PERL TUTORIAL

PERL Introduction

Perl is a general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, GUI development, and more.

What You Should Already Know

If you have basic knowledge of C or UNIX Shell then PERL is very easy to learn. If this is your first language to learn then you may take one to two week to be fluent in PERL

What is PERL?

- Perl is a stable, cross platform programming language.
- Perl stands for Practical Extraction and Report Language.
- It is used for mission critical projects in the public and private sectors.
- Perl is Open Source software, licensed under its Artistic License, or theGNU General Public License (GPL).
- Perl was created by Larry Wall.
- Perl 1.0 was released to usenet's alt.comp.sources in 1987
- *PC Magazine* named Perl a finalist for its 1998 Technical Excellence Award in the Development Tool category.
- Perl is listed in the Oxford English Dictionary.

Supported Operating Systems

- Unix systems
- Macintosh - (OS 7-9 and X) see The MacPerl Pages.
- Windows - see ActiveState Tools Corp.
- VMS
- And many more...

PERL Features

- Perl takes the best features from other languages, such as C, awk, sed, sh, and BASIC, among others.
- Perls database integration interface (DBI) supports third-party databases including Oracle, Sybase, Postgres, MySQL and others.
- Perl works with HTML, XML, and other mark-up languages.
- Perl supports Unicode.
- Perl is Y2K compliant.
- Perl supports both procedural and object-oriented programming.
- Perl interfaces with external C/C++ libraries through XS or SWIG.

- Perl is extensible. There are over 500 third party modules available

  from the Comprehensive Perl Archive Network (CPAN).

- The Perl interpreter can be embedded into other systems.

PERL and the Web

- Perl is the most popular web programming language due to its text manipulation capabilities and rapid development cycle.
- Perl is widely known as " the duct-tape of the Internet".
- Perl's CGI.pm module, part of Perl's standard distribution, makes handling HTML forms simple.
- Perl can handle encrypted Web data, including e-commerce transactions.
- Perl can be embedded into web servers to speed up processing by as much as 2000%.
- mod_perl allows the Apache web server to embed a Perl interpreter.
- Perl's DBI package makes web-database integration easy.

Is Perl Compiled or Interpreted

Good question. The sort answer is interpreted, which means that your code can be run as is, without a compilation stage that creates a nonportable executebale program.

Tradional compilers convert programs into machine language. When you run a Perl program, it's first compiled into a bytecode, which is then converted (as the program runs) into machine instructions. So it is not quite the same as shells, or Tcl, which are "strictly" interpreted without an intermediate representation. Nor it is like most versions of C or C++, which are compiled directly into a machine dependent format. It is somewhere in between, along with *Python* and *awk* and Emacs .elc files.

PERL Syntax Overview

A Perl script or program consists of one or more statements. These statements are simply written in the script in a straightforward fashion. There is no need to have a main() function or anything of that kind.

Perl statements end in a semi-colon:

```
print "Hello, world";
```

Comments start with a hash symbol and run to the end of the line:

```
# This is a comment
```

Whitespace is irrelevant:

```
print      "Hello, world";
```

... except inside quoted strings:

```
# this would print with a linebreak in the middle
print "Hello
          world";
```

Double quotes or single quotes may be used around literal strings:

```
print "Hello, world";
print 'Hello, world';
```

However, only double quotes "interpolate" variables and special characters such as newlines (\n):

```
print "Hello, $name\n";     # works fine

print 'Hello, $name\n';     # prints $name\n literally
```

Numbers don't need quotes around them:

```
print 42;
```

You can use parentheses for functions' arguments or omit them according to your personal taste. They are only required occasionally to clarify issues of precedence. Following two statements produce same result.

```
print("Hello, world\n");
print "Hello, world\n";
```

PERL File Extension:

A PERL script can be created inside of any normal simple-text editor program. There are several programs available for every type of platform. There are many programs designed for programmers available for download on the web.

Regardless of the program you choose to use, a PERL file must be saved with a .pl (.PL) file extension in order to be recognized as a functioning PERL script. File names can contain numbers, symbols, and letters but must not contain a space. Use an underscore (_) in places of spaces.

First PERL Program:

Assuming you are already on Unix $ prompt. So now open a text file hello.pl using vi or vim editor and put the following lines inside your file.

```
#!/usr/bin/perl

# This will print "Hello, World" on the screen
```

```
print "Hello, world";
```

#!/usr/bin is the path where you have installed PERL

Execute PERL Script:

Before you execute your script be sure to change the mode of the script file and give execution priviledge, generally a setting of 0755 works perfectly.

Now to run hello.pl Perl program from the Unix command line, issue the following command at your UNIX $ prompt:

```
$perl hello.pl
```

This will produce following result:

```
Hello, World
```

Perl Command Line Flags:

Command line flags affect how Perl runs your program.

```
$perl -v
```

This will produce following result:

```
This is perl, v5.001 built for i386-linux-thread-multi
.................
```

You can use -e option at command line which lets you execute Perl statements from the command line.

```
$perl -e 'print 4;\n'
RESULT: 4
$perl -e "print 'Hello World!\n";'
RESULT: Hello World!
```

Perl has three built in variable types:

- Scalar
- Array
- Hash

PERL Variable Types

Scalar variable type

A scalar represents a single value as follows:

```
my $animal = "camel"; my $answer = 42;
```

Here **my** is a keyword which has been explained in the same section at the bottom.

A scalar values can be strings, integers or floating point numbers, and Perl will automatically convert between them as required. There is no need to pre-declare your variable types. Scalar values can be used in various ways:

```
print $animal;
print "The animal is $animal\n";
print "The square of $answer is ", $answer * $answer, "\n";
```

There are a number of "magic" scalars with names that look like punctuation or line noise. These special variables are used for all kinds of purposes and they wioll be discussed in Special Variables sections. The only one you need to know about for now is **$_** which is the "default variable". It's used as the default argument to a number of functions in Perl, and it's set implicitly by certain looping constructs.

```
print;       # prints contents of $_ by default
```

Array variable type:

An array represents a list of values:

```
my @animals = ("camel", "llama", "owl");

my @numbers = (23, 42, 69);

my @mixed   = ("camel", 42, 1.23);
```

Arrays are zero-indexed but you can change this setting by changing default variable $[or $ARRAY_BASE. Here's how you get at elements in an array:

```
print $animals[0];              # prints "camel"
print $animals[1];              # prints "llama"
```

The special variable $#array tells you the index of the last element of an array:

```
print $mixed[$#mixed];       # last element, prints 1.23
```

You might be tempted to use $#array + 1 to tell you how many items there are in an array. Don't bother. As it happens, using @array where Perl expects to find a scalar value ("in scalar context") will give you the number of elements in the array:

```
if (@animals < 5) { ... } # Here @animals will return 3
```

The elements we're getting from the array start with a **$** because we're getting just a single value out of the array -- you ask for a scalar, you get a scalar.

To get multiple values from an array:

```
@animals[0,1];          # gives ("camel", "llama");

@animals[0..2];         # gives ("camel", "llama", "owl");

@animals[1..$#animals]; # gives all except the first element
```

This is called an "array slice". You can do various useful things to lists as follows:

```
my @sorted    = sort @animals;

my @backwards = reverse @numbers;

# Here sort and reverse are Perl's built-in functions
```

There are a couple of special arrays too, such as **@ARGV** (the command line arguments to your script) and **@_** (the arguments passed to a subroutine). These are documented in next section "Special Variables".

Hash variable type:

A hash represents a set of key/value pairs. Actaully hash are type of arrays with the exception that hash index could be a number or string. They are prefixed by**%** sign as follows:

```
my %fruit_color = ("apple", "red", "banana", "yellow");
```

You can use whitespace and the => operator to lay them out more nicely:

```
my %fruit_color = (
   apple  => "red",

   banana => "yellow",
);
```

To get a hash elements:

```
$fruit_color{"apple"};           # gives "red"
```

You can get at lists of keys and values with keys() and values() built-in functions.

```
my @fruits = keys %fruit_colors;

 my @colors = values %fruit_colors;
```

Hashes have no particular internal order, though you can sort the keys and loop through them. Just like special scalars and arrays, there are also special hashes. The most well known of these is %ENV which contains environment variables.

More complex data types can be constructed using references, which allow you to build lists and hashes within lists and hashes. A reference is a scalar value and can refer to any other Perl data type. So by storing a reference as the value of an array or hash element, you can easily create lists and hashes within lists and hashes.

The following example shows a 2 level hash of hash structure using anonymous hash references.

```
my $variables = {

   scalar  =>  {
      description => "single item",
      sigil => '$',

     },
   array   =>  {
      description => "ordered list of items",

      sigil => '@',
     },
   hash    =>  {

      description => "key/value pairs",
      sigil => '%',
     },

};
```

Following line will print $

```
 print "$variables->{'scalar'}->{'sigil'}\n"    ;
```

Variable Context:

PERL treats same variable differently based on Context. For example

```
my @animals = ("camel", "llama", "owl");
```

Here @animals is an array, but when it is used in scalar context then it returnes number of elements contacined in it as following.

```
if (@animals < 5) { ... } # Here @animals will return 3
```

Another examples:

```
my $number = 30;
```

Here $number is an scalar and contained number in it but when it is called along with a string then it becomes number which is 0, in stead of string:

```
$result = "This is " + "$number";
print "$result";
```

Here output will be 30

Escaping Characters:

In PERL we use the backslash (\) character to escape any type of character that might interfere with our code. Below is the example

```
$result = "This is " . "\"number\"";
print "$result";
```

Here output will be This is "number"

Case Sensitivity:

Variable names are case sensitive; $foo, $FOO, and $fOo are all separate variables as far as Perl is concerned.

Variable Scoping:

Throughout the previous section all the examples have used the following syntax:

```
my $var = "value";
```

The my is actually not required; you could just use:

```
$var = "value";
```

However, the above usage will create global variables throughout your program, which is bad programming practice. But **my** creates lexically scoped variables instead. The variables are scoped to the block (i.e. a bunch of statements surrounded by curly-braces) in which they are defined. Have a look at the following example:

```
my $a = "foo";
if ($some_condition) {

   my $b = "bar";
   print $a;    # prints "foo"

   print $b;    # prints "bar"
}
print $a;       # prints "foo"

print $b;       # prints nothing; $b has fallen out of scope
```

Using **my** in combination with a use **strict**; at the top of your Perl scripts means that the interpreter will pick up certain common programming errors. For instance, in the example above, the final print $b would cause a compile-time error and prevent you from running the program. Using **strict** is highly recommended. Following is the usage:

```
use strict;

my $a = "foo";
if ($some_condition) {

   my $b = "bar";
   print $a;     # prints "foo"

   print $b;     # prints "bar"
}
print $a;        # prints "foo"

print $b;        # prints nothing; $b has fallen out of scope
```

Private Variables via my():

The **my** operator declares the listed variables to be lexically confined to the following but not limited to

- Enclosing blocks,
- Conditional (if/unless/elsif/else)
- Loop (for/foreach/while/until/continue)
- Subroutine
- eval or do/require/use'd file

If more than one value is listed, the list must be placed in parentheses. All listed elements must be legal lvalues. Scoped--magical built-ins like $/ must currently be localized with local instead.

```
my $foo;           # declare $foo lexically local
my (@wid, %get);           # declare list of variables local
my $foo = "flurp"; # declare $foo lexical, and init it
my @oof = @bar;   # declare @oof lexical, and init it
my $x : Foo = $y;          # similar, with an attribute applied
```

Lexical scopes of control structures are not bounded precisely by the braces that delimit their controlled blocks; control expressions are part of that scope, too. Thus in the loop the scope of $line extends from its declaration throughout the rest of the loop construct (including the continue clause), but not beyond it.

```
while (my $line = <>) {
   $line = lc $line;
}continue {
    print $line;
}
```

Similarly, in the conditional the scope of $answer extends from its declaration through the rest of that conditional, including any elsif and else clauses, but not beyond it.

```
if ((my $answer = <STDIN>> =~ /^yes$/i) {
     user_agrees();
} elsif ($answer =~ /^no$/i) {
     user_disagrees();
} else {
chomp $answer;
     die "'$answer' is neither 'yes' nor 'no'";
}
```

We encourage the use of lexically scoped variables. Use the following line at the top of your program file to avoid any error. This will remind you to scope the variable using **local** or **my** keyword.

```
use strict 'vars';
```

Temporary Values via local():

A **local** modifies its listed variables to be "local" to the enclosing block, eval, or do FILE --and to any subroutine called from within that block. A local just gives temporary values to global (meaning package) variables. It does not create a local variable. This is known as dynamic scoping.

