroutine.

Anonymous Arrays

Creating reference to the anonymous array wll result in a complex structure. But reduces the time taken for the program to run.

Let us consider a anonymous array of colors.

$color = ["Red", "Yellow", "Black"]

This assigns the array of variable that are represented within a square bracket to the scalar variable $colour.

Here, right side of the assignment we have the array and lift side of the assignment will act as the reference to the array.

We can also use the nested arrays like.

@nestarray = ["Grey","Pink",["Red", "Yellow", "Black"]];

This representation results in some more complex data structure.

Same representation can be used in creating array of arrays.

$nestarray = ["Grey","Pink",["Red", "Yellow", "Black"]]

$nestarray is the reference to an anonymous arrays were the third argument is a reference to another anonymous array.

Anonymous Hashes

Like Anonymous arrays, anonymous Hashes are easy to create, but we use braces instead of square brackets.

$color = {"Blue"= "light",
"Black"= "Dark",
"Pink"= "Bright"};

An element uses a string literal, an expression, or an variable to create the structure.

It is not necessary to create the anonymous hashes using braces, but to define the elements to a confined group. We use them.

Q14. Explain two-dimensional arrays in perl.

Answer :

Two Dimensonal Array

A two dimensional array is also known as array of arrays. Unlike other programming language, Perl does not support the syntax.

variable name[dimensional size][dimensional size]

Since, it is impossible to create lists of lists. So, Perl creates array of references to other arrays that show similar effects as array of arrays in other programming languages.

Creation

Let us create a two dimensional array of colors with 2,2 as its dimension.

```
@color = ([“Red”,“Yellow”,“Light Green”],
          [“Green”, “Brown”, “Black”]
          [“Blue”, “Pink”, “Dark Green”]);
```

If a list is assigned, then the whole lists should be enclosed within the parenthesis but not using brackets.

Otherwise, if the array of reference is created, then we use brackets as shown below,

```
$ref_color = ([“Red”,“Yellow”,“Light Green”],
              [“Green”, “Brown”, “Black”]
              [“Blue”, “Pink”, “Dark Green”]);
```

Accessing an Array

In order to access an element from an array @ color and print them. We represent it as print $ color [1][2] which is equivalent to $color[1] –> [2]. This will print “BLACK”. To access the element from the array of reference $ref_color and then print.

print $ref_color –> [1][2] which is equivalent to $ref_color –> [1] –> [2]. This will print Black. Deference requires an implide –> symbol between the brackets][.

Note: Negative dimensional value are also accessable in perl, they count backward from the end of an array.

Example: $color[0][–2]

This will result in Red.

Now, let us consider another example.

$color = (‘Bright’,‘Dark’,‘Light’);

In order to replace a value present in array, we use eval operator.

$color = $color[0];

Q15. Explain about creation of complex data structure.

Answer :

By keeping the array of arrays concept in mind, Perl grasps an opportunity to form a hash of hashes concepts. This provides the capability of records. In the same way array of hashes and hashes of arrays can be created. By putting all the concept together we can form a complex data structure.

Example

Let us illustrate the implementation of doubly linked list. One way to represent it is to use an array, where every element, contains the hash with three fields (i) ‘*l*’ (left neighbour) (ii) ‘r’ (Right Neighbour) and (iii) ‘c’ (content).

L, and R will be assigned with the reference to the hash elements. And ‘C’ can be assigned with any of the values like scalar, array, hash or reference. Hashes are referenced by two scalar variables.

(i) $head: It will point (or refer) first element.

(ii) $present: It will point or refer to the element of current interest $present –> {‘C’}.

We can traverse in the forward direction as:

$present = $present –>{‘R’}

and in backward direction as:

$present = $present –>{‘L’}

If a new element is created including 3 feilds L,C,R

$new = {L ⇒ undef, R ⇒ undef, C ⇒ .....};

To insert the element after the current element these step are to be followed.

```
$new –>{‘R’} = $ present –> {‘R’};
$present {‘R’} –> {‘L’}; = $new;
$present {‘R’} = $new;
$new –> {‘L’} = $present;
```

If the current element is to be deleted, then

```
$present –> {‘L’} –> {‘R’} = $present –> {‘R’};
$present –> {‘R’} –> {‘L’} = $present –> {‘L’};
```

2.6 Packages, Modules

Q16. Explain about packages.

Answer :

A package is a collection of code that is identified by its own namespace. To define a new package the following declaration statement is required,

package Packagename;

Initially, “main” will be the default package to run the code. Variables declared in the package are global used anywhere in (i.e., they can be the package). And the variable specified in one package are invisible to another package until the ‘complete qualified name’ is given to access those variables i.e., $ package name:: variable name

Example

If ‘m’ is a variable of package P1, then it is denoted as $P1::m and package P2 also has a variable ‘m’ it is denoted as $p2::m and if main package has a variable ‘m’, it is denoted as simply $m variable in the Main package should be represented as $_. These variable are visible to all other package without giving any qualfied name.

