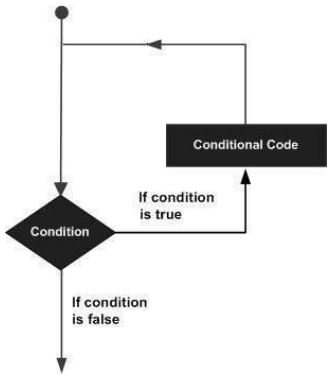
UNIT – 4

ADVANCED PERL

Finer Points Of Looping:

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −



Perl programming language provides the following types of loop to handle the loopingrequirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
until loop	Repeats a statement or group of statements until a given condition becomes true. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
foreach loop	The foreach loop iterates over a normal list value and sets the variable VAR to be each element of the list in turn.
do...while loop	Like a while statement, except that it tests the condition at theend of the loop body
nested loops	You can use one or more loop inside any another while, for ordo..while loop.

Loop Control Statements

Loop control statements change the execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements. Click the following links to check their detail.

Control Statement	Description
next statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
last statement	Terminates the **loop** statement and transfers execution to thestatement immediately following the loop.
continue statement	A continue BLOCK, it is always executed just before theconditional is about to be evaluated again.
redo statement	The redo command restarts the loop block without evaluatingthe conditional again. The continue block, if any, is not executed.
goto statement	Perl supports a goto command with three forms: goto label,goto expr, and goto &name.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

```
#!/usr/local/bin/perl
For(;;)
{
Printf “This loop will run foreve.\n”;
}
```

You can terminate the above infinite loop by pressing the Ctrl + C keys.

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but as a programmer more commonly use the for (;;) construct to signify an infinite loop.

Multiple Loop Variables:

For loop can iterate over two or more variables simultaneously. Eg:

```
for($m=1,$n=1,$m<10,$m++,$n+=2)
{

…….

}
```

Here (,) operator is a list constructor, it evaluates its left hand argument.

Pack Function:

The pack function evaluates the expressions in LIST and packs them into a binary structure specified by EXPR. The format is specified using the characters shown in Table below:
Each character may be optionally followed by a number, which specifies a repeat count for the type of value being packed, that is nibbles, chars, or even bits, according to the format. A value of * repeats for as many values remain in LIST. Values can be unpacked with the unpack function.

For example, a5 indicates that five letters are expected. b32 indicates that 32 bits are expected. h8 indicates that 8 nybbles (or 4 bytes) are expected. P10 indicates that the structure is 10 bytes long.

Following is the simple syntax for this function:

```
pack EXPR, LIST
```

Return Value
This function returns a packed version of the data in LIST using TEMPLATE to determine how it is coded.

Here is the table which gives values to be used in TEMPLATE.

Character	Description
A	ASCII character string padded with null characters
A	ASCII character string padded with spaces
B	String of bits, lowest first
B	String of bits, highest first
C	A signed character (range usually -128 to 127)
C	An unsigned character (usually 8 bits)
D	A double-precision floating-point number
F	A single-precision floating-point number
H	Hexadecimal string, lowest digit first
H	Hexadecimal string, highest digit first
I	A signed integer
I	An unsigned integer
L	A signed long integer
L	An unsigned long integer
N	A short integer in network order
N	A long integer in network order
P	A pointer to a string
S	A signed short integer
S	An unsigned short integer
U	Convert to uuencode format
V	A short integer in VAX (little-endian) order
V	A long integer in VAX order
X	A null byte
X	Indicates "go back one byte"
@	Fill with nulls (ASCII 0)

Example
Following is the example code showing its basic usage −

```
#!/usr/bin/perl -w

$bits =pack("c",65);
# prints A, which is ASCII 65.
print"bits are $bits\n";
$bits =pack("x");
# $bits is now a null chracter.
print"bits are $bits\n";
$bits =pack("sai",255,"T",30);
# creates a seven charcter string on most computers'
print"bits are $bits\n";

@array=unpack("sai","$bits");

#Array now contains three elements: 255, T and 30.
print"Array $array[0]\n";
print"Array $array[1]\n";
print"Array $array[2]\n";
```

When above code is executed, it produces the following result −

```
bits are A
bits are
bits are ?T-
Array 255
Array T
Array 30
```

Unpack Function

The unpack function unpacks the binary string STRING using the format specified in TEMPLATE.Basically reverses the operation of pack, returning the list of packed values according to the supplied format.