Lexical scoping is done with **my**, which works more like C's auto declarations.

```
local $foo;             # make $foo dynamically local
local (@wid, %get);     # make list of variables local
local $foo = "flurp";   # make $foo dynamic, and init it
local @oof = @bar;      # make @oof dynamic, and init it
```

Because local is a run-time operator, it gets executed each time through a loop. Consequently, it's more efficient to localize your variables outside the loop.

If you localize a special variable, you'll be giving a new value to it, but its magic won't go away. That means that all side-effects related to this magic still work with the localized value. This feature allows code like this to work :

```
# Read the whole contents of FILE in $slurp
{ local $/ = undef; $slurp = <FILE> ; }
```

PERL Scalar Variable

Scalar variables are simple variables containing only one element--a string or a number. Strings may contain any symbol, letter, or number. Numbers may contain exponents, integers, or decimal values. The bottom line here with scalar variables is that they contain only one single piece of data. What you see is what you get with scalar variables.

Following are examples of Scalar Variables

```
#!/usr/bin/perl

$number = "5";
$exponent = "2 ** 8";
$string = "Hello, PERL!";
$float = 12.39;

# We can also assign a scalar an empty (undefined) value:
$nothing = undef;

# Printing all the above values
print "$number\n";
print "$exponent\n";
```

```
print "$string\n";
print "$float\n";
print "There is nothing: $nothing\n";

This will give following result
5
2 ** 8
Hello, PERL!
12.39
There is nothing:
```

Scalar Strings

Strings are scalar as we mentioned previously. There is no limit to the size of the string, any amount of characters, symbols, or words can make up your strings.

When defining a string you may use single or double quotations, you may also define them with the q subfunction. Following are examples of how to define strings

```
$single = 'This string is single quoted';
$double = "This string is double quoted";
$userdefined = q^Carrot is now our quote^;
```

Formatting Strings w/ Formatting Characters

Strings can be formatted to your liking using formatting characters. Some of these characters also work to format files created in PERL. Think of these characters as miniature functions.

Character	Description
\L	Transform all letters to lowercase
\l	Transform the next letter to lowercase
\U	Transform all letters to uppercase
\u	Transform the next letter to uppercase
\n	Begin on a new line
\r	Applys a carriage return
\t	Applys a tab to the string
\f	Applys a formfedd to the string
\b	Backspace
\a	Bell
\e	Escapes the next character
\0nn	Creates Octal formatted numbers
\xnn	Creates Hexideciamal formatted numbers
\cX	Control characters, x may be any character
\Q	Backslash (escape) all following non-alphanumeric characters.
\E	Ends \U, \L, or \Q functions

Here are few examples

```
#!/usr/bin/perl

$newline = "Welcome to \ntutorialspoint.com!";
```

```
$small = "\uthis, Here t will become capital !"
$ALLCAPS = "\UThis completely will become capital!";
$PARTIALCAPS = "\UThis half will/E become capital!";
$backslah = "\Q'Hello World'\E!";

print "$newline\n";
print "$small\n";
print "$ALLCAPS\n";
print "$PARTAILCAPS\n";
print "$backslah\n";

This will give following result
Welcome to
tutorialspoint.com!
This, Here t will become capital !
THIS COMPLETELY WILL BECOME CAPITAL!
THIS HALF WILL become capital!
\'Hello World\'!
```

Substr() and String Indexing

The substr() function allows for the temporary replacement of characters in a string. We can change the string "Hello, PERL" to "Hello, World!" quite easily. Each character of the string is automatically assigned a numeric value by PERL, which means that we can index any of the characters in our strings with this number. PERL counts each character of the string beginning with 0 for the first character and continuing until it reaches the end of a string

Two arguments must be sent with our substr() function, the string you wish to index and the index number. If two arguments are sent, PERL assumes that you are replacing every character from that index number to the end.

```
#!/usr/bin/perl

# Define a string to replace
$mystring = "Hello, PERL!";

print "Before replacement : $mystring\n";

substr($mystring, 7) = "World!";

print "After replacement : $mystring\n";

This will give following result
Before replacement : Hello, PERL!
After replacement : Hello, World!
```

Because we only specified one numeric parameter for the string, PERL assumed we wanted to replace every character after the 7th, with our new string. If we throw a third parameter in our function we can replace only a chunk of our string with a new string.

```
#!/usr/bin/perl
$mystring = "Hello, PERL!";

print "Before replacement : $mystring\n";

substr($mystring, 3, 6) = "World!";
```

```
print "After replacement : $mystring\n";

This will give following result
Before replacement : Hello, PERL!
After replacement : HelWorld!RL!
```

Multiline Strings

If you want to introduce multiline strings into your programs, you can use standard quotes:

```
$string = 'This is
a multiline
string';
```

But this is messy and is subject to the same basic laws regarding interpolation and quote usage. We could get around it using the q// or qq// operator with different delimiters.

Another better way of printing multilines

```
print <<EOF;
This is
a multiline
string
EOF
```

V-Strings

V-strings can be a useful way of introducing version numbers and IP addresses into Perl. They are any literal that begins with a v and is followed by one or more dot-separated elements. For example:

```
$name = v77.97.114.116.105.110;
```

Numeric Scalars

Perl supports a number of a fairly basic methods for specifying a numeric literal in decimal:

```
$num = 123;      # integer
$num = 123.45;   # floating point
$num = 1.23e45;  # scientific notation
```

You can also specify literals in hexadecimal, octal, and binary by specifying a preceding character to highlight the number types:

```
$num = 0xff;        # Hexadecimal
$num = 0377;        # Octal
$num = 0b0010_0000; # Binary
```


Note that the numbers are not stored internally using these bases.Perl converts the literal representation into a decimal internally.

PERL Array Variable

An array is just a set of scalars. It's made up of a list of individual scalars that are stored within a single variable. You can refer to each scalar within that list using a numerical index.

Array Creation:

Array variables are prefixed with the @ sign and are populated using either parentheses or the qw operator. For example:

```
@array = (1, 2, 'Hello');
@array = qw/This is an array/;
```

The second line uses the qw// operator, which returns a list of strings, separating the delimited string by white space. In this example, this leads to a four-element array; the first element is 'this' and last (fourth) is 'array'. This means that you can use newlines within the specification:

```
@days = qw/Monday
Tuesday
...
Sunday/;
```

We can also populate an array by assigning each value individually:

```
$array[0] = 'Monday';
...
$array[6] = 'Sunday';
```

Extracting Individual Indices

When extracting individual elements from an array, you must prefix the variable with a dollar sign and then append the element index within square brackets after the name. For example:

```
#!/usr/bin/perl

@shortdays = qw/Mon Tue Wed Thu Fri Sat Sun/;
print $shortdays[1];

This will print
Tue
```

Array indices start at zero, so in the preceding example we.ve actually printed "Tue". You can also give a negative index.in which case you select the element from the end, rather than the beginning, of the array. This means that

```
print $shortdays[0]; # Outputs Mon
print $shortdays[6]; # Outputs Sun
print $shortdays[-1]; # Also outputs Sun
print $shortdays[-7]; # Outputs Mon
```

Sequential Number Arrays

PERL offers a shortcut for sequential numbers and letters. Rather than typing out each element when counting to 100 for example, we can do something like this:

```
#!/usr/bin/perl

@10 = (1 .. 10);
@100 = (1 .. 100;
@1000 = (100 .. 1000);
@abc = (a .. z);

print "@10";   # Prints number starting from 1 to 10
print "@100";  # Prints number starting from 1 to 100
print "@1000"; # Prints number starting from 1 to 1000
print "@abc";  # Prints number starting from a to z
```

Array Size

The size of an array can be determined using scalar context on the array - the returned value will be the number of elements in the array:

```
@array = (1,2,3);
print "Size: ",scalar @array,"\n";
```

The value returned will always be the physical size of the array, not the number of valid elements. You can demonstrate this, and the difference between scalar @array and $#array, using this fragment:

```
#!/uer/bin/perl

@array = (1,2,3);
$array[50] = 4;

print "Size: ",scalar @array,"\n";
print "Max Index: ", $#array,"\n";

This will return
Size: 51
Max Index: 50
```

There are only four elements in the array that contain information, but the array is 51 elements long, with a highest index of 50.


Here **scalar** function is used to enforce scalar context so that @array can return size of the array otherwise @array will return a lisrt of all the elements contacined in it.

Adding and Removing Elements in Array

Use the following functions to add/remove and elements:

- **push():** adds an element to the end of an array.
- **unshift():** adds an element to the beginning of an array.

- **pop():** removes the last element of an array.
- **shift() :** removes the first element of an array.

When adding elements using push() or shift() you must specify two arguments, first the array name and second the name of the element to add. Removing an element with pop() or shift() only requires that you send the array as an argument.

```
#!/usr/bin/perl

# Define an array
@coins = ("Quarter","Dime","Nickel");
print "First Statement : @coins";
print "\n";

# Add one element at the end of the array
push(@coins, "Penny");
print "Second Statement : @coins";
print "\n";
# Add one element at the beginning of the array
unshift(@coins, "Dollar");
print "Third Statement : @coins";
print "\n";

# Remove one element from the last of the array.
pop(@coins);
print "Fourth Statement : @coins";
print "\n";
# Remove one element from the beginning of the array.
shift(@coins);
print "Fifth Statement : @coins";
print "@coins";


Now this will produce following result
First Statement : Quarter Dime Nickel
Second Statement : Quarter Dime Nickel Penny
Third Statement : Dollar Quarter Dime Nickel Penny
Fourth Statement : Dollar Quarter Dime Nickel
Fifth Statement : Quarter Dime Nickel
```

Slicing Array Elements

You can also extract a "slice" from an array - that is, you can select more than one item from an array in order to produce another array.

```
@weekdays = @shortdays[0,1,2,3,4];
```

The specification for a slice must a list of valid indices, either positive or negative, each separated by a comma. For speed, you can also use the **..** range operator:

```
@weekdays = @shortdays[0..4];
```

Ranges also work in lists:

```
@weekdays = @shortdays[0..2,6,7];
```

Replacing Array Elements

Replacing elements is possible with the splice() function. Splice() requires a handful of arguments and the formula reads:

```
splice(@array,first-element,sequential_length, new elements)
```

Essentially, you send PERL an array to splice, then direct it to the starting element, count through how many elements to replace, and then fill in the missing elements with new information.

```
#!/usr/bin/perl

@nums = (1..20);
splice(@nums, 5,5,21..25);
print "@nums";
```

Here actual replacement begins after the 5th element, starting with the number 6. Five elements are then replaced from 6-10 with the numbers 21-25

Transform Strings to Arrays

With the split function, it is possible to transform a string into an array. To do this simply define an array and set it equal to a split function. The split function requires two arguments, first the character of which to split and also the string variable.

```
#!/usr/bin/perl

# Define Strings
$astring = "Rain-Drops-On-Roses-And-Whiskers-On-Kittens";
$namelist = "Larry,David,Roger,Ken,Michael,Tom";

# Strings are now arrays. Here '-' and ',' works as delimeter
@array = split('-',$astring);
@names = split(',',$namelist);

print $array[3];  # This will print Roses
print "\n";       # This is a new line
print $names[4];  # This will print Michael
```

Likewise, we can use the join() function to rejoin the array elements and form one long, scalar string.

```
#!/usr/bin/perl

# Define Strings
$astring = "Rain-Drops-On-Roses-And-Whiskers-On-Kittens";
$namelist = "Larry,David,Roger,Ken,Michael,Tom";

# Strings are now arrays. Here '-' and ',' works as delimeter
@array = split('-',$astring);
@names = split(',',$namelist);

$string1 = join(",", @array);
$string2 = join("-", @names);
```

```
print $string1;
print "\n" ;
print $string2;


This will produce following result
Rain,Drops,On,Roses,And,Whiskers,On,Kittens
Larry-David-Roger-Ken-Michael-Tom
```

Sorting Arrays

The sort() function sorts each element of an array according to ASCII Numeric standards.

```
#!/usr/bin/perl

# Define an array
@foods = qw(pizza steak chicken burgers);
print "Before sorting: @foods\n";

# Sort this array
@foods = sort(@foods);
print "After sorting: @foods\n";

This will produce following result
Before sorting: pizza steak chicken burgers
After sorting: burgers chicken pizza steak
```


Please note that sorting is performed based on ASCII Numeric value of the words. So the best option is to first transform every element of the array into lowercase letters and then perform the sort function.