After a package declaration, the code written below it will belong to it until another package is declared. When another package is encountered the original package will be temporarily invisible.

Example

```
package P1
        $a = 9;
package P2
        $a = 10;
package P1
print $a;
```

This will result in 9.

In this example, P2 package will switch onto different symbol table, but when package P1 declaration appears again it will return to the original symbol table that has $a as 9. If package P1 and P2 has no connections (i.e package P1::P2) then it will refer the variables present outside the package which must be in a fully qualified form. Like

Example: $P1::P2::a

One or more BEGIN routines can be processed in a single package at compile time.

Q17. Write short notes on,

(a) Libraries

(b) Modules.

Answer :

(a) Libraries

A library is a package that contains a set of related subroutines that can be used for a particular purpose. A library is stored in a file with extension.pl and it is reusable. For example, a Math.pl is a library that stores all mathematical function. A library (e.g., Math.pl) can be loaded in a program by writing the following at the head of the program.

```
require "Math.pl";
```

The subroutines present in the library can be accessed using fully qualified names, as shown below.

```
$res = Math:: sqrt ($ num);
```

(b) Modules

Like a library, a module is also a package but it provides enhanced functionality. A module is stored in a file whose name is same as the package name with the extension.pm. Modules are more reusable than libraries because they follow certain conventions and take advantage of built-in support. When a program loads a module, the subroutines in the module are automatically imported into the namespace of that program. For example, a module Math.pm is loaded at the starting of the program by writing use Math;

Now the program can access the subroutines present in the module as if they were defined in the program itself. The above statement has the effect of writing the following.

```
BEGIN
{
        require "Math.pm";
        Math: : import ();
}
```

A user can also import only selected subroutines of a module. However this selection must be from the export list of the module. For example, the following statement imports only sqrt, sin and cos subroutine of Math.pm module.

```
use Math('sqrt', 'sin', 'cos');
```

The above statement has the effect of writing the following,

```
BEGIN {
        Require "Math.pm";
        Math:: import('sqrt', 'sin', 'cos');
```

When the user requests a module using "use" statement Perl searches the current directory. If module is not available then it searches the array @INC. This array holds a list of directories for a specific platform.

Q18. Explain about importing the module names.

Answer :

A module is imported using the import() method defined in the module. This method can be defined in many ways. Mostly modules inherit it from the exporter module. For example, the module Math.pm is written using exporter module is as follows:

```
package Math;
require Exporter;
@ ISA = 'Exporter';
@EXPORT = qw (----);
@EXPORT _ Ok = qw (----);
```

In the Math.pm module the require Exporter; statement loads the Exporter package. If there is request for any subroutine that is not declared in the package then Perl searches the module specified in the @ ISA array. Since this array specifies 'Exporter' this package is looked in the names of subroutines specified in the @EXPORT array are exported by default whereas the names specified in the @EXPORT_OK array are exported only on the request.

When a program includes the statement use Math;

As the first statement, the import () method invokes the built-in function caller to obtain the package name that has called it. Since this statement is used outside a named package its caller is main then the import () method sets up a selective alias in the caller's symbol table for each subroutine listed in the @EXPORT array.

2.7 Objects

Q19. Write short notes on objects.

Answer :

Objects in Perl have different meanings from the objects in object-oriented programming. In Perl, an object is an anonymous data structure to which access is given by a reference. A reference to an object is created by a constructor subroutine called new.

When a built-in function bless is called in a constructor then an object is said to be blessed into a class. A class is a package that provides methods for the objects that belong to this class. A method is a subroutine whose first argument is a reference to an object.

For example, consider the following,

```
package Person;
Sub new {
        My $reference = { };
        Bless reference;
        Return reference;
}
```

Here Person is a package and { } returns a reference to an anonymous hash. The subroutine new returns a reference to this empty hash (an object). This object knows that it belongs to the package Person.

An instance of the object is created as follows,

```
        $stud = new Person;
```

Q20. Explain the following,
(a) Class methods and attributes
(b) Method invocation.

Answer :

(a) Class Methods and Attributes

The instance of the object class is called by the methods in that class. There exist few operations which are applicable to the class, but not to the object. Such operations are called class methods.

In the same way, the attributes that are common to all instances of the class are class attributes.

The Global variables of a package are considered as class attributes. The class methods are subroutines whose first argument is not an object reference like that of the instance method.

Perl provides polymorphism for class methods and instance methods through late binding.

The code below in the definition of the new constructor for the fruit class

Example

Code:

```
package Fruit;
sub new
{
My $ pack = shift;
My $ obj = { };
Bless $ obj , $ package;
Return $ obj;
}
```

When the package name is found by the dereferencing operator as its left argument as shown below,

```
$apple = fruit –> new ();
```

It searches for the subroutine, new () in the specified package and then passes the package name as first argument of the subroutine call.