You can also prefix any format field with a %<number> to indicate that you want a 16-bit checksum of the value of STRING, instead of the value.

Following is the simple syntax for this function :

unpack TEMPLATE, STRING

Return Value

This function returns the list of unpacked values.

Here is the table which gives values to be used in TEMPLATE.

Character	Description
A	ASCII character string padded with null characters
A	ASCII character string padded with spaces
B	String of bits, lowest first
B	String of bits, highest first
C	A signed character (range usually -128 to 127)
C	An unsigned character (usually 8 bits)
D	A double-precision floating-point number
F	A single-precision floating-point number
H	Hexadecimal string, lowest digit first
H	Hexadecimal string, highest digit first
I	A signed integer
I	An unsigned integer
L	A signed long integer
L	An unsigned long integer
N	A short integer in network order
N	A long integer in network order
P	A pointer to a string
S	A signed short integer
S	An unsigned short integer
U	Convert to uuencode format
V	A short integer in VAX (little-endian) order
V	A long integer in VAX order
X	A null byte
X	Indicates "go back one byte"
@	Fill with nulls (ASCII 0)

Example:

Following is the example code showing its basic usage −

```
#!/usr/bin/perl -w

$bits =pack("c",65);
# prints A, which is ASCII 65.
print"bits are $bits\n";
$bits =pack("x");
# $bits is now a null chracter.
print"bits are $bits\n";
$bits =pack("sai",255,"T",30);
# creates a seven charcter string on most computers'
print"bits are $bits\n";

@array=unpack("sai","$bits");

#Array now contains three elements: 255, A and 47.
print"Array $array[0]\n";
print"Array $array[1]\n";
print"Array $array[2]\n";
```

When above code is executed, it produces the following result −

```
bits are A
bits are
bits are ?T-
Array255
Array T
Array30
```

2.1 Files:

The basics of handling files are simple: you associate a **filehandle** with an external entity (usually a file) and then use a variety of operators and functions within Perl to read and updatethe data stored within the data stream associated with the filehandle.

A filehandle is a named internal Perl structure that associates a physical file with a name. All filehandles are capable of read/write access, so you can read from and update any file or device associated with a filehandle. However, when you associate a filehandle, you can specifythe mode in which the filehandle is opened.

Three basic file handles are - **STDIN**, **STDOUT**, and **STDERR,** which represent standard input, standard output and standard error devices respectively.

Opening and Closing Files

There are following two functions with multiple forms, which can be used to open any new or existing file in Perl.

```
open FILEHANDLE, EXPR
open FILEHANDLE

sysopen FILEHANDLE, FILENAME, MODE, PERMS
sysopen FILEHANDLE, FILENAME, MODE
```

Here FILEHANDLE is the file handle returned by the **open** function and EXPR is the expressionhaving file name and mode of opening the file.

Open Function

Following is the syntax to open **file.txt** in read-only mode. Here less than < sign indicates thatfile has to be opend in read-only mode.

```
open(DATA,"<file.txt");
```

Here DATA is the file handle which will be used to read the file. Here is the example which willopen a file and will print its content over the screen.

```
 #!/usr/bin/perl
 open(DATA,"<file.txt")ordie"Couldn't open file file.txt, $!";
while(<DATA>){
print"$_";
 }
```

Following is the syntax to open file.txt in writing mode. Here less than > sign indicates that filehas to be opend in the writing mode.

```
open(DATA,">file.txt")ordie"Couldn't open file file.txt, $!";
```

This example actually truncates (empties) the file before opening it for writing, which may not be the desired effect. If you want to open a file for reading and writing, you can put a plus sign before the > or < characters.

For example, to open a file for updating without truncating it −

```
open(DATA,"+<file.txt");ordie"Couldn't open file file.txt, $!";
```

To truncate the file first

```
open DATA,"+>file.txt"ordie"Couldn't open file file.txt, $!";
```

You can open a file in the append mode. In this mode writing point will be set to the end of thefile.

```
open(DATA,">>file.txt")||die"Couldn't open file file.txt, $!";
```

A double >> opens the file for appending, placing the file pointer at the end, so that you can immediately start appending information. However, you can't read from it unless you also place a plus sign in front of it −

```
open(DATA,"+>>file.txt")||die"Couldn't open file file.txt, $!";
```

Following is the table which gives the possible values of different modes.