The $[Special Variable

$[is a special variable. This particular variable is a scalar containing the first index of all arrays. because Perl arrays have zero-based indexing, $[will almost always be 0. But if you set $[to 1 then all your arrays will use on-based indexing. It is recommended not to use any other indexing other than zero.

The Lists

Lists are really a special type of array - .essentially, a list is a temporary construct that holds a series of values. The list can be "hand" generated using parentheses and the comma operator,

```
@array = (1,2,3);
```

or it can be the value returned by a function or variable when evaluated in list context:

```
print join(',' @array);
```

Here, the @array is being evaluated in list context because the join function is expecting a list.

Merging Lists (or Arrays)

Because a list is just a comma-separated sequence of values, you can combine lists together:

```
@numbers = (1,3,(4,5,6));
```

The embedded list just becomes part of the main list.this also means that we can combine arrays together:

```
@numbers = (@odd,@even);
```

Functions that return lists can also be embedded to produce a single, final list:

```
@numbers = (primes(),squares());
```

Selecting Elements from Lists

The list notation is identical to that for arrays - .you can extract an element from an array by appending square brackets to the list and giving one or more indices:

```
#!/usr/bin/perl

$one = (5,4,3,2,1)[4];

print "Value of \$one is $one\n"

This will produce follwoing result
Value of $one is 1
```

Similarly, we can extract slices, although without the requirement for a leading @ character:

```
#!/usr/bin/perl

@newlist = (5,4,3,2,1)[1..3];

print "value of new list is @newlist\n";

This will produce follwoing result
value of new list is 4 3 2
```

PERL Hash Variable

Hashes are an advanced form of array. One of the limitations of an array is that the information contained within it can be difficult to get to. For example, imagine that you have a list of people and their ages.

The hash solves this problem very neatly by allowing us to access that @ages array not by an index, but by a scalar key. For example to use age of different people we can use thier names as key to define a hash.

```
%ages = ('Martin' => 28,
         'Sharon' => 35,
         'Rikke' => 29,);

print "Rikke is $ages{Rikke} years old\n";
```

```
This will produce following result
Rikke is 29 years old
```

Creation of Hash

Hashes are created in one of two ways. In the first, you assign a value to a named key on a one-by-one basis:

```
$ages{Martin} = 28;
```

In the second case, you use a list, which is converted by taking individual pairs from the list: the first element of the pair is used as the key, and the second, as the value. For example,

```
%hash = ('Fred' , 'Flintstone', 'Barney', 'Rubble');
```

For clarity, you can use => as an alias for , to indicate the key/value pairs:

```
%hash = ('Fred' => 'Flintstone',
         'Barney' => 'Rubble');
```

Extracting Individual Elements

You can extract individual elements from a hash by specifying the key for the value that you want within braces:

```
print $hash{Fred};

This will print following result
Flintstone
```

Extracting Slices

You can extract slices out of a hash just as you can extract slices from an array. You do, however, need to use the @ prefix because the return value will be a list of corresponding values:

```
#!/uer/bin/perl

%hash = (-Fred => 'Flintstone', -Barney => 'Rubble');
print join("\n",@hash{-Fred,-Barney});

This will produce following result
Flintstone
Rubble
```

Note: Using $hash{-Fred, -Barney} would return nothing.

Extracting Keys and Values

You can get a list of all of the keys from a hash by using keys

```
#!/usr/bin/perl

%ages = ('Martin' => 28, 'Sharon' => 35, 'Rikke' => 29);
print "The following are in the DB: ",join(', ',values
%ages),"\n";

This will produce following result
The following are in the DB: 29, 28, 35
```

These can be useful in loops when you want to print all of the contents of a hash:

```
#!/usr/bin/perl

%ages = ('Martin' => 28, 'Sharon' => 35, 'Rikke' => 29);
foreach $key (%ages)
{
  print "$key is $ages{$key} years  old\n";
}

This will produce following result
Rikke is 29 years  old
29 is  years  old
Martin is 28 years  old
28 is  years  old
Sharon is 35 years  old
35 is  years  old
```

The problem with this approach is that (%ages) returns a list of values. So to resolve this problem we have **each** function which will retun us **key** and **value** pair as given below

```
#!/usr/bin/perl

%ages = ('Martin' => 28, 'Sharon' => 35, 'Rikke' => 29);
while (($key, $value) = each %ages)
{
  print "$key is $ages{$key} years old\n";
}

This will produce following result
Rikke is 29 years old
Martin is 28 years old
Sharon is 35 years old
```

Checking for Existence

If you try to access a key/value pair from a hash that doesn.t exist, you.ll normally get the undefined value, and if you have warnings switched on, then you.ll get a warning generated at run time. You can get around this by using the **exists**function, which returns true if the named key exists, irrespective of what its value might be:

```
#!/usr/bin/perl

%ages = ('Martin' => 28, 'Sharon' => 35, 'Rikke' => 29);
if (exists($ages{"mohammad"}))
```

```
{
  print "mohammad if $ages{$name} years old\n";
}
else
{
  print "I don't know the age of mohammad\n";
}

This will produce following result
I don't know the age of mohammad
```

Sorting/Ordering Hashes

There is no way to simply guarantee that the order in which a list of keys, values, or key/value pairs will always be the same. In fact, it's best not even to rely on the order between two sequential evaluations:

```
#!/usr/bin/perl

print(join(', ',keys %hash),"\n");
print(join(', ',keys %hash),"\n");
```

If you want to guarantee the order, use **sort**, as, for example:

```
  print(join(', ',sort keys %hash),"\n");
```

If you are accessing a hash a number of times and want to use the same order, consider creating a single array to hold the sorted sequence, and then use the array (which will remain in sorted order) to iterate over the hash. For example:

```
my @sortorder = sort keys %hash;
foreach my $key (@sortorder)
```

Hash Size

You get the size - that is, the number of elements - from a hash by using scalar context on either keys or values:

```
#!/usr/bin/perl

%ages = ('Martin' => 28, 'Sharon' => 35, 'Rikke' => 29);
print "Hash size: ",scalar keys %ages,"\n";

This will produce following result
Hash size: 3
```

Add & Remove Elements in Hashes

Adding a new key/value pair can be done with one line of code using simple assignment operator. But to remove an element from the hash you need to use**delete** function.

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

```
%ages = ('Martin' => 28, 'Sharon' => 35, 'Rikke' => 29);

# Add one more element in the hash
$age{'John'} = 40;
# Remove one element from the hash
delete( $age{'Sharon'} );
```

PERL Special Variables

Some variables have a predefined and special meaning in Perl. They are the variables that use punctuation characters after the usual variable indicator ($, @, or %), such as $_ (explained below).

Most of the special variables have an english like long name eg. Operating System Error variable $! can be written as $OS_ERROR. But if you are going to use english like names then you would have to put one line "use English;" at the top of your program file. This guides the interter to pickup exact meaning of the variable.

The most commonly used special variable is $_, which contains the default input and pattern-searching string. For example, in the following lines:

```
#!/usr/bin/perl

foreach ('hickory','dickory','doc') {
         print;
         print "\n";
}


This will produce following result
hickory
dickory
doc
```

The first time the loop is executed, "hickory" is printed. The second time around, "dickory" is printed, and the third time, "doc" is printed. That's because in each iteration of the loop, the current string is placed in $_, and is used by default by print. Here are the places where Perl will assume $_ even if you don't specify it:

- Various unary functions, including functions like ord and int, as well as the all file tests (-f, -d) except for -t, which defaults to STDIN.
- Various list functions like print and unlink.
- The pattern-matching operations m//, s///, and tr/// when used without an =~ operator.
- The default iterator variable in a foreach loop if no other variable is supplied.
- The implicit iterator variable in the grep and map functions.
- The default place to put an input record when a line-input operation's result is tested by itself as the sole criterion of a while test (i.e.,). Note that outside of a while test, this will not happen.

Special variable types

Based on the usage and nature of special variables we can categories them in the following categories

- Global Scalar Special Variables.
- Global Array Special Variables.
- Global Hash Special Variables.
- Global Special Filehandles.
- Global Special Constants.
- Regular Expression Special Variables.
- Filehandle Special Variables.

Global Scalar Special Variables

Here is the list of all the scalar special variables. We have listed corresponding english like names along with the symbolic names.

$_	The default input and pattern-searching space.
$ARG	
$.	The current input line number of the last filehandle that was read. An explicit close on the filehandle resets the line number.
$NR	
$/	The input record separator; newline by default. If set to the null string, it treats blank lines as delimiters.
$RS	
$,	The output field separator for the print operator.
$OFS	
$\	The output record separator for the print operator.
$ORS	
$"	Like "$," except that it applies to list values interpolated into a double-quoted string (or similar interpreted string). Default is a space.
$LIST_SEPARATOR	
$;	The subscript separator for multidimensional array emulation. Default is "\034".
$SUBSCRIPT_SEPARATOR	
$^L	What a format outputs to perform a formfeed. Default is "\f".
$FORMAT_FORMFEED	
$:	The current set of characters after which a string may be broken to fill continuation fields (starting with ^) in a format. Default is "\n"".
$FORMAT_LINE_BREAK_CHARACTERS	
$^A	The current value of the write accumulator for format lines.
$ACCUMULATOR	

$#	Contains the output format for printed numbers (deprecated).
$OFMT	
$?	The status returned by the last pipe close, backtick (``) command, or system operator.
$CHILD_ERROR	
$!	If used in a numeric context, yields the current value of the errno variable, identifying the last system call error. If used in a string context, yields the corresponding system error string.
$OS_ERROR or $ERRNO	
$@	The Perl syntax error message from the last eval command.
$EVAL_ERROR	
$$	The pid of the Perl process running this script.
$PROCESS_ID or $PID	
$<	The real user ID (uid) of this process.
$REAL_USER_ID or $UID	
$>	The effective user ID of this process.
$EFFECTIVE_USER_ID or $EUID	
$(	The real group ID (gid) of this process.
$REAL_GROUP_ID or $GID	
$)	The effective gid of this process.
$EFFECTIVE_GROUP_ID or $EGID	
$0	Contains the name of the file containing the Perl script being executed.
$PROGRAM_NAME	
$[	The index of the first element in an array and of the first character in a substring. Default is 0.
$]	Returns the version plus patchlevel divided by 1000.
$PERL_VERSION	
$^D	The current value of the debugging flags.
$DEBUGGING	
$^E	Extended error message on some platforms.
$EXTENDED_OS_ERROR	
$^F	The maximum system file descriptor, ordinarily 2.
$SYSTEM_FD_MAX	
$^H	Contains internal compiler hints enabled by certain pragmatic modules.