(b) Method Invocation

Consider a 'Fruit' class having a fruit object.

The data of the fruit object is based on three properties i.e., type, color, name. There is a package that contains instance methods to set or to return a particular fruit. The first argument of an instance method is always an object reference. The target for the operation is identifying the particular instance of the class.

Let us see few instance methods:

```
1.  Sub get _ color
    {
    My $c = shift;
        Return $c–> {'Color'};
    }
2.  Sub set _ color
    {
    My $c = shift;
    $c –> {'Color'} = shift;
    }
```

The above instance methods are invoked as follows,

```
1.  $Mango –> set _ color 'yellow';
2.  $Mango _ is = $ Mango –> get _ color;
```

The first call is resolved dynamically to

```
        set _ color ($Mango, 'Yellow');
```

This is possible when the left argument of (–> arrow operator is used with a 'blessed' reference. Hence, the right argument becomes a call to subroutine in the package which in turn is called with the object reference inserted before the arguments supplied. It is then resolved to

Fruit:: set _ color ($Mango, 'Yellow');

Via the tag attached to $Mango by the bless operation in the constructor. This dynamic resolution of the methods provides polymorphism. The object other than mango can have their own set_color and get_color methods.

The method can also be invoked in the 'indirect object' form. This is considered as an alternative syntax for method invocation. For example,

set _ color $Mango 'yellow';

$Mango_is = get_color $Mango;

Here, in the first call there is no comma after the object reference. Let us compare the syntax of print statement with explicit file handle.

print STDERR "Error"

The above two forms just vary in syntax. They are provided so that there is a choice to choose the easier way to invoke a method.

Q21. Explain how inheritance is implemented in Perl.

Answer :

In Perl, only methods are to be inherited. There is no special syntax to represent inheritance of one class by another. Inheritance is accomplished by including a special array @ ISA in the package that defines the derived class.

Single Inheritance

The special array @ ISA contains only one element that represents the name of the base class. Each element in @ ISA in the name of another package. If a method is missing, then all the packages in @ ISA are searched recursively.

Example: Consider an Amphibian class as a subclass of a Animal class. The array @ ISA is declared with 'our' and not 'my' as it is a package variable and not a lexical variable as shown below,

package Amphibian

Our @ ISA = "Animal";

It is possible to access the Amphibian class wherever the Animal class is used directly or through a object. If Amphibian class object is declared as $ frog, then invoke a method called move on it as follows:

$frog –> move (12)

Since, frog is an instance of an amphibian, Perl will first search the method move in 'Amphibian' class as shown below,

Amphibian:: move

If the method does not belong to Amphibian class then it searches @ISA instead of raising a runtime exception. That is, @ Amphibian:: ISA array is searched directly for the method Animal :: move in Animal package.

If the move is not present in Animal package, then Animal will provide its own @ISA array. Now, a search will be made to its ancestral package where move method can be found. Single inheritance is same as linked list of related packages.

Multiple Inheritance

A special array @ ISA in a package contains more than one package as its elements. And, each package has more than one immediate parent. There are six different ways to find a subroutine,

1. Perl looks into its own package named class name:: method name. If this fails, then the next step will be performed.
2. Perl will search the inherited subroutine in all the parent packages listed in @ class name:: ISA for a parent : : method name subroutine. It is recursively searched from left to right and depth-first. In other words, grandparent-classes, great grand parent's classes, great great-grand parents etc are recursively searched. If this fails, proceed to step 3
3. Perl searches for the subroutine in UNIVERSAL:: method name.
4. Now, Perl leaves the method name and starts searching for an AUTOLOAD. It looks for a subroutine named as follows. Class name :: AUTOLOAD. If this fails, proceeds to step 5.
5. Perl now looks for the subroutine parent:: AUTOLOAD in all parent packages listed in @ class name :: ISA. The search is done recursively from left to right and depth – first.
6. Finally, Perl searches in UNIVERSAL:: AUTOLOAD for a subroutine.

The subroutine will be invoked after the search is successful. If subroutine is not found, an exception will be raised such as can't locate object method "Method name", via package "class name".

2.8 Interfacing to the Operation System

Q22. Describe about the operating system interface that is common to UNIX and windows NT.

Answer : **Model Paper-II, Q2(a)**

In order to use Perl as an alternative to write shell scripts, the UNIX implementation of Perl duplicated (replicated) the system calls as built-in functions.

However, the duplication of system calls depends upon the capabilities of the host operating system. The equivalents for most of the UNIX facilities are provided by the Windows NT. The facilities that are common to the UNIX and NT implementations of Perl are given below,

1. Environment Variables

A special hash % ENV holds the value of current environment variables. It is accessible by the script for reading or modifying. The local operator can be used to perform temporary change in the values of individual environmental variables.