Entities	Definition
< or r	Read Only Access
> or w	Creates, Writes, and Truncates
>> or a	Writes, Appends, and Creates
+< or r+	Reads and Writes
+> or w+	Reads, Writes, Creates, and Truncates
+>> or a+	Reads, Writes, Appends, and Creates

Sysopen Function

The **sysopen** function is similar to the main open function, except that it uses the

system **open()** function, using the parameters supplied to it as the parameters for the system function.

For example, to open a file for updating, emulating the +<**filename** format from open −

```
sysopen(DATA,"file.txt", O_RDWR);
```

Or to truncate the file before updating

```
sysopen(DATA,"file.txt", O_RDWR|O_TRUNC );
```

You can use O_CREAT to create a new file and O_WRONLY- to open file in write only mode andO_RDONLY - to open file in read only mode.

The **PERMS** argument specifies the file permissions for the file specified if it has to be created.By default it takes **0x666**.

Following is the table, which gives the possible values of MODE.

Entities	Definition
O_RDWR	Read and Write
O_RDONLY	Read Only
O_WRONLY	Write Only
O_CREAT	Create the file
O_APPEND	Append the file
O_TRUNC	Truncate the file
O_EXCL	Stops if file already exists
O_NONBLOCK	Non-Blocking usability

Close Function

To close a filehandle, and therefore disassociate the filehandle from the corresponding file, you use the **close** function. This flushes the filehandle's buffers and closes the system's file descriptor.

```
close FILEHANDLE
close
```

If no FILEHANDLE is specified, then it closes the currently selected filehandle. It returns trueonly if it could successfully flush the buffers and close the file.

```
close(DATA)||die"Couldn't close file properly";
```

Reading and Writing Files

Once you have an open filehandle, you need to be able to read and write information. There are a number of different ways of reading and writing data into the file.

The <FILEHANDLE> Operator

The main method of reading the information from an open filehandle is the <FILEHANDLE> operator. In a scalar context, it returns a single line from the filehandle. For example

```
#!/usr/bin/perl
print"What is your name?\n";
$name =<STDIN>;
print"Hello $name\n";
```

When you use the <FILEHANDLE> operator in a list context, it returns a list of lines from the specified filehandle. For example, to import all the lines from a file into an array

```
#!/usr/bin/perl
open(DATA,"<import.txt")ordie"Can't open data";
@lines=<DATA>;
close(DATA);
```

getc Function

The getc function returns a single character from the specified FILEHANDLE, or STDIN if none is specified

```
getc FILEHANDLE
getc
```

If there was an error, or the filehandle is at end of file, then undef is returned instead.

read Function

The read function reads a block of information from the buffered filehandle: This

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

function isused to read binary data from the file.

The length of the data read is defined by LENGTH, and the data is placed at the start of SCALARif no OFFSET is specified. Otherwise data is placed after OFFSET bytes in SCALAR. The function returns the number of bytes read on success, zero at end of file, or undef if there was an error.

print Function

For all the different methods used for reading information from filehandles, the main function for writing information back is the print function.

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

The print function prints the evaluated value of LIST to FILEHANDLE, or to the current outputfilehandle (STDOUT by default). For example

```
print "Hello World!\n";
```

Copying Files

Here is the example, which opens an existing file file1.txt and read it line by line and generateanother copy file file2.txt.

```
#!/usr/bin/perl
# Open file to read
open(DATA1,"<file1.txt");
# Open new file to write
open(DATA2,">file2.txt");
# Copy data from one file to another.
while(<DATA1>)
{
print DATA2 $_;
}
close( DATA1 );
close( DATA2 );
```

Renaming a file
Here is an example, which shows how we can rename a file file1.txt to file2.txt. Assuming file is available in /usr/test directory.

```
#!/usr/bin/perl
rename("/usr/test/file1.txt","/usr/test/file2.txt");
```

This function **renames** the takes two arguments and it just rename existing file.