$^I $INPLACE_EDIT	The current value of the inplace-edit extension. Use undef to disable inplace editing.
$^M 	The contents of $M can be used as an emergency memory pool in case Perl dies with an out-of-memory error. Use of $M requires a special compilation of Perl. See the INSTALL document for more information.
$^O $OSNAME	Contains the name of the operating system that the current Perl binary was compiled for.
$^P $PERLDB	The internal flag that the debugger clears so that it doesn't debug itself.
$^T $BASETIME	The time at which the script began running, in seconds since the epoch.
$^W $WARNING	The current value of the warning switch, either true or false.
$^X $EXECUTABLE_NAME	The name that the Perl binary itself was executed as.
$ARGV 	Contains the name of the current file when reading from .

Global Array Special Variables

@ARGV	The array containing the command-line arguments intended for the script.
@INC	The array containing the list of places to look for Perl scripts to be evaluated by the do, require, or use constructs.
@F	The array into which the input lines are split when the -a command-line switch is given.

Global Hash Special Variables

%INC	The hash containing entries for the filename of each file that has been included via do or require.
%ENV	The hash containing your current environment.
%SIG	The hash used to set signal handlers for various signals.

Global Special Filehandles

ARGV	The special filehandle that iterates over command line filenames in @ARGV. Usually written as the null filehandle in <>.

STDERR	The special filehandle for standard error in any package.
STDIN	The special filehandle for standard input in any package.
STDOUT	The special filehandle for standard output in any package.
DATA	The special filehandle that refers to anything following the __END__ token in the file containing the script. Or, the special filehandle for anything following the __DATA__ token in a required file, as long as you're reading data in the same package __DATA__ was found in.
_ (underscore)	The special filehandle used to cache the information from the last stat, lstat, or file test operator.

Global Special Constants

__END__	Indicates the logical end of your program. Any following text is ignored, but may be read via the DATA filehandle.
__FILE__	Represents the filename at the point in your program where it's used. Not interpolated into strings.
__LINE__	Represents the current line number. Not interpolated into strings.
__PACKAGE__	Represents the current package name at compile time, or undefined if there is no current package. Not interpolated into strings.

Regular Expression Special Variables

$digit	Contains the text matched by the corresponding set of parentheses in the last pattern matched. For example, $1 matches whatever was contained in the first set of parentheses in the previous regular expression.
$&	The string matched by the last successful pattern match.
$MATCH	
$`	The string preceding whatever was matched by the last successful pattern match.
$PREMATCH	
$'	The string following whatever was matched by the last successful pattern match.
$POSTMATCH	
$+	The last bracket matched by the last search pattern. This is useful if you don't know which of a set of alternative patterns was matched. For example: /Version: (.*)|Revision: (.*)/ && ($rev = $+);
$LAST_PAREN_MATCH	

Filehandle Special Variables

$|	If set to nonzero, forces an fflush(3) after every write or print on the currently selected output channel.
$OUTPUT_AUTOFLUSH	

$%	The current page number of the currently selected output channel.
$FORMAT_PAGE_NUMBER	
$=	The current page length (printable lines) of the currently selected output channel. Default is 60.
$FORMAT_LINES_PER_PAGE	
$-	The number of lines left on the page of the currently selected output channel.
$FORMAT_LINES_LEFT	
$~	The name of the current report format for the currently selected output channel. Default is the name of the filehandle.
$FORMAT_NAME	
$^	The name of the current top-of-page format for the currently selected output channel. Default is the name of the filehandle with _TOP appended.
$FORMAT_TOP_NAME	

PERL Conditional Statements

The conditional statements are **if** and **unless**, and they allow you to control the execution of your script. There are five different formats for the **if** statement:

```
if (EXPR)
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ...
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
STATEMENT if (EXPR)
```

The first format is classed as a simple statement, since it can be used at the end of another statement without requiring a block, as in:

```
print "Happy Birthday!\n" if ($date == $today);
```

In this instance, the message will only be printed if the expression evaluates to a true value.

The second format is the more familiar conditional statement that you may have come across in other languages:

```
if ($date == $today)
{
   print "Happy Birthday!\n";
}
```

This produces the same result as the previous example.

The third format allows for exceptions. If the expression evaluates to true, then the first block is executed; otherwise (else), the second block is executed:

```
if ($date == $today)
{
   print "Happy Birthday!\n";
}
else
{
   print "Happy Unbirthday!\n";
}
```

The fourth form allows for additional tests if the first expression does not return true. The elsif can be repeated an infinite number of times to test as many different alternatives as are required:

```
if ($date == $today)
{
   print "Happy Birthday!\n";
}
elsif ($date == $christmas)
{
   print "Happy Christmas!\n";
}
```

The fifth form allows for both additional tests and a final exception if all the other tests fail:

```
if ($date == $today)
{
   print "Happy Birthday!\n";
}
elsif ($date == $christmas)
{
   print "Happy Christmas!\n";
}else
{
   print "Happy Unbirthday!\n";
}
```

The unless statement automatically implies the logical opposite of **if**, so unless the**EXPR** is true, execute the block. This means that the statement

```
print "Happy Unbirthday!\n" unless ($date == $today);

is equivalent to

print "Happy Unbirthday!\n" if ($date != $today);
```

For example, the following is a less elegant solution to the preceding if...else. Although it achieves the same result, example:

```
unless ($date != $today)
{
   print "Happy Unbirthday!\n";
}
else
{
   print "Happy Birthday!\n";
}
```

The final conditional statement is actually an operator.the conditional operator. It is synonymous with the if...else conditional statement but is shorter and more compact. The format for the operator is:

```
(expression) ? (statement if true) : (statement if false)
```

For example, we can emulate the previous example as follows:

```
($date == $today) ? print "Happy B.Day!\n" : print "Happy
Day!\n";
```

PERL Loops

Perl supports four main loop types:

1. while
2. for
3. until
4. foreach

In each case, the execution of the loop continues until the evaluation of the supplied expression changes.

- In the case of a **while** loop execution continues while the expression evaluates to true.
- The until loop executes while the loop expression is false and only stops when the expression evaluates to a true value.
- The list forms of the **for** and **foreach** loop are special cases. They continue until the end of the supplied list is reached.

while Loops

The while loop has three forms:

```
while EXPRLABEL
while (EXPR) BLOCKLABEL
while (EXPR) BLOCK continue BLOCK
```

In first form, the expression is evaluated first, and then the statement to which it applies is evaluated. For example, the following line increases the value of $linecount as long as we continue to read lines from a given file:

For example, the following line increases the value of $linecount as long as we continue to read lines from a given file:

```
$linecount++ while ();
```

To create a loop that executes statements first, and then tests an expression, you need to combine while

with a preceding **do {}** statement. For example:

```
do
{
   $calc += ($fact*$ivalue);
} while $calc <100;
```

In this case, the code block is executed first, and the conditional expression is only evaluated at the end of each loop iteration.

The second two forms of the while loop repeatedly execute the code block as long as the result from the conditional expression is true. For example, you could rewrite the preceding example as:

```
while($calc < 100)
{
   $calc += ($fact*$ivalue);
}
```

The **continue** block is executed immediately after the main block and is primarily used as a method for executing a given statement (or statements) for each iteration, irrespective of how the current iteration terminated. It is somehow equivalent to for loop

```
{
   my $i = 0;
   while ($i <100)
   { ... }
   continue
   {
      $i++;
   }
}

This is equivalent to
for (my $i = 0; $i < 100; $i++)
{ ... }
```

for Loops

A for loop is basically a while loop with an additional expression used to reevaluate the original conditional expression. The basic format is:

```
LABEL for (EXPR; EXPR; EXPR) BLOCK
```

The first EXPR is the initialization - the value of the variables before the loop starts iterating. The second is the expression to be executed for each iteration of the loop as a test. The third expression is executed for each iteration and should be a modifier for the loop variables.

Thus, you can write a loop to iterate 100 times like this:

```
for ($i=0;$i<100;$i++)
{
...
}
```

You can place multiple variables into the expressions using the standard list operator (the comma):

```
for ($i=0, $j=0;$i<100;$i++,$j++)
```

You can create an infinite loop like this:

```
for(;;)
{
...
}
```

until Loops

The inverse of the while loop is the until loop, which evaluates the conditional expression and reiterates over the loop only when the expression returns false. Once the expression returns true, the loop ends.

In the case of a do.until loop, the conditional expression is only evaluated at the end of the code block. In an until (EXPR) BLOCK loop, the expression is evaluated before the block executes. Using an until loop, you could rewrite the previous example as:

```
do
{
  $calc += ($fact*$ivalue);
} until $calc >= 100;

This is equivalent to

do
{
  $calc += ($fact*$ivalue);
} while $calc <100;
```

foreach Loops

The last loop type is the foreach loop, which has a format like this:

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

Using a for loop, you can iterate through the list using:

```
for ($index=0;$index<=@months;$index++)
{
   print "$months[$index]\n";
}
```

This is messy, because you.re manually selecting the individual elements from the array and using an additional variable, $index, to extract the information. Using a foreach loop, you can simplify the process:

```
foreach (@months)
{
```

```
   print "$_\n";
}
```

The foreach loop can even be used to iterate through a hash, providing you return the list of values or keys from the hash as the list:

```
foreach $key (keys %monthstonum)
{
   print "Month $monthstonum{$key} is $key\n";
}
```

Labled Loops

Labels can be applied to any block, but they make the most sense on loops. By giving your loop a name, you allow the loop control keywords to specify which loop their operation should be applied to. The format for a labeled loop is:

```
LABEL: loop (EXPR) BLOCK ...
```

For example, to label a for loop:

```
ITERATE: for (my $i=1; $i<100; $i++)
{
   print "Count: $i\n";
}
```

Loop Control - next, last and redo

There are three loop control keywords: next, last, and redo.

The **next** keyword skips the remainder of the code block, forcing the loop to proceed to the next value in the loop. For example:

```
while (<DATA>)
{
   next if /^#/;
}
```

Above code would skip lines from the file if they started with a hash symbol. If there is a **continue** block, it is executed before execution proceeds to the next iteration of the loop.

The **last** keyword ends the loop entirely, skipping the remaining statements in the code block, as well as dropping out of the loop. The last keyword is therefore identical to the break keyword in C and Shellscript. For example:

```
while ()
{
   last if ($found);
}
```

Would exit the loop if the value of $found was true, whether the end of the file had actually been reached or

not. The **continue** block is not executed.

The **redo** keyword reexecutes the code block without reevaluating the conditional statement for the loop. This skips the remainder of the code block and also the **continue** block before the main code block is reexecuted. For example, the following code would read the next line from a file if the current line terminates with a backslash:

```
while(<DATA>)
{
   if (s#\\$#)
   {
      $_ .= <DATA>;
      redo;
   }
}
```

Here is an example showing how labels are used in inner and outer loops

```
OUTER:
   while(<DATA>)
   {
      chomp;
      @linearray=split;
      foreach $word (@linearray)
      {
         next OUTER if ($word =~ /next/i)
      }
   }
```

goto Statement

There are three basic forms: **goto LABEL, goto EXPR,** and **goto &NAME**. In each case, execution is moved from the current location to the destination.

In the case of goto **LABEL**, execution stops at the current point and resumes at the point of the label specified.

The **goto &NAME** statement is more complex. It allows you to replace the currently executing subroutine with a call to the specified subroutine instead.

PERL Built-in Operators

There are many Perl operators but here are a few of the most common ones:

Arithmetic Operators

```
+   addition
-   subtraction
*   multiplication
/   division
```

Numeric Comparison Operators

```
==  equality
!=  inequality
<   less than
>   greater than
<=  less than or equal
>=  greater than or equal
```

String Comparison Operators

```
eq  equality
ne  inequality
lt  less than
gt  greater than
le  less than or equal
ge  greater than or equal
```

(Why do we have separate numeric and string comparisons? Because we don't have special variable types, and Perl needs to know whether to sort numerically (where 99 is less than 100) or alphabetically (where 100 comes before 99).