Example

```
{Local $ENV {"PATH"} = "new value";
}
```

The commands that are inside in this block will be executed with new variable PATH. However, the value of PATH will be replaced by the original value when it exits from the block.

2. File System Calls

There are many file system calls that are used for handling the file system. These return a value that indicates the success or failure.

Example, the two equivalent forms of a file system call are given below,

```
chdir $d or die "cannot chdir to $×\n";
chdir $d\\ die "cannot chdir";
```

Even though the ultra-low precedence "or" operator is replaced with logical // operator, the unary operator chdir have the higher precedence than the logical operators.

Examples of some popular file system calls are,

chdir $d Change the current working directory.

unlink $d Similar to "run" command in UNIX or "delete" command in NT

rename ($a,$b)	In UNIX, it is "mv" command.
Link ($a, $b)	In UNIX, command it is "ln" command. NT doesn't support this command.
symlink ($a, $b)	In UNIX, it is same as ln-s. Not supported by NT
mkdir ($a, 0755)	Creates directory with specified modes
rm dir $a	Deletes directory
chmod (0644, $a)	Assign file permissions

3. Shell Commands

Shell commands in Perl Can be executed either by using the built-in system function or by quoted execution.

(i) Built-in System Function

Consider an example of the system function

```
System ("date");
```

In the given argument, the string "date" is treated as a shell command. In UNIX, this command is executed by \bin\sh and in Windows NT that can be by \bin\sh on UNIX it is executed by "cmd. exe". and by cmd.exe on NT. The existing STDIN, STDOUT and STDERR can be used for execution. These can be redirected by using the Bourne shell notation.

Example

System ("dates date file") & & die "cannot redirect";

Since, the shell commands returns zero to indicate success and non-zero value to indicate failure, & & is used as an connector instead of logical operator (//). When a list of arguments are given to a system function, the first argument is treated as the command that have to be run. In, other words, the argument might be a scalar value containing the name of the program and its arguments as blank separated words. In such cases the shell meta characters that are present are processed before passing the commands to the shell

Example

System ("echo c.pp files: *.cpp");

In the above function, "*.cpp" denotes a list of files in the current directory with "*.cpp" extension. It will be expended before the execution of echo.

Quoted Execution

Quoted execution can be used to capture the output of a shell command.

Example

$date= '/date/';

The date command output is assigned to $date variable. The result is replaced by the back quoted string. Because, it is a quoting, the variable can be substituted by the analog with q, qq and qw operators.

Example

$date = qx /date/;

Where qx is the quoted execution,

4. **Exec**

The "exec" function stops the execution of current script and executes the program that is specified as its argument. It is to be noted that exec () function never returns, hence it is unnecessary to have any other Perl statements following the exec function except "or die" clause.

Example

Exec "sort $sorted output" or die "exec sort failed /n";

A script can be replaced by the new script by using the "exec"

exec "Perl-w demo. p/" or die "cannot exec/n";

The argument processing is same as in case of system function

Q23. Discuss in detail about process control in UNIX.

Answer :

The process control mechanism in Perl is similar to classic UNIX. The child process is created by invoking the fork () function. This process runs in parallel with the parent process. The distinction between a child process and a parent process depends on the value returned by them. The child process always returns zero, whereas the parent process return the process identifier of its child.

The return value of the function fork () is used to control the if-then-else statement. One block of statement is executed in the child process and another block in parent process. Using fork () the child process can create its own children. Repeated calls to a fork () function by a parent process creates desired number of child processes.

It is not compulsory to follow the UNIX mechanism all the time, in which a child process always tries use 'exec' command to start running different programs in similar process. This can result in two different programs running in different processes.

In order to avoid the confusion between the two different processes that handle the same file. The child process and the parent process closes the unwanted file handles (E.g. pipes). Since, a child process can inherit all the open file handles, it is possible to close file handles which are not in use, before it starts anything.

Consider an example, in which a child process sends the data to its parent process using a pipe.

```
FRK:{
        Pipe (READHANDLE, WRITEHANDLE);
        If ($Pid = fork)
         {
                Cclose WRITEHANDLE; # Parent process
        }
        else if ($ pid = =0)
        {
        close READHANDLE;
        #child process
        exit;
        }
        else if ($ ! = ~ / no processes/)
        {
        sleep 4
        redo FORK
        else
        {
        Die "for fails \n"
        }
    }
```

READHANDLE and WRITEHANDLE are the two file handles returned by the pipe. In the parent process, the file handle 'WRITEHANDLE' is closed since this handle in example parent is supposed to only reads the data from pipe. Similarly in the child process 'READHANDLE' is closed, since a child process is supposed to only write the data to the pipe. By this, when a child process closes, the pipe indicates EOF (end of file) to the parent process observe the explicit 'exit' in child process, this 'exit' indicates the termination of child process, generating 'CHLD' signal which the parent process can trap when it requires to know about child termination.