Deleting an Existing File
Here is an example, which shows how to delete a file file1.txt using the **unlink** function.

```
#!/usr/bin/perl
unlink("/usr/test/file1.txt");
```

Positioning inside a File
You can use to **tell** function to know the current position of a file and **seek**function to point aparticular position inside the file.

tell Function
The first requirement is to find your position within a file, which you do using the tell function

```
tell FILEHANDLE
```

```
tell
```

This returns the position of the file pointer, in bytes, within FILEHANDLE if specified, or thecurrent default selected filehandle if none is specified.

seek Function
The seek function positions the file pointer to the specified number of bytes within a file

```
seek FILEHANDLE, POSITION, WHENCE
```

The function uses the fseek system function, and you have the same ability to position relative to three different points: the start, the end, and the current position. You do this by specifying a value for WHENCE.

Zero sets the positioning relative to the start of the file. For example, the line sets the file pointer to the 256th byte in the file.

```
seek DATA, 256, 0;
```

File Information

You can test certain features very quickly within Perl using a series of test operators known collectively as -X tests. For example, to perform a quick test of the various permissions on afile, you might use a script like this

```
#/usr/bin/perl
my $file ="/usr/test/file1.txt";
my(@description, $size);
if(-e $file)
{
push@description,'binary'if(-B _);
```

```
push@description,'a socket'if(-S _);
push@description,'a text file'if(-T _);
push@description,'a block special file'if(-b _);
push@description,'a character special file'if(-c _);
push@description,'a directory'if(-d _);
push@description,'executable'if(-x _);
push@description,(($size =-s _))?"$size bytes":'empty';
print"$file is ", join(', ',@description),"\n";
}
```

Here is the list of features, which you can check for a file or directory

Operator	Definition
-A	Script start time minus file last access time, in days.
-B	Is it a binary file?
-C	Script start time minus file last inode change time, in days.
-M	Script start time minus file modification time, in days.
-O	Is the file owned by the real user ID?

-R	Is the file readable by the real user ID or real group?
-S	Is the file a socket?
-T	Is it a text file?
-W	Is the file writable by the real user ID or real group?
-X	Is the file executable by the real user ID or real group?
-b	Is it a block special file?
-c	Is it a character special file?
-d	Is the file a directory?
-e	Does the file exist?
-f	Is it a plain file?
-g	Does the file have the setgid bit set?
-k	Does the file have the sticky bit set?
-l	Is the file a symbolic link?
-o	Is the file owned by the effective user ID?
-p	Is the file a named pipe?
-r	Is the file readable by the effective user or group ID?
-s	Returns the size of the file, zero size = empty file.
-t	Is the filehandle opened by a TTY (terminal)?
-u	Does the file have the setuid bit set?
-w	Is the file writable by the effective user or group ID?
-x	Is the file executable by the effective user or group ID?
-z	Is the file size zero?

EVAL
The eval operator comes in 2 forms:

In the first form , eval takes an arbitrary string as an operand and evaluates the string and executed in the current context.

The value returned is the value of the last expression evaluated.

In case of syntax error or runtime error, eval returns the value “undefined” and placesthe error in the variable $@

Ex:

$myvar =’ …’;

….

$value = eval “ \$$myvar “;

In the second form ,it takes a block as an argument and the block is compiled only once.

If there is a runtime error,the error is returned in $@.

Instead of try we use eval and instead of catch we test $@

Ex:

```
Eval
{
…….

}
If ($@ ne ‘ ‘)
{
…..

}
```

Data Structures:

ARRAYS OF ARRAYS

- In perl a two dimensional array is constructed by creating an array of references to anonymous arrays.
  For ex: @colors = ([35,39,43] , [4,5,8] , [32,31,25]) ;

- The array composer converts each comma-separated list to an anonymous array in memory and returns a reference and when we write an exp. Like
  $colors [0][1] = 39;

  $colors [0] is a reference to an array and 2nd subscript represents the element present in that array.

- A two dimensional array can be dynamically created by using PUSH operator to add areference to an anonymous array to the top level array.
- For ex: we are interested in converting a table set of data having white spaces between the fields can be converted to two dimensional array by repeteadly using split to put thefields of a line into list and then using push to add the reference to an array.

  While (<STDIN>)
  Push @table , [split]

  }

COMPLEX DATA STRUCTURES

- Not only an array of arrays can be created but we can create hashes of hashes ,arrays ofhashes and hashes of arrays.
- By combining all these possibilities ,data structures of great complexity can be createdex: doubly linked list.
- We can make an element of the array a hash containing three fields with keys 'L'(left neighbour) ,' R'(right neighbour) and 'C'(content).
- The values related to L and R are references to element hashes and the value of C canbe anything(scalar,variables,hash ,reference).