Boolean Logic Operators

```
&&  and
||  or
!   not
```

(`and` , `or` and `not` aren't just in the above table as descriptions of the operators -- they're also supported as operators in their own right. They're more readable than the C-style operators, but have different precedence to `&&` and friend.

Miscellaneous Operators

```
=   assignment
.   string concatenation
x   string multiplication
..  range operator (creates a list of numbers)
```

Many operators can be combined with a = as follows:

```
$a += 1;        # same as $a = $a + 1
$a -= 1;        # same as $a = $a - 1
$a .= "\n";     # same as $a = $a . "\n";
```

Operator Precedence and Associativity

Perl operators have the following associativity and precedence, listed from highest precedence to lowest. Operators borrowed from C keep the same precedence relationship with each other, even where C's precedence is slightly screwy. (This makes learning Perl easier for C folks.) With very few exceptions, these all operate on scalar values only, not array values.

```
left  terms and list operators (leftward)
left  ->
nonassoc     ++ --
right        **
right        ! ~ \ and unary + and -
left  =~ !~
left  * / % x
left  + - .
left  << >>
nonassoc     named unary operators
nonassoc     < > <= >= lt gt le ge
nonassoc     == != <=> eq ne cmp
left  &
left  | ^
left  &&
left  ||
nonassoc     ..  ...
right        ?:
right        = += -= *= etc.
left  , =>
nonassoc     list operators (rightward)
right        not
left  and
left  or xor
```

PERL Files & I/O

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 update the 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 specify the mode in which the filehandle is opened.

Three basic file handles are - **STDIN**, **STDOUT**, and **STDERR**.

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 open function and EXPR is the expression having file name and mode of opening the file.

Following is the syntax to open file.txt in read-only mode. Here less than < signe indicates that file 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 will open a file and will print its content over the screen.

```
#!/usr/bin/perl

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

while(<DATA>)
{
   print "$_";
}
```

Open Function

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

```
open(DATA, ">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");
```

To truncate the file first:

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

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

```
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 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 and O_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 possible values of MODE

```
Value            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 true only if it could successfully flush the buffers and close the file.

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

Reading and Writing Filehandles

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 <FILEHANDL> 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") or die "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 function is used to read binary data from the file.

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

The length of the data read is defined by LENGTH, and the data is placed at the start of SCALAR if 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 output filehandle (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 generate another copy 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 **rename** takes two arguments and it just rename existing file

Deleting an exiting file

Here is an example which shows how to delete a file file1.txt using **unlink** function.

```
#!/usr/bin/perl

unlink ("/usr/test/file1.txt");
```

Locating Your Position Within a File

You can use to **tell** function to know the current position of a file and **seek** function to point a particular 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 the current 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;
```

Getting 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 a file, you might use a script like this:

```
#/usr/bin/perl
```

```
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

```
Operator Description
-A      Age of file (at script startup) in days since
modification.
-B      Is it a binary file?
-C      Age of file (at script startup) in days since
modification.
-M      Age of file (at script startup) in days since
modification.
-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?
```

Working with Directories

Following are the standard functions used to play with directories.

```
opendir DIRHANDLE, EXPR  # To open a directory
readdir DIRHANDLE        # To read a directory
rewinddir DIRHANDLE      # Positioning pointer to the begining
```

```
telldir DIRHANDLE        # Returns current position of the dir
seekdir DIRHANDLE, POS   # Pointing pointer to POS inside dir
closedir DIRHANDLE       # Closing a directory.
```

Here is an example which opens a directory and list out all the files available inside this directory.

```
#!/usr/bin/perl

opendir (DIR, '.') or die "Couldn't open directory, $!";
while ($file = readdir DIR)
{
  print "$file\n";
}
closedir DIR;
```

Another example to print the list of C source code files, you might use

```
#!/usr/bin/perl

opendir(DIR, '.') or die "Couldn't open directory, $!";
foreach (sort grep(/^.*\.c$/,readdir(DIR)))
{
   print "$_\n";
}
closedir DIR;
```

You can make a new directory using the **mkdir** function:

To remove a directory, use the **rmdir** function:

To change the directory you can use **chdir** function.

PERL Regular Expressions

A regular expression is a string of characters that define the pattern or patterns you are viewing. The syntax of regular expressions in Perl is very similar to what you will find within other regular expression.supporting programs, such as **sed**, **grep**, and**awk**.

The basic method for applying a regular expression is to use the pattern binding operators =~ and !~. The first operator is a test and assignment operator.

There are three regular expression operators within Perl

- Match Regular Expression - m//
- Substitute Regular Expression - s///
- Transliterate Regular Expression - tr///

The forward slashes in each case act as delimiters for the regular expression (regex) that you are specifying.

If you are comfortable with any other delimiter then you can use in place of forward slash.

The Match Operator

The match operator, m//, is used to match a string or statement to a regular expression. For example, to match the character sequence "foo" against the scalar $bar, you might use a statement like this:

```
if ($bar =~ /foo/)
```

The m// actually works in the same fashion as the q// operator series.you can use any combination of naturally matching characters to act as delimiters for the expression. For example, m{}, m(), and m>< are all valid.

You can omit the m from m// if the delimiters are forward slashes, but for all other delimiters you must use the m prefix.

Note that the entire match expression.that is the expression on the left of =~ or !~ and the match operator, returns true (in a scalar context) if the expression matches. Therefore the statement:

```
$true = ($foo =~ m/foo/);
```

Will set $true to 1 if $foo matches the regex, or 0 if the match fails.

In a list context, the match returns the contents of any grouped expressions. For example, when extracting the hours, minutes, and seconds from a time string, we can use:

```
my ($hours, $minutes, $seconds) = ($time =~ m/(\d+):(\d+):(\d+)/);
```

Match Operator Modifiers

The match operator supports its own set of modifiers. The /g modifier allows for global matching. The /i modifier will make the match case insensitive. Here is the complete list of modifiers

```
Modifier Description
i        Makes the match case insensitive
m        Specifies that if the string has newline or carriage
         return characters, the ^ and $ operators will now
         match against a newline boundary, instead of a
         string boundary
o        Evaluates the expression only once
s        Allows use of . to match a newline character
x        Allows you to use white space in the expression for
clarity
g        Globally finds all matches
cg       Allows the search to continue even after a global match
fails
```

Matching Only Once

There is also a simpler version of the match operator - the ?PATTERN? operator. This is basically identical to the m// operator except that it only matches once within the string you are searching between each call to reset.

For example, you can use this to get the first and last elements within a list:

```
#!/usr/bin/perl

@list = qw/food foosball subeo footnote terfoot canic footbrdige/;

foreach (@list)
{
   $first = $1 if ?(foo.*)?;
   $last = $1 if /(foo.*)/;
}
print "First: $first, Last: $last\n";

This will produce following result
First: food, Last: footbrdige
```

The Substitution Operator

The substitution operator, s///, is really just an extension of the match operator that allows you to replace the text matched with some new text. The basic form of the operator is:

```
s/PATTERN/REPLACEMENT/;
```

The PATTERN is the regular expression for the text that we are looking for. The REPLACEMENT is a specification for the text or regular expression that we want to use to replace the found text with.

For example, we can replace all occurrences of .dog. with .cat. using

```
$string =~ s/dog/cat/;
```

Another example:

```
#/user/bin/perl

$string = 'The cat sat on the mat';
$string =~ s/cat/dog/;

print "Final Result is $string\n";

This will produce following result

The dog sat on the mat
```

Substitution Operator Modifiers

Here is the list of all modifiers used with substitution operator

```
Modifier Description
i        Makes the match case insensitive
m        Specifies that if the string has newline or carriage
         return characters, the ^ and $ operators will now
         match against a newline boundary, instead of a
         string boundary
```

```
o       Evaluates the expression only once
s       Allows use of . to match a newline character
x       Allows you to use white space in the expression
        for clarity
g       Replaces all occurrences of the found expression
        with the replacement text
e       Evaluates the replacement as if it were a Perl statement,
        and uses its return value as the replacement text
```

Translation

Translation is similar, but not identical, to the principles of substitution, but unlike substitution, translation (or transliteration) does not use regular expressions for its search on replacement values. The translation operators are:

```
tr/SEARCHLIST/REPLACEMENTLIST/cds
y/SEARCHLIST/REPLACEMENTLIST/cds
```

The translation replaces all occurrences of the characters in SEARCHLIST with the corresponding characters in REPLACEMENTLIST. For example, using the "The cat sat on the mat." string we have been using in this chapter:

```
#/user/bin/perl

$string = 'The cat sat on the mat';
$string =~ tr/a/o/;

print "$string\n";

This will produce following result

The cot sot on the mot.
```

Standard Perl ranges can also be used, allowing you to specify ranges of characters either by letter or numerical value. To change the case of the string, you might use following syntax in place of the **uc** function.

```
$string =~ tr/a-z/A-Z/;
```

Translation Operator Modifiers

Following is the list of operators related to translation

```
Modifier         Description
c       Complement SEARCHLIST.
d       Delete found but unreplaced characters.
s       Squash duplicate replaced characters.
```

The /d modifier deletes the characters matching SEARCHLIST that do not have a corresponding entry in REPLACEMENTLIST. For example:

```
#!/usr/bin/perl

$string = 'the cat sat on the mat.';
$string =~ tr/a-z/b/d;
```

```
print "$string\n";

This will produce following result
b b   b.
```

The last modifier, /s, removes the duplicate sequences of characters that were replaced, so:

```
#!/usr/bin/perl

$string = 'food';
$string = 'food';
$string =~ tr/a-z/a-z/s;

print $string;

This will produce following result
fod
```

More complex regular expressions

You don't just have to match on fixed strings. In fact, you can match on just about anything you could dream of by using more complex regular expressions. Here's a quick cheat sheet:

Character	Description
.	a single character
\s	a whitespace character (space, tab, newline)
\S	non-whitespace character
\d	a digit (0-9)
\D	a non-digit
\w	a word character (a-z, A-Z, 0-9, _)
\W	a non-word character
[aeiou]	matches a single character in the given set
[^aeiou]	matches a single character outside the given set
(foo|bar|baz)	matches any of the alternatives specified

Quantifiers can be used to specify how many of the previous thing you want to match on, where "thing" means either a literal character, one of the metacharacters listed above, or a group of characters or metacharacters in parentheses.

Character	Description
*	zero or more of the previous thing
+	one or more of the previous thing
?	zero or one of the previous thing
{3}	matches exactly 3 of the previous thing
{3,6}	matches between 3 and 6 of the previous thing
{3,}	matches 3 or more of the previous thing

The ^ metacharacter matches the beginning of the string and the $ metasymbol matches the end of the string.