Consider the case when the fork function is unable to create a new process. The reason for this situation is that, the limit imposed by operating system is reached or the system's process table is full. To overcome this situation, the fork function is invoked after few seconds. Which is allowed by the dynamic nature of Unix, in process creation. The 'redo' command used in the else if 'block' call the 'fork' function again after 4 seconds by seeing the error manage "No Process'.

If the fork function is still unable to create a new process then it returns '1' which is caught by the else block, in which "die" is invoked

Consider the scenario in which a parent process waits until its child process terminates. The 'wait' function returns 'pid' (process identifier) of the child process, when' the child process is terminated or returns '– 1' when there are no child process is.

The 'wait pid' function takes 'pid' as an argument and return 'true' when the child process terminates or '– 1' when there are no child processes. In the above two cases the exit status is returned using a special variable $?in the child process.

Q24. Discuss in detail about process control in Windows NT.

Answer :

Win 32 :: Process module takes the responsibility of the process control in Windows NT. The code below is used to call the constructor.

```
Win32 :: Process :: Create ($child,
"D:\\winnt \\ system32\\mspaint.exe",
"mspaint temptxt", O, NORMAL_PRIORITY_CLASS,
".") or
die "process cannot be created "\n";
```

By invoking the constructor a new process can be created. The variable $child is used to store the object reference created by the constructor. To run the process, the path "D:\\winnt\\system32\\notepad.exe", "notepadtemp.txt", is followed. This path provides the executable and shell command that a process needs. A '1' is used as fourth argument indirecate, that the filehandles opened by parent process are inherited by a new process. "O" indicates that filehandles are not inhereted. The priority of the childprocess is defined by a constant NORMAL_PRIOTIRY_CLASS, the current directory for child process is denoted by dot "." Operator.

Under the control of parentprocess, the childprocess runs in parallel with it.

Example

```
$child → resume ();
$child → suspend ();
```

'wait function' is used by the parent process, to wait until the child process terminates.

```
$child → wait (INFINITE);
```

It is always better to mention the waiting time

```
$child → wait ($timeout);
```

The child process is forcibly terminated when the waiting time is completed. The child process can be forcibly killed by using the command!

```
$child → kill ($exit_code);
```

Q25. Explain OLE automation server.

Answer : Model Paper-II, Q2(b)

OLE Automation server provides an interface through which other applications can control it. It is basically a windows application. The interface consist of a set of objects that have related properties (instance variables) and methods. A perl script can control an automation server by using win32 :: OLE module. All the components of Microsoft office are implemented as automation servers to make it possible for the VBA programs to control them. Traditionally VBA (Visual Basic Application) was used as primary macro language for the office suite. However, Perl can also be used as an alternative macro language with the help of win32 :: OLE.

An automation server instance is initiated by the constructor and reference is returned using which the properties and methods of the objects that are implemented by the server can be accessed.

Example

```
i = Win32 :: OLE –> new  ("Excel.Application");
```

In the above statement, a new instance of Excel is started and the methods of "Excel.Application" object are also invoke in the similar way.

```
k = i.checkspelling ($word);
```

As an alternative below statement returns a reference to a currently running version of excel if it exists, or else returns undef.

```
i = Win32 :: OLE –> GetActiveObject ("Excel.Application");
```

The another form of starting up Excel can be with a worksheet preloaded.

```
i = Win32 :: OLE –> GetObject ('c:\\sheets\\marks.xls');
```

The automation object properties appear as hash to Perl.

Example

Excel's default path can be known with

```
$filepath = i –> {'Defaultfilepath'};
```

However, we cannot set a new value by just writing into the hash. To do, so we have to use the method LetProperty () as shown below,

```
i –> LetProperty ('DefaultFilePath', $newpath);
```

Here, the LetProperty performs a 'by value' assignment Win32 :: OLE explores itself when it is used with Microsoft's 'Active data objects' to access remote databases.

2.9 Creating Internet Ware Applications

Q26. Discuss in detail about creating internet-aware applications.

Answer :

Internet is a huge repository of information residing on FTP servers, Web servers, POP/IMAP mail servers, News servers, etc. Information on Web servers and FTP servers can be accessed by a web browser and mail and News servers can be accessed by specialists clients. However, there is another way of accessing the information i.e through 'Internet-aware' application which an access the server and get the information without any manual involvement.

Example

A website providing a 'lookup facility' where a user defines a query by filling a form and clicking the 'Submit' button. The data in that filled form is forwarded to a CGI program residing on the server. The server then gets this information formats it into a web page and returns this to the browser. A perl application can connect to the server to send the request in the browser understandable format, take the returned HTML and retrieve the fields which is the answer to the query. Likewise, a perl application can also connect to a POP3 Mail Server to forward a request by which the server returns a message which lists the number of available unread messages.

