PERL

Genesis InSoft Limited

A name you can trust

1-7-1072/A, Opp. Saptagiri Theatre, RTC * Roads, Hyderabad - 500 020

PERL

Genesis Computers (A unit of Genesis InSoft Limited) started its operations on March 16th 1992 in Hyderabad, India primarily as a centre for advanced software education, development and consultancy.

Training is imparted through lectures supplemented with on-line demonstrations using audio visual aids. Seminars, workshops and demonstrations are organized periodically to keep the participants abreast of the latest technologies. Genesis InSoft Ltd. is also involved in software development and consultancy.

We have implemented projects/training in technologies ranging from client server applications to web based. Skilled in PowerBuilder, Windows C SDK, VC++, C++, C, Visual Basic, Java, J2EE, XML, WML, HTML, UML, Java Script, MS.NET (C#, VB.NET and ASP.NET, ADO.NET etc), PHP, Jhoomla, Zend Framework, JQuery, ExtJS, PERL, Python, TCL/TK etc. using Oracle, MySql and SQL Server as backend databases.

Genesis has earned a reputation of being in forefront on Technology and is ranked amongst the top training institutes in Hyderabad city. The highlight of Genesis is that new emerging technologies are absorbed quickly and applied in its areas of operation.

We have on our faculty a team of highly qualified and trained professionals who have worked both in India and abroad. So far we have trained about 48,000+ students who were mostly engineers and experienced computer professionals themselves.

Tapadia (MS, USA), the founder of Genesis Computers, has about 25+ years experience in software industry. He worked for OMC computers, Intergraph India Private Ltd., Intergraph Corporation (USA), and was a consultant to D.E. Shaw India Software Private Ltd and iLabs Limited . He has more than 22 years of teaching experience, and has conducted training for the corporates like ADP, APSRTC, ARM, B.H.E.L, B2B Software Technologies, Cambridge Technology Enterprises Private Limited, CellExchange India Private Limited, Citicorp Overseas Software Limited, CMC Centre (Gachibowli, Posnett Bhavan), CommVault Systems (India) Private Limited, Convergys Information Management (India) Private Limited, D.E. Shaw India Software Private Limited, D.R.D.L, Deloitee Consulting India Private Limited, ELICO Limited, eSymbiosis ITES India Private Limited, Everypath Private Limited, Gold Stone Software, HCL Consulting (Chennai), iLabs Limited, Infotech Enterprises, Intelligroup Asia Private Limited, Intergraph India Private Limited, Invensys Development Centre India Private Limited, Ivy Comptech, JP Systems (India) Limited, Juno Online Services Development Private Limited, Malpani Soft Private Limited, Mars Telecom Systems Private Limited, Mentor Graphics India Private Limited, Motorola India Electronics Limited, NCR Corporation India Private Limited, Netrovert Software Private Limited, Nokia India Private Limited, Optima Software Services, Oracle India Private Limited, Polaris Software Lab Limited, Qualcomm India Private Limited (Chennai, Hyderabad, Bangalore), Quantum Softech Limited, R.C.I, Renaissance Infotech, Satyam Computers, Satyam GE, Satyam Learning Centre, SIS Software (India) Private Limited, Sriven Computers, Teradata - NCR, Vazir Sultan Tobacoo, Verizon, Virtusa India Private Limited, Wings Business Systems Private Limited, Wipro Systems, Xilinx India Technology Services Private Limited, Xilinc, Inc (San Jose).

Genesis InSoft Limited
1-7-1072/A, RTC * Roads, Hyderabad - 500 020
Tel. (040) 4203 0013

rtapadia@genesisinsoft