Here are some brief examples

```
# nothing in the string (start and end are adjacent)
```

```
/^$/

# a three digits, each followed by a whitespace
# character (eg "3 4 5 ")
/(\d\s){3}/

# matches a string in which every
# odd-numbered letter is a (eg "abacadaf")
/(a.)+/

# string starts with one or more digits
/^\d+/

# string that ends with one or more digits
/\d+$/
```

Lets have alook at another example

```
#!/usr/bin/perl

$string = "Cats go Catatonic\nWhen given Catnip";
($start) = ($string =~ /\A(.*?) /);
@lines = $string =~ /^(.*?) /gm;
print "First word: $start\n","Line starts: @lines\n";


This will produce following result
First word: Cats
Line starts: Cats When
```

Matching Boundaries

The **\b** matches at any word boundary, as defined by the difference between the \w class and the \W class. Because \w includes the characters for a word, and \W the opposite, this normally means the termination of a word. The **\B** assertion matches any position that is not a word boundary. For example:

```
/\bcat\b/ # Matches 'the cat sat' but not 'cat on the mat'
/\Bcat\B/ # Matches 'verification' but not 'the cat on the mat'
/\bcat\B/ # Matches 'catatonic' but not 'polecat'
/\Bcat\b/ # Matches 'polecat' but not 'catatonic'
```

Selecting Alternatives

The | character is just like the standard or bitwise OR within Perl. It specifies alternate matches within a regular expression or group. For example, to match "cat" or "dog" in an expression, you might use this:

```
if ($string =~ /cat|dog/)
```

You can group individual elements of an expression together in order to support complex matches. Searching for two people.s names could be achieved with two separate tests, like this:

```
if (($string =~ /Martin Brown/) ||
   ($string =~ /Sharon Brown/))
```

```
This could be written as follows

if ($string =~ /(Martin|Sharon) Brown/)
```

Grouping Matching

From a regular-expression point of view, there is no difference between except, perhaps, that the former is slightly clearer.

```
$string =~ /(\S+)\s+(\S+)/;

and

$string =~ /\S+\s+\S+/;
```

However, the benefit of grouping is that it allows us to extract a sequence from a regular expression. Groupings are returned as a list in the order in which they appear in the original. For example, in the following fragment we have pulled out the hours, minutes, and seconds from a string.

```
my ($hours, $minutes, $seconds) = ($time =~ m/(\d+):(\d+):(\d+)/);
```

As well as this direct method, matched groups are also available within the special $x variables, where x is the number of the group within the regular expression. We could therefore rewrite the preceding example as follows:

```
$time =~ m/(\d+):(\d+):(\d+)/;
my ($hours, $minutes, $seconds) = ($1, $2, $3);
```

When groups are used in substitution expressions, the $x syntax can be used in the replacement text. Thus, we could reformat a date string using this:

```
#!/usr/bin/perl

$date = '03/26/1999';
$date =~ s#(\d+)/(\d+)/(\d+)#$3/$1/$2#;

print "$date";

This will produce following result
1999/03/26
```

Using the \G Assertion

The \G assertion allows you to continue searching from the point where the last match occurred.

For example, in the following code we have used \G so that we can search to the correct position and then extract some information, without having to create a more complex, single regular expression:

```
#!/usr/bin/perl

$string = "The time is: 12:31:02 on 4/12/00";
```

```
$string =~ /:\s+/g;
($time) = ($string =~ /\G(\d+:\d+:\d+)/);
$string =~ /.+\s+/g;
($date) = ($string =~ m{\G(\d+/\d+/\d+)});

print "Time: $time, Date: $date\n";

This will produce following result
Time: 12:31:02, Date: 4/12/00
```

The \G assertion is actually just the metasymbol equivalent of the pos function, so between regular expression calls you can continue to use pos, and even modify the value of pos (and therefore \G) by using pos as an lvalue subroutine:

Regular Expression Variables

Regular expression variables include **$**, which contains whatever the last grouping match matched; **$&**, which contains the entire matched string; **$`**, which contains everything before the matched string; and **$'**, which contains everything after the matched string.

The following code demonstrates the result:

```
#!/usr/bin/perl

$string = "The food is in the salad bar";
$string =~ m/foo/;
print "Before: $`\n";
print "Matched: $&\n";
print "After: $'\n";

This code prints the following when executed:
Before: The
Matched: foo
After: d is in the salad bar
```

PERL Subrountines

The two terms function and subroutine are used interchangeably in Perl. A function is a named code block that is generally intended to process specified input values into an output value, although this is not always the case. For example, the print function takes variables and static text and prints the values on the screen.

- Like many languages, Perl provides for user-defined subroutines.
- These may be located anywhere in the main program.
- Loaded in from other files via the do, require, or use keywords.
- Any arguments passed in show up in the array @_.
- A return statement may be used to exit a subroutine.
- If no return is found and if the last statement is an expression, its value is returned.
- If the last statement is a loop control structure like a foreach or a while , the returned value is unspecified.
- Subroutines can return Scalar, Array or Hash.

Subroutines, like variables, can be declared (without defining what they do) or declared and defined. To simply declare a subroutine, you use one of the following forms:

```
sub NAME
```

```
sub NAME PROTO
sub NAME ATTRS
sub NAME PROTO ATTRS
```

where NAME is the name of the subroutine you are creating, PROTO is the prototype for the arguments the subroutine should expect when called, and ATTRS is a list of attributes that the subroutine exhibits.

If you want to declare and define a function, then you need to include the BLOCK that defines its operation:

```
sub NAME BLOCK
sub NAME PROTO BLOCK
sub NAME ATTRS BLOCK
sub NAME PROTO ATTRS BLOCK
```

You can also create anonymous subroutines - subroutines without a name by omitting the NAME component:

```
sub BLOCK
sub PROTO BLOCK
sub ATTRS BLOCK
sub PROTO ATTRS BLOCK
```

To call a function you would use one of the following forms:

```
NAME
NAME LIST
NAME (LIST)
&NAME
```

To give a quick example of a simple subroutine:

```
sub message
{
   print "Hello!\n";
}
```

Function Arguments

The first argument you pass to the subroutine is available within the function as $_[0], the second argument is $_[1], and so on. For example, this simple function adds two numbers and prints the result:

```
sub add
{
   $result = $_[0] + $_[1];
   print "The result was: $result\n";
}
```

To call the subroutine and get a result:

```
add(1,2);
```

The preceding subroutine is fairly simple, but what if you wanted to have named arguments? The simple answer is to assign the values of @_ to a list of variables:

```
sub add
{
   ($numbera, $numberb) = @_;

   $result = $numbera + $numberb;
   print "The result was: $result\n";
}
```

The **shift** function is one of the .stack. operands supported by Perl. The shift function returns (and removes) the first element of an array. For example:

```
sub add
{
   my $numbera = shift;
   my $numberb = shift;

   my $result = $numbera + $numberb;
   print "The result was: $result\n";
}
```

The effect is exactly the same as we have shown earlier but we have just obtained the arguments in a different way.

Return Values from a Function

The return value of any block, including those used in subroutines, is taken as the value of the last evaluated expression. For exampl,e the return value here is the result of the calculation.:

```
sub myfunc
{
   $_[0]+$_[1];
}
```

You can also explicitly return a value using the return keyword:

```
sub myfunc
{
   if (@_)
   {
      return $_[0]+$_[1];
   }
   else
   {
      return 0;
   }
}
```

When called, return immediately terminates the current subroutine and returns the value to the caller. If you don't specify a value then the return value is**undef**.

A Function Call Context

The context of a subroutine or statement is defined as the type of return value that is expected. This allows you to use a single function that returns different values based on what the user is expecting to receive. For

example, the following two calls to the **getpwent** function return a list or a scalar, according to what was used in the assignation:

```
$name = getpwent();
($name, $passwd, $uid, $gid, $quota,
$comment, %gcos, $dir, $shell) = getpwent();
```

In the first case, the user expects a scalar value to be returned by the function, because that is what the return value is being assigned to. In the second case, the user expects an array as the return value, again because a list of scalars has been specified for the information to be inserted into.

Here's another example, again from the built-in Perl functions, that shows the flexibility:

```
my $timestr = localtime(time);
```

In this example, the value of $timestr is now a string made up of the current date and time, for example, Thu Nov 30 15:21:33 2000. Conversely:

```
($sec,$min,$hour,$mday,$mon,
               $year,$wday,$yday,$isdst) = localtime(time);
```

Now the individual variables contain the corresponding values returned by localtime.

Lvalue subroutines

WARNING: Lvalue subroutines are still experimental and the implementation may change in future versions of Perl.

It is possible to return a modifiable value from a subroutine. To do this, you have to declare the subroutine to return an lvalue. See the following example

```
    my $val;
    sub canmod : lvalue {
         # return $val; this doesn't work, don't say "return"
         $val;
    }
    sub nomod {
         $val;
    }
```

Now see the magic

```
    canmod() = 5;   # assigns to $val in the above subroutine
    nomod()  = 5;   # ERROR
```

Passing Lists to Subroutines

Because the @_ variable is an array, it can be used to supply lists to a subroutine. However, because of the way in which Perl accepts and parses lists and arrays, it can be difficult to extract the individual elements from @_. All the followings are valid

```
mysub(1,2,3);
@args = (2,3);
mysub(1,@args);
@args = (1,2,3);
mysub(@args);
```

Finally when we receive thes values in@_ variable then we can not recognize if we had passed one array or two value arraysbecause finally it is getting merged into one.

o

If you want to work with and identify the individual lists passed to Perl, then you need to use references:

```
(@listc, @listd) = simplesort(\@lista, \@listb);
```

The leading \ character tells Perl to supply a reference, or pointer, to the array. A reference is actually just a scalar, so we can identify each list by assigning the reference to each array within our subroutine. Now you can write your subroutineas follows:

```
sub simplesort
{
   my ($listaref, $listbref ) = @_;

   # De-reference the array list
   my (@lista) = @$listaref;
   my (@listb) = @$listbref;
   # Now you can play with both arrays.
}
```

Passing Hashes to Subroutines

When you supply a hash to a subroutine or operator that accepts a list, the hash is automatically translated into a list of key/value pairs. For example:

```
%hash = ('name' => 'Tom', 'age' => 19);
print %hash;
```

This will output .nameTomage19.. However, the same process works in reverse, so we can extract a list and convert it to a hash:

```
sub display_hash
{
   my (%hash) = @_;
   foreach (%hash)
   {
      print "$_ => $hash{$_}\n";
   }
}
```

In this case, we output the key/value pairs of the hash properly, displaying each pair on its own line. As with arrays, care needs to be taken if you expect to pick out a single hash from a list of arguments. The following will work because we extract the hash last:

```
sub display_has_regexp
{
   my ($regex, %hash) = @_;
   ...
}
```

while this one won.t because we try to extract the hash first (there will be an extra element, and Perl won.t know how to assign this to the hash):

```
sub display_has_regexp
{
   my (%hash, $regex) = @_;
   ...
}
```

If you want to work with multiple hashes, then use references. For example, the following subroutine returns the key intersection of two hashes:

```
sub intersection
{
   my ($hasha, $hashb) = @_;
   my %newhash;
   foreach my $key (keys %{$hasha})
   {
      $newhash{$key} = $$hasha{$key} if (exists
$$hashb{$key});
   }
   return %newhash;
}
```

To use the subroutine:

```
%hasha = ('a' => 'b',
          'b' => 'b',
          'c' => 'b');
%hashb = ('b' => 'b',
          'c' => 'b',
          'd' => 'b');
%newhash = intersection(\%hasha, \%hashb);
```

PERL Formats - Writing Reports

As stated earlier that Perl stands for Practical Extraction and Reporting Language, and we'll now discuss using Perl to write reports.

Perl uses a writing template called a 'format' to output reports. To use the format feature of Perl, you must:

- Define a Format
- Pass the data that will be displayed on the format
- Invoke the Format

Define a Format

Following is the syntax to define a Perl format

```
format FormatName =
fieldline
value_one, value_two, value_three
fieldline
value_one, value_two
.
```

`FormatName` represents the name of the format. The fieldline is the specific way the data should be formatted. The values lines represent the values that will be entered into the field line. You end the format with a single period.

`fieldline` can contain any text or fieldholders. Fieldholders hold space for data that will be placed there at a later date. A fieldholder has the format:

```
@<<<<
```

This fieldholder is left-justified, with a field space of 5. You must count the @ sign and the < signs to know the number of spaces in the field. Other field holders include:

```
@>>>> right-justified
@|||| centered
@#####.## numeric field holder
@* multiline field holder
```

An example format would be:

```
format EMPLOYEE =
===================================
@<<<<<<<<<<<<<<<<<<<<<<< @<<
$name $age
@######.##
$salary
===================================
.
```

In this example $name would be written as left justify within 22 character spaces and after that age will be written in two spaces.