Perl provides a specialized module, called LWP (Library of www access in perl), for creating internet aware applications. The use of this specialized module reduces the coding effort to a greater extent.

Accessing Web Using LWP::Simple

Web servers are given some simple interfaces by LWP::Simple module. Also the contents of a web page can be retrieved in one statement as follows,

```
use LWP::Simple
$urlname = "http://www.anysite.com/index.html";
$page = get($urlname);
```

This method does not make use of objects instead it makes a procedural interface for web access.

The task of "get" function is to extract the contents of the page and store it as a character string. This function returns undef in case of error during access. However usage of this function is not helpful as many error conditions can possibily occur such as page may no longer exist or the host may be down or busy. So, an alternative function with error checking can be used,

```
$extract = getstore($urlname, "page.html");
```

Extracts the page and preserves its contents in page.html file.

Similarly,

```
$extract = get_print($urlname);
```

forwards the content of the page to STDOUT. In $extract, the returned response is the original code returned by http server.

Example

if success then 200 or

if page not found then 404 etc.

LWP

Simple provides two functions is_success( ) and is_error( ) takes this returned code as an argument and providevs with a Boolean value. In order to test few specific outcomes, constants defined is HTTP::Status package (imported by LWP::Simple) can be used.

Example

```
use LWP::Simple
$urlname = "http://www.anysite.com/index.html";
$extract = get store ($urlname, "page.html");
die "page not found" if $ extract = = RC_NOT_FOUND;
```

These constants are the functions defined in HTTP::Status having definitions like

```
sub EXTRACT_NOT_FOUND( ) {404}
```

Moreover modification of a page is verified by mirror( ) method:

```
use LWP::Simple
$urlname = "http://www.anysite.com/index.html";
$extract = mirror ($urlname, "page.html");
```

After this there are two possibilities,

1. If the page is not modified then this method will set $extract to RC_NOT_MODIFIED.
2. If the page is modified then this method will set $extract to RC_OK.

And the new version of the page is stored in the file page.html.

LWP: Simple has another important function called Lead( ) which returns an array containing content type, document length, modification time, expiry time and server name. Therefore the content type of a given page can be extracted as,

```
print "Content type is", (head($urlname))[0], "\n";
```

A Detailed Example

Consider an example of a database where each record contains a name and telephone number.

Eaxmple: Baig, Tabrez. M. 9984262626

After entering "Baig" as the search name and clicking on the "Submit" button or GET request with the query data appended to the URL as follows sent by the browser.

http://www.oxford.com/cgi-bin/lookup?Baig

This can be taken up by the perl script to take out the relative lines from the HTML page which are returned

Script

```
Use LWP::Simple
#if the required name is in $name
$url = "http://www.oxford.com/cgi-bin/lookup? "$name";
$extract = get store ($url, "c:\temp\page.html");
die " page not found"
if $extract = = RC_NOT_FOUND;
open HTML, "<c:\temp\page.html";
while <HTML>{
print if /dddddddddd/
}
```

URL is constructed along with the query and get store( ) method is used to read the page into a file. The lines having ten digit string like that of a telephone number is printed.

Accessing Web through LWP::UserAgent

LWP::UserAgent is another module for developing internet aware applications. It is a powerful model that overcomes many of the limitations of LWP::Simple module. Consider that we are retrieving a page as follows,

```
Use LWP::simple
$url = "http://www.anysite".com/index.html;
$extract = getpreserve($url, "page.html");
```

If the required page is removed and the $extract is set to EXTRACT_MOVED_PERMANENTLY or EXTRACT_MOVED_TEMPORARILY, Then LWP::UserAgent can redirect our request automatically to a new location.

An object approach is used by this module thereby implementing a user-agent object which uses HTTP::Request and HTTP::Response classes. Hence, a page is retrieved in the following manner,

```
Use LWP::UserAgent
$agent = LWP::UserAgent –> new( )
$url = "http://www.anysite.com/index.html";
$request = HTTP::Request –> new("GET", $url);
$response = $agent –> request($request);
⋮
```

An Agent object has been created and a request object has been fixed for that. A Request method of the Agent object is used to do the work.

Advantages of LWP::UserAgent over LWP::Simple

(i) An automatic redirection is performed (if it is needed) by the request method

(ii) Pages can be requested using 'POST' method. Recall that LWP::Simple allows us to use only GET method.

Following the five commonly used methods of the response object:

(i) content( ): original contents of the page are retrieved

(ii) code( ) : HTTP server returned code.

(iii) message( ) : textual readable message relative to the code.

Example: "OK" for 200 or "NOT FOUND" for 404

(iv) is_success( ) : Boolean denoting success

(v) is_error( ) : Boolean denoting error.