  Ex:
  We can move forwards along the list with

$current = $current->{‘R’} ;And backwards with
$current = $current->{‘L’} ;Create a new element
$new = { L =>undef , R=>undef , C=>…} ;

And we can insert new element after current element as
$new->{‘R’}=$current->{‘R’} ;
$current{‘R’}->{‘L’}= $new;
$current{‘R’}=$new ;
$new->{‘L’} = $current ;
And the current element can be deleted as
$current->{‘L’}->{‘R’} = $current->{‘R’} ;
$current->{‘R’}->{‘L’} = $current->{‘L’} ;

2.2 Packages:

Packages are the basis for libraries, modules and objects.

It is the unit of code with its own namespace (i.e. separate symbol table), which determines bindings of names both at compile-time and run-time.
Initially code runs in default package main.

Variables used in a package are global to that package only.

Ex:$A::x is the variable x in package A.Package A ;
$x = 0;

…….

Package B ;

$x = 1 ;

…….

Package A ;

Print $x;

output: zero.

The package B declaration switches to a different symbol table then the package A

points to the original symbol table having $x =0;.

Nested packages can be created of the form A::B provided the variables should be of the fully qualified form Ex>$A::B::x.

A package can have one or more BEGIN routines and also END routines.

Package declaration is rarely used on its own.

Modules:

Libraries and modules are packages contained within a single file and are units of program reusability.

The power of perl is increased by the usage of modules that provide functionality in specific application areas.

To be fact module is nothing but a package contained in a separate file whose name is same as the package name with the extension .pm and makes use of built-in-support.

The use of modules make mathematical routines in the library math.p1 are converted into a module math.pm and can be written as

Ex: Use math ; at the start of the program and the subroutines are available .

The subroutine names imported are those defined in the export list of the math module and it is possible to supress the import of names but loses the point of the module.

Ex: use IO : : File ;

Indicates a requirement for the module File.pm which will be found in a directory called IO.

The use of " use math ('sin' , 'cos' , 'tan')" is same as
BEGIN {
Require " Math.pm" ;

Math :: import (‘sin’ , ‘cos’ , ‘tan’);

}

The module names are imported by calling the import() method defined in the module.

The package writer is free to define import() in any way.

Objects:
Objects in Perl provide a similar functionality as objects in real object oriented programming(OOP), but in a different way. They use the same terminology as OOP, but the words have different meanings as given below.

Object: An **object** with in Perl is a reference to a data type that knows what class it belongs to. The object is stored as a reference in a scalar variable. The object is said to be blessed into a class: this is done by calling the built in function **bless** in a constructor.

Constructor: A constructor is just a subroutine that returns a reference to an object.

Class: A class is a package that provides methods to deal with objects that belong to it.

Method: A method is a subroutine that expects an object reference as its first argument.

Constructors:

Objects are created by a constructor subroutine which is generally called **new.**
Eg.

```
Package Animal;
sub new { my $ref ={ };
 bless ref;
 return  ref;
}
```

The flower brackets { } returns a reference to an anonymous hash. So the **new** constructor returns a reference to an object that is an empty hash, and knows that it

belongs to the package Animal.

Instances:

We can create the instances for the object with this defined constructor as

$Dougal = new Animal;

$Ermyntrude = new Animal;

This makes $Dougal and $Ermyntrude references to objects that are empty hashes, andknow that they belong to the Animal class.

Method Invocation:

Perl supports two syntactic forms for invoking methods one is by using arrow operator and another one is by using Indirect objects. If a class is used to invoke the method, that argumentwill be the name of the class. If an object is used to invoke the method, that argument will be the reference to the object. Whichever it is, we'll call it the method's invocant. For a class method, the invocant is the name of a package. For an instance method, the invocant is a reference that specifies an object.

Method Invocation Using the Arrow Operator:

For example if set_species, get_species are the methods they can be invoked using arrowoperator as follows.