Invoke the Format to write Data

In order to invoke this format declaration we would use the write keyword:

```
write EMPLOYEE; #send to the output
```

The problem is that the format name is usually the name of an open file handle, and the write statement will send the output to this file handle. As we want the data sent to the STDOUT, we must associate EMPLOYEE

with the STDOUT filehandle. First, however, we must make sure that that STDOUT is our selected file handle, using the select() function

```
select(STDOUT);
```

We would then associate EMPLOYEE with STDOUT by setting the new format name with STDOUT, using the special variable $~

```
$~ = "EMPLOYEE";
```

When we now do a write(), the data would be sent to STDOUT. Remember: if you didn't have STDOUT set as your default file handle, you could revert back to the original file handle by assigning the return value of select to a scalar value, and using select along with this scalar variable after the special variable is assigned the format name, to be associated with STDOUT.

The above example will generate a report in the following format

```
Kirsten              12
Mohammad             35
Suhi                 15
Namrat               10
```

Defining a Report Header

Everything looks fine. But you would be interested in adding a header to your report. This header will be printed on top of each page. It is very simple to do this. Apart from defining a template you would have to define a header which will have same name but appended with _TOP keyword as follows

```
format EMPLOYEE_TOP =
------------------------
Name                 Age
------------------------
.
```

Now your report will look like

```
------------------------
Name                 Age
------------------------
Kirsten              12
Mohammad             35
Suhi                 15
Namrat               10
```

Defining a Pagination & Number of Lines on a Page

What about if your report is taking more than one page ? You have a solution for that. Use `$%` vairable along with header as follows

```
format EMPLOYEE_TOP =
```

```
   ------------------------
   Name                 Age   Page @<
   ------------------------        $%
   .
```

Now your output will look like

```
   ------------------------
   Name                 Age   Page 1
   ------------------------
   Kirsten              12
   Mohammad             35
   Suhi                 15
   Namrat               10
```

You can set the number of lines per page using special variable `$=` (or $FORMAT_LINES_PER_PAGE) By default $= will be 60

Defining a Report Footer

One final thing is left which is footer. Very similar to header, you can define a footer and it will be written after each page. Here you will use _BOTTOM keyword instead of _TOP.

```
   format EMPLOYEE_BOTTOM =
   End of Page @<
               $%
   .
```

This will give you following result

```
   ------------------------
   Name                 Age   Page 1
   ------------------------
   Kirsten              12
   Mohammad             35
   Suhi                 15
   Namrat               10
   End of Page 1
```

For a complete set of variables related to formating, please refer to Perl Special Variables section.

Error Handling in PERL

You can identify and trap an error in a number of different ways. Its very easy to trap errors in Perl and then handling them properly. Here are few methods which can be used.

Using if

The **if** statement is the obvious choice when you need to check the return value from a statement; for example:

```
if (open(DATA,$file))
{
   ...
}
else
{
   die "Error: Couldn't open the file $!";
}
```

Here variable $! returns the actual error message

Alternatively, we can reduce the statement to one line in situations where it makes sense to do so; for example:

```
die "Error: Something went wrong\n" if (error());
```

Using unless

The **unless** function is the logical opposite to if: statements can completely bypass the success status and only be executed if the expression returns false. For example:

```
unless(chdir("/etc"))
{
   die "Error: Can't change directory!: $!";
}
```

The **unless** statement is best used when you want to raise an error or alternative only if the expression fails. The statement also makes sense when used in a single-line statement:

```
die "Error: Can't change directory!: $!"
unless(chdir("/etc"));
```

Here we die only if the chdir operation fails, and it reads nicely.

Using the Conditional Operator

For very short tests, you can use the conditional operator:

```
print(exists($hash{value}) ? 'There' : 'Missing',"\n");
```

It's not quite so clear here what we are trying to achieve, but the effect is the same as using an if or unless

statement. The conditional operator is best used when you want to quickly return one of two values within an expression or statement.

The Warn Function

The warn function just raises a warning, a message is printed to STDERR, but no further action is taken.

```
chdir('/etc') or warn "Can't change directory";
```

The Die Function

The die function works just like warn, except that it also calls exit. Within a normal script, this function has the effect of immediately terminating execution.

```
chdir('/etc') or die "Can't change directory";
```

Reporting Errors Within Modules

There are two different situations we need to be able to handle:

- Reporting an error in a module that quotes the module's filename and line number - this is useful when debugging a module, or when you specifically want to raise a module-related, rather than script-related, error.
- Reporting an error within a module that quotes the caller's information so that you can debug the line within the script that caused the error. Errors raised in this fashion are useful to the end-user, because they highlight the error in relation to the calling script's origination line.

The warn and die functions work slightly differently than you would expect when called from within a module. For example, the simple module:

```
package T;

require Exporter;
@ISA = qw/Exporter/;
@EXPORT = qw/function/;
use Carp;

sub function
{
   warn "Error in module!";
}
1;
```

when called from a script

```
use T;
function();

It will produce following result
Error in module! at T.pm line 11.
```

This is more or less what you might expect, but not necessarily what you want. From a module

programmer's perspective, the information is useful because it helps to point to a bug within the module itself. For an end-user, the information provided is fairly useless, and for all but the hardened programmer, it completely pointless.

The solution for such problems is the Carp module, which provides a simplified method for reporting errors within modules that return information about the calling script. The Carp module provides four functions: carp, cluck, croak, and confess. These functions are discussed below

The Carp Function

The carp function is the basic equivalent of warn and prints the message to STDERR without actually exiting the script and printing the script name.

```
carp "Error in module!";

This would result in
Error in module! at test.pl line 3
```

The Cluck Function

The cluck function is a sort of supercharged carp, it follows the same basic principle but also prints a stack trace of all the modules that led to the function being called, including information on the original script.

```
cluck "Error in module!";

This would result in something like
Error in module! at T.pm line 11
    T::function() called at S.pm line 13
    S::raise() called at test.pl line 3
```

The Croak Function

The croak function is the equivalent of die, except that it reports the caller one level up. Like die, this function also exits the script after reporting the error to STDERR:

```
croak "Definitely didn't work";

This would result in
Error in module! at S.pm line 13
```

As with carp, the same basic rules apply regarding the including of line and file information according to the warn and die functions.

The Confess Function

The confess function is like cluck; it calls die and then prints a stack trace all the way up to the origination script.

```
confess "Failed around about there";

This would result in
Error in module! at T.pm line 11
   T::function() called at S.pm line 13
   S::raise() called at t2.pl line 3
```

PERL Coding Standard

Each programmer will, of course, have his or her own preferences in regards to formatting, but there are some general guidelines that will make your programs easier to read, understand, and maintain

The most important thing is to run your programs under the -w flag at all times. You may turn it off explicitly for particular portions of code via the no warnings pragma or the $^W variable if you must. You should also always run under use strict or know the reason why not. The use sigtrap and even use diagnostics pragmas may also prove useful.

Regarding aesthetics of code lay out, about the only thing Larry cares strongly about is that the closing curly bracket of a multi-line BLOCK should line up with the keyword that started the construct. Beyond that, he has other preferences that aren't so strong:

- 4-column indent.
- Opening curly on same line as keyword, if possible, otherwise line up.
- Space before the opening curly of a multi-line BLOCK.
- One-line BLOCK may be put on one line, including curlies.
- No space before the semicolon.
- Semicolon omitted in "short" one-line BLOCK.
- Space around most operators.
- Space around a "complex" subscript (inside brackets).
- Blank lines between chunks that do different things.
- Uncuddled elses.
- No space between function name and its opening parenthesis.
- Space after each comma.
- Long lines broken after an operator (except and and or).
- Space after last parenthesis matching on current line.
- Line up corresponding items vertically.
- Omit redundant punctuation as long as clarity doesn't suffer.

Here are some other more substantive style issues to think about:

- Just because you CAN do something a particular way doesn't mean that you SHOULD do it that way. Perl is designed to give you several ways to do anything, so consider picking the most readable one. For instance

```
open(FOO,$foo) || die "Can't open $foo: $!";
```

is better than

```
die "Can't open $foo: $!" unless open(FOO,$foo);
```

because the second way hides the main point of the statement in a modifier. On the other hand

```
print "Starting analysis\n" if $verbose;
```

is better than

```
$verbose && print "Starting analysis\n";
```

because the main point isn't whether the user typed -v or not.

- Don't go through silly contortions to exit a loop at the top or the bottom, when Perl provides the last operator so you can exit in the middle. Just "outdent" it a little to make it more visible:

```
        LINE:
        for (;;) {
            statements;
          last LINE if $foo;
            next LINE if /^#/;
            statements;
        }
```

- Don't be afraid to use loop labels--they're there to enhance readability as well as to allow multilevel loop breaks. See the previous example.
- Avoid using grep() (or map()) or `backticks` in a void context, that is, when you just throw away their return values. Those functions all have return values, so use them. Otherwise use a foreach() loop or the system() function instead.
- For portability, when using features that may not be implemented on every machine, test the construct in an eval to see if it fails. If you know what version or patchlevel a particular feature was implemented, you can test $] ($PERL_VERSION in English) to see if it will be there. The Config module will also let you interrogate values determined by the Configure program when Perl was installed.
- Choose mnemonic identifiers. If you can't remember what mnemonic means, you've got a problem.
- While short identifiers like $gotit are probably ok, use underscores to separate words in longer identifiers. It is generally easier to read $var_names_like_this than $VarNamesLikeThis, especially for non-native speakers of English. It's also a simple rule that works consistently with VAR_NAMES_LIKE_THIS.
  Package names are sometimes an exception to this rule. Perl informally reserves lowercase module names for "pragma" modules like integer and strict. Other modules should begin with a capital letter and use mixed case, but probably without underscores due to limitations in primitive file systems' representations of module names as files that must fit into a few sparse bytes.
- If you have a really hairy regular expression, use the /x modifier and put in some whitespace to make it look a little less like line noise. Don't use slash as a delimiter when your regexp has slashes or backslashes.
- Always check the return codes of system calls. Good error messages should go to STDERR, include which program caused the problem, what the failed system call and arguments were, and (VERY IMPORTANT) should contain the standard system error message for what went wrong. Here's a simple but sufficient example:

```
opendir(D, $dir)     or die "can't opendir $dir: $!";
```

- Think about reusability. Why waste brainpower on a one-shot when you might want to do something like it again? Consider generalizing your code. Consider writing a module or object class. Consider making your code run cleanly with use strict and use warnings (or -w) in effect. Consider giving away your code. Consider changing your whole world view. Consider... oh, never mind.
- Be consistent.
- Be nice.

Object Oriented Programming in PERL

Before we start Object Oriented concept of perl, lets understand references and anonymous arrays and hashes

References

- A reference is, exactly as the name suggests, a reference or pointer to another object.
- There are two types of references: symbolic and hard.
- A symbolic reference enables you to refer to a variable by name, using the value of another variable. For example, if the variable $foo contains the string "bar", the symbolic reference to $foo refers to the variable $bar.
- A hard reference refers to the actual data contained in a data structure.

Creating Hard References

The unary backslash operator is used to create a reference to a named variable or subroutine, for example:

```
$foo = 'Bill';
$fooref = \$foo;
```

The $fooref variable now contains a hard reference to the $foo variable. You can do the same with other variables:

```
$array = \@ARGV;
$hash = \%ENV;
$glob = \*STDOUT;
```

To create a reference to a subroutine:

```
sub foo { print "foo" };
$foosub = \&foo;
```

Anonymous Arrays

When you create a reference to an array directly - that is, without creating an intervening named array - you are creating an anonymous array.

Creating an anonymous array is easy:

```
$array = [ 'Bill', 'Ben, 'Mary' ];
```

This line assigns an array, indicated by the enclosing square brackets instead of the normal parentheses, to the scalar $array. The values on the right side of the assignment make up the array, and the left side contains the reference to this array.

You can create more complex structures by nesting arrays:

```
@arrayarray = ( 1, 2, [1, 2, 3]);
```

The @arrayarray now contains three elements; the third element is a reference to an anonymous array of three elements.