Requests( ) is polymorphic method, where we can specify a second scalar argument (i.e., a filename/path). The content will be preserved in this file, similar to that of getstore( ) of LWP::Simple module

$response = $agent –> request ($request, 'page.html');

Another form is available which allows to process the data before the entire page has been loaded:

$response = $agent –> request ($request, \&process, $blocklength);

Everytime when the block defined by the third argument is read, the sub-routine referenced by the second argument is called for processing it. This can aid when large pages have to be loaded and when the connection is slow.

2.10 Dirty Hands Internet Programming

Q27. Define socket, server and server sockets. Also explain how to create and use sockets.

Answer :

Sockets

Sockets refer to network communication channels which allows two-way communication between different processes running on different machines. In Unix, socket became the default mechanism for network communication. In windows, similar functionality is provided by WinSock package.

According to Perl programmer's, a network socket can be seen like an open file which is known by a filehandle where you write with "print" and read through "angle operates. The socket interface is dependent on the TCP/IP protocol system, due to which the routing information is managed automatically. Moreover, a reliable channel is provided by TCP along with automatic recovery from data loss, due to this a TCP connection is also called as a virtual circuit. The socket implementations in Perl is similar sockets provided by Unix. Sockets also allow connections using UDP.

Servers

Generally, a network system has several client systems looking for services from servers which are nothing but long process in running state which manages resource grants.

Examples

HTTP servers grant access to web servers, FTP servers allows files to be retrieved using File Transfer Protocol (FTP) and POP/IMAP servers grant access to electronic mails.

Every socket has its relative port number assigned dynamically by the software system which listens for the connections only on some special ports. These ports are defined by the client while socket creation. In general, to communicate with a web server a connection on port 80 is established.

Server Sockets

A server socket has to serve multiple connections. A well-known port number relative to the server socket is used for its establishment. The server listens to the requests on this port. Whenever the connection is established with that well-known port, the server generates a new instance of itself processing in a different thread or manages the request by processing it. Then the actual version listens for the upcoming requests.

Creating and Using Sockets

Socket programming in Perl is exactly similar to UNIX socket implementation useful in writing client-side and server-side socket codes using TCP or UDP protocols.

Following steps are to be followed in order to establish a socket for communicating with a remote server following the UNIX mode,

(i) Create a socket and name it

(ii) Establish a connection between the socket and the remote host on a particular port number.

After the connection is established successfully, the socket identifier can be used like filehandle to perform operations like,

(i) usage of "print" to write to the remote host and

(ii) usage of "angle bracket" operator to listen to the response.

To create a socket, various kinds of protocol numbers which may be platform dependent must be provided. But the socket module is responsible for providing accurate values for any specific platform.

Consider the following example, where a POP3 server is accessed using a simple script and any mail waiting for collection is identified. In a simple client-server communication, a POP3 client gets mail from the POP3 sever. After the connection establishment with the server, authentication is performed, where the username and the password are sent to the server. If everything goes on smoothly then a message is sent by the server indicating the number of available new messages. The client and the server statements are as follows,

Client:	Connection Establishment
Server:	+OK POP3 penelope V6.50 server ready
Client:	USERNAME abcd
Server:	+OK enter the password
Client:	PASSWORD wxyz
Server:	+OK mailbox open, 2 messages.

+OK represents a successful operation. The above statements can be implemented with a Perl script using the following code,

```
use socket;
my ($hostname, $portno, $ipaddress, $prototype, $reply);
$hostname = “Penelope.ecs.soton.ac.uk”;
$portno = 110;
# to convert hostname to IP address use DNS
$ipaddress = inet_aton $hostname
        or die “DNS lookup unsuccessful \n”;
# Retrieve code for TCP protocol
$prototype = get_prototypebyname ‘tcp’;
Socket CHECKMAIL, PF_INET, SOCK_STREAM, $prototype;
Connect CHECKMAIL, sockaddr_in ($portno, $ipaddress) or die “connection unsuccessful : $! \n”;
BODY:{
$! = 1; #fix the socket to auto Flush on output
#bring server response:
#in case of bad replay, exit prematurely from the block
$reply = <CHECKMAIL>;
unless ($reply = ~/\+ok/)
{
print STDERR
“Failure! Bad Reply from Server \n”;
print CHECKMAIL “USERNAME abcd”;
$reply = <CHECKMAIL>;
Unless ($reply = ~/\+ok/)
        {
              print STDERR” username not accepted by Server”;
              last BODY
        }
print CHECKMAIL “PASSWORD wxyz”;
$reply = <CHECKMAIL>;
unless ($reply = ~ /\+ok/)
        {
                print STDERR" password not accepted by server";
                last BODY
            }
        $reply = ~ S/^.*,//;
        print "Reply from Server: \n $reply \n";
            }
        close CHECKMAIL;
```

Q28. Explain the usage of IO :: Socket module.

Answer :

IO :: Socket

For low-level socket functions the IO :: Socket package can be used. A new function creates a reference to socket in the following manner,

$conn = IO :: Socket :: INET ⇒ new (proto ⇒ 'tcp', peeerAddr ⇒ $hostname, PeerPort ⇒ $Portno)

or die "connection with $hostname failed: ($portno) \n";

The job of new function is to create the socket and establish the connection. The returned object reference can be used anywhere like a filehandle.

The socket functions can be obtained as object methods.

Example, To use recv function, $remote ⇒ recv (……) can be called.

2.11 Security Issues

Q29. Explain about the perl's approach to provide security.

Answer : **Model Paper-I, Q2(b)**

Perl's approach to Provide Security

The languages which support complete access the operating system internals may be susceptible to threats.

Example

A script which is badly written may take a user inputted string as a shell command. By issuing a Unix string rm-r* may delete the whole file system. This will become more dangerous when scripts allow network access to untrusted parties. Hence, peruses a method called "taint checking" for providing effective security control.

Taint Checking

This approach assumes that all the incoming data is vulnerable and untrust worthy. Hence, such data should not affect any element outside the program. Taint checking method is called up automatically by the UNIX when the script is running with more permissions than its owner has. This method can also be called upon by calling the perl interpreter with -T option

If taint checking is enabled, then environment variables, command line arguments and file input are marked as tainted. Then, a flow analysis is performed by Perl to determine indirectly tainted data. These data cannot be used in commands which calls a sub-shell which modifies the files, processes or directories. Usage of these data can raise errors referencing a pattern variable set by a regular expression match is the only alternative to bypass the TaintChecks.

The Safe Module

Taint checking offers security for the written code which uses untrusted data from outside. A different case occurs when the code itself is coming from a stranger or untrusted party. The safe module provides an approach to build 'safe objects' each of which is a compartment where execution of the untrusted code takes place.

Separating it from the remaining part of the program. In the package hierarchy, the namespace of a safe object is rooted at a particular level, and it is ensured that the executed code cannot access the variables declared above the specified level. Moreover, a safe obejct can also be created to make particular operators unavailable introducing another security level.

OBJECTIVE TYPE

I. Fill in the Blanks

1. File handler are created by the ____________ function.
2. The last, next and redo are referred to as ____________ and not ____________.
3. The ____________ function combines a list of items into a strings in perl.
4. The I/O channels are named by ____________.
5. The ____________ and ____________ commands are used for reading and writing block of data of known length in perl.
6. The ____________ operator helps in providing robust exception handling mechanism in perl.
7. An effective way of maintaining control by a process in perl is called ____________.
8. A handle on a directory is given by, ____________.
9. ____________ is a unit of code with its own name space.
10. The rich source of information held on web servers, FTP servers, mail servers etc is ____________.

II. Multiple Choice

1. Which of the following loop can iterate over two or more variables simultaneously? [ ]
   (a) While (b) For
   (c) Do - while (d) If - else
2. Which of the following language does not have a case statement? [ ]
   (a) Perl (b) Python
   (c) TCL (d) Java
3. Which of the following function is used by a subroutine to examine the call stack? [ ]
   (a) Sub (b) Call
   (c) Caller (d) Stack
4. Which of the following function resembles join function? [ ]
   (a) Pack (b) Unpack
   (c) Combine (d) None of the above
5. Which of the following operator is used with a wild card pattern in perl? [ ]
   (a) Single quotes (b) * operator
   (c) Eval operator (d) Angle operator
6. Which of the following represents a variable ‘x’ in the package ‘A’? [ ]
   (a) $x (b) $ A : : x
   (c) $ x : : A (d) $ A : x

7. Which of the following is a collection of subroutines that are used in some specific area? [ ]

   (a) Modules (b) Packages

   (c) Libraries (d) All the above

8. Which of the following is an anonymous data structure that is assigned to a class and can be accessed by a reference? [ ]

   (a) Constructor (b) Object

   (c) Method (d) Class

9. Which of the following stores the current environment variables? [ ]

   (a) Special hash % ENV (b) #ENV

   (c) %ENVR (d) $ENV

10. Which of the following package provides an object-oriented wrapper for the low-level socket functions? [ ]

   (a) IO : : FILE (b) IO : : SOCK

   (c) IO : : SOCKET (d) IO : SOCKET

KEY

I. Fill in the Blanks

1. Open
2. Commands, statement
3. Pack
4. Filehandle
5. Sysread, syswrite
6. Eval
7. Taint checking
8. Opendir
9. Package
10. Internet

II. Multiple Choice

1. (b)
2. (a)
3. (c)
4. (a)
5. (d)
6. (b)
7. (c)
8. (b)
9. (a)
10. (c)