$Dougal -> set_species 'Dog';

$Dougal_is ->= $Dougal->get_species;

Method Invocation Using Indirect Objects:

The methods can be invoked by using indirect objects as given belowset_species
$Dougal, 'Dog';
$Dougal_is = get _species $Dougal;

Attributes:

Subroutine declarations and definitions may optionally have attribute lists associated

with them. An attribute is a piece of data belonging to a particular object. Unlike most object- oriented languages, Perl provides no special syntax or support for declaring and manipulating

attributes. Attributes are often stored in the object itself. For example, if the object is an anonymous hash, we can store the attribute values in the hash using the attribute name as thekey.

E.g: sub species {my $self =shift;
my $was = $self->{'species'};

- - - - - - - - - -
- - - - - - - -
}

Class Methods And Attributes:

There are operations that are relevant to the class and not need to operate on a specific instance are called class methods or static methods.
Similarly attributes that are common to all instances of a class are called as class attributes. Class attributes are just package global variables and class methods are just subroutines that donot require an object reference as the first argument e.g the new constructor.

Inheritance:

Perl only provides method inheritance. Inheritance is realized by including a special array @ISAin the package that defines the derived class.
For single inheritance @ISA is an array of one element, the name of the base class. Multiple inheritance can be realized by making @Isa an array of more than one element.
Each element in the array @ISA is the name of the another package that is being used as aclass.

If a method cannot be found, the packages referenced in @Isa are recursively searched, depthfirst. The current class is derived class and those referenced in @ISA are the base classes.

e.g :

```
package Employee;use Person;
  use strict;
```

our @ISA = qw(Person); # inherits from Person

Interfacing to the OS:
Creating Internet Ware Applications:

The internet is a rich source of information, held on web servers, FTP servers, POP/IMAP mail servers, news servers etc. A web browser can access information on web servers and FTP servers, and clients access mail and news servers. however, this is not the way of to the information: an 'internet-aware' application can access a server and collect the information without manual intervention. For suppose that a website offers 'lookup' facility in which the user a query by filling in a then clicks the 'submit' button . the data from the form in sent to a CGI program on the server(probably written in which retrieves the information, formats it as a webpage, and returns the page to the browser. A perl application can establish a connection to the server, send the request in the format that the browser would use, collect the returned HTML and then extract the fields that form the answer to the query. In the same way, a perl application can establish a connection to a POP3 mail server and send a request which will result in the server returning a message listing the number of currently unread messages.

Much of the power of scripting languages comes from the way in which they hide the complexity of operations, and this is particularly the case when we make use of specialized modules: tasks that might pages of code in C are achieved in few lines. The LWP (library for WWW access in perl) collection of modules is a very good case in point it makes the kind of interaction described above almost trivial. The LWP:: simple module is a interface to web servers. it can be achieved by exploiting modules, LWP::simple we can retrieve the contents of a web page in a statement:
use LWP::simple $url=...http://www.somesite.com/index.html..;

$page=get($url);

Dirty Hands Internet Programming:

Modules like LWP: : Simple and LWP: :User Agent meet the needs of most programmers requiring web access, and there are numerous other modules for other types of Internet access.

EX:

Net: : FTP for access to FTP servers

Some tasks may require a lower level of access to the network, and this is provided by Perl bothin the form of modules (e.g IO: : Socket) and at an even lower level by built-in functions.

Support for network programming in perl is so complete that you can use the language to writeany conceivable internet application

Access to the internet at this level involves the use of sockets, and we explain what a socket isbefore getting down to details of the programming.
Sockets are network communication channels, providing a bi-directional channel betweenprocesses on different machines.

Sockets were originally a feature of UNIX: other UNIX systems adopted them and the socketbecame the de facto mechanism of network communication in the UNIX world.

The popular Winsock provided similar functionality for Windows, allowing Windows systems to communicate over the network with UNIX systems, and sockets are a built-in feature of Windows 9X and WindowsNT4.

From the Perl programmer's point a network socket can be treated like an open file it isidentified by a you write to it with print, and read it from operator.
The socket interface is based on the TCP/IP protocol suite, so that all information is handledautomatically.

In TCP a reliable channel, with automatic recovery from data loss or corruption: for this reasona TCP connection is often described as a virtual circuit.
The socket in Perl is an exact mirror of the UNIX and also permits connections using UDP(Unreliable Datagram Protocol).

Security Issues:

One big source of security problems in Perl scripts is improperly validated (or unvalidated) user input. Any time your program might take input from an untrusted user, even indirectly, you should be cautious. For example, if you are writing CGI scripts in Perl, expect that malicious users will send you bogus input.