Anonymous Hashes

Anonymous hashes are similarly easy to create, except you use braces instead of square brackets:

```
$hash = { 'Man' => 'Bill',
          'Woman' => 'Mary,
          'Dog' => 'Ben'
};
```

Dereferencing

The most direct way of dereferencing a reference is to prepend the corresponding data type character ($ for scalars, @ for arrays, % for hashes, and & for subroutines) that you are expecting in front of the scalar variable containing the reference. For example, to dereference a scalar reference $foo, you would access the data as $$foo. Other examples are:

```
$array = \@ARGV;     # Create reference to array
$hash = \%ENV;       # Create reference to hash
$glob = \*STDOUT;    # Create reference to typeglob
$foosub = \&foo;     # Create reference to subroutine
push (@$array, "From humans");
$$array[0] = 'Hello'
$$hash{'Hello'} = 'World';
&$foosub;
print $glob "Hello World!\n";
```

Object Basics

There are three main terms, explained from the point of view of how Perl handles objects. The terms are object, class, and method.

- Within Perl, an object is merely a reference to a data type that knows what class it belongs to. The object is stored as a reference in a scalar variable. Because a scalar only contains a reference to the object, the same scalar can hold different objects in different classes.
- A class within Perl is a package that contains the corresponding methods required to create and manipulate objects.
- A method within Perl is a subroutine, defined with the package. The first argument to the method is an object reference or a package name, depending on whether the method affects the current object or the class.

Perl provides a **bless()** function which is used to return a reference and which becomes an object.

Defining a Class

Its very simple to define a class. In Perl, a class is corresponds to a Package.To create a class in Perl, we first build a package. A package is a self-contained unit of user-defined variables and subroutines, which can be re-used over and over again. They provide a separate namespace within a Perl program that keeps subroutines and variables from conflicting with those in other packages.

To declare a class named Person in Perl we do:

```
package Person;
```

The scope of the package definition extends to the end of the file, or until another package keyword is encountered.

Creating and Using Objects

To create an instance of a class (an object) we need an object constructor. This constructor is a method defined within the package. Most programmers choose to name this object constructor method new, but in Perl one can use any name.

One can use any kind of Perl variable as an object in Perl. Most Perl programmers choose either references to arrays or hashes.

Let's create our constructor for our Person class using a Perl hash reference;

When creating an object, you need to supply a constructor. This is a subroutine within a package that returns an object reference. The object reference is created by blessing a reference to the package's class. For example:

```
package Person;
sub new
{
    my $class = shift;
    my $self = {
        _firstName => shift,
        _lastName  => shift,
        _ssn       => shift,
    };
    # Print all the values just for clarification.
    print "First Name is $self->{_firstName}\n";
    print "Last Name is $self->{_lastName}\n";
    print "SSN is $self->{_ssn}\n";
    bless $self, $class;
    return $self;
}
```

Every method of a class passes first argument as class name. So in the above example class name would be "Person". You can try this out by printing value of $class. Next rest of the arguments will be rest of the arguments passed to the method.

Now Let us see how to create an Object

```
$object = new Person( "Mohammad", "Saleem", 23234345);
```

You can use simple hash in your consturctor if you don't want to assign any value to any class variable. For example

```
package Person;
sub new
{
    my $class = shift;
```

```
    my $self = {};
    bless $self, $class;
    return $self;
}
```

Defining Methods

Other object-oriented languages have the concept of security of data to prevent a programmer from changing an object data directly and so provide accessor methods to modify object data. Perl does not have private variables but we can still use the concept of helper functions methods and ask programmers to not mess with our object innards.

Lets define a helper method to get person first name:

```
sub getFirstName {
    return $self->{_firstName};
}
```

Another helper function to set person first name:

```
sub setFirstName {
    my ( $self, $firstName ) = @_;
    $self->{_firstName} = $firstName if defined($firstName);
    return $self->{_firstName};
}
```

Lets have a look into complete example: Keep Person package and helper functions into Person.pm file

```
#!/usr/bin/perl

package Person;

sub new
{
    my $class = shift;
    my $self = {
        _firstName => shift,
        _lastName  => shift,
        _ssn       => shift,
    };
    # Print all the values just for clarification.
    print "First Name is $self->{_firstName}\n";
    print "Last Name is $self->{_lastName}\n";
    print "SSN is $self->{_ssn}\n";
    bless $self, $class;
    return $self;
}
sub setFirstName {
    my ( $self, $firstName ) = @_;
    $self->{_firstName} = $firstName if defined($firstName);
    return $self->{_firstName};
}

sub getFirstName {
    my( $self ) = @_;
    return $self->{_firstName};
```

```
}
1;
```

Now create Person object in mail.pl fileas follows

```
#!/usr/bin/perl

use Person;

$object = new Person( "Mohammad", "Saleem", 23234345);
# Get first name which is set using constructor.
$firstName = $object->getFirstName();

print "Before Setting First Name is : $firstName\n";

# Now Set first name using helper function.
$object->setFirstName( "Mohd." );

# Now get first name set by helper function.
$firstName = $object->getFirstName();
print "Before Setting First Name is : $firstName\n";

This will produce following result
First Name is Mohammad
Last Name is Saleem
SSN is 23234345
Before Setting First Name is : Mohammad
Before Setting First Name is : Mohd.
```

Inheritance

Object-oriented programming sometimes involves inheritance. Inheritance simply means allowing one class called the Child to inherit methods and attributes from another, called the Parent, so you don't have to write the same code again and again. For example, we can have a class Employee which inherits from Person. This is referred to as an "isa" relationship because an employee is a person. Perl has a special variable, @ISA, to help with this. @ISA governs (method) inheritance.

Following are noteable points while using inheritance

- Perl searches the class of the specified object for the specified object.
- Perl searches the classes defined in the object class's @ISA array.
- If no method is found in steps 1 or 2, then Perl uses an AUTOLOAD subroutine, if one is found in the @ISA tree.
- If a matching method still cannot be found, then Perl searches for the method within the UNIVERSAL class (package) that comes as part of the standard Perl library.
- If the method still hasn't been found, then Perl gives up and raises a runtime exception.

So to create a new Employee class that will inherit methods and attributes from our Person class, we simply code: Keep this code into Employee.pm

```
#!/usr/bin/perl

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

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

Now Employee Class has all the methods and attributes inherited from Person class and you can use it as follows: Use main.pl file to test it

```
#!/usr/bin/perl

use Employee;

$object = new Employee( "Mohammad", "Saleem", 23234345);
# Get first name which is set using constructor.
$firstName = $object->getFirstName();

print "Before Setting First Name is : $firstName\n";

# Now Set first name using helper function.
$object->setFirstName( "Mohd." );

# Now get first name set by helper function.
$firstName = $object->getFirstName();
print "After Setting First Name is : $firstName\n";

This will produce following result
First Name is Mohammad
Last Name is Saleem
SSN is 23234345
Before Setting First Name is : Mohammad
Before Setting First Name is : Mohd.
```

Method Overriding

The child class Employee inherits all the methods from parent class Person. But if you would like to override those methods in your child class then you can do it by givig your implementation. You can add your additional functions in child class. It can done as follows: modify Employee.pm file

```
#!/usr/bin/perl

package Employee;
use Person;
use strict;
our @ISA = qw(Person);    # inherits from Person

# Override constructor
sub new {
    my ($class) = @_;

    # Call the constructor of the parent class, Person.
    my $self = $class->SUPER::new( $_[1], $_[2], $_[3] );
    # Add few more attributes
    $self->{_id}   = undef;
    $self->{_title} = undef;
    bless $self, $class;
    return $self;
}

# Override helper function
sub getFirstName {
    my( $self ) = @_;
```

```
    # This is child class function.
    print "This is child class helper function\n";
    return $self->{_firstName};
}

# Add more methods
sub setLastName{
    my ( $self, $lastName ) = @_;
    $self->{_lastName} = $lastName if defined($lastName);
    return $self->{_lastName};
}

sub getLastName {
    my( $self ) = @_;
    return $self->{_lastName};
}

1;
```

Now put following code into main.pl and execute it.

```
#!/usr/bin/perl

use Employee;

$object = new Employee( "Mohammad", "Saleem", 23234345);
# Get first name which is set using constructor.
$firstName = $object->getFirstName();

print "Before Setting First Name is : $firstName\n";

# Now Set first name using helper function.
$object->setFirstName( "Mohd." );

# Now get first name set by helper function.
$firstName = $object->getFirstName();
print "After Setting First Name is : $firstName\n";


This will produce following result
First Name is Mohammad
Last Name is Saleem
SSN is 23234345
This is child class helper function
Before Setting First Name is : Mohammad
This is child class helper function
After Setting First Name is : Mohd.
```

Default Autoloading

Perl offers a feature which you would not find any many other programming languages: a default subroutine.

If you define a function called AUTOLOAD() then any calls to undefined subroutines will call AUTOLOAD() function. The name of the missing subroutine is accessible within this subroutine as $AUTOLOAD. This function is very useful for error handling purpose. Here is an example to implement AUTOLOAD, you can implement this function in your way.

```
sub AUTOLOAD
{
   my $self = shift;
   my $type = ref ($self) || croak "$self is not an object";
   my $field = $AUTOLOAD;
   $field =~ s/.*://;
   unless (exists $self->{$field})
   {
      croak "$field does not exist in object/class $type";
   }
   if (@_)
   {
      return $self->($name) = shift;
   }
   else
   {
      return $self->($name);
   }
}
```

Destructors and Garbage Collection

If you have programmed using objects before, then you will be aware of the need to create a .destructor. to free the memory allocated to the object when you have finished using it. Perl does this automatically for you as soon as the object goes out of scope.

In case you want to implement your destructore which should take care of closing files or doing some extra processing then you need to define a special method called **DESTROY**. This method will be called on the object just before Perl frees the memory allocated to it. In all other respects, the DESTROY method is just like any other, and you can do anything you like with the object in order to close it properly.

A destructor method is simply a member function (subroutine) named DESTROY which will be automatically called

- When the object reference's variable goes out of scope.
- When the object reference's variable is undef-ed
- When the script terminates
- When the perl interpreter terminates

For Example:

```
package MyClass;
...
sub DESTROY
{
    print "   MyClass::DESTROY called\n";
}
```

Another OOP Example

Here is another nice example which will help you to understand Object Oriented Concepts of Perl. Put this source code into any file and execute it.

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

```
# Following is the implementation of simple Class.
package MyClass;

sub new
{
   print "   MyClass::new called\n";
   my $type = shift;            # The package/type name
   my $self = {};               # Reference to empty hash
   return bless $self, $type;
}

sub DESTROY
{
   print "   MyClass::DESTROY called\n";
}

sub MyMethod
{
   print "   MyClass::MyMethod called!\n";
}


# Following is the implemnetation of Inheritance.
package MySubClass;

@ISA = qw( MyClass );

sub new
{
   print "   MySubClass::new called\n";
   my $type = shift;            # The package/type name
   my $self = MyClass->new;     # Reference to empty hash
   return bless $self, $type;
}

sub DESTROY
{
   print "   MySubClass::DESTROY called\n";
}

sub MyMethod
{
   my $self = shift;
   $self->SUPER::MyMethod();
   print "   MySubClass::MyMethod called!\n";
}

# Here is the main program using above classes.
package main;

print "Invoke MyClass method\n";

$myObject = MyClass->new();
$myObject->MyMethod();

print "Invoke MySubClass method\n";

$myObject2 = MySubClass->new();
$myObject2->MyMethod();

print "Create a scoped object\n";
{
```

```
        my $myObject2 = MyClass->new();
}
# Destructor is called automatically here

print "Create and undef an object\n";
$myObject3 = MyClass->new();
undef $myObject3;

print "Fall off the end of the script...\n";
# Remaining destructors are called automatically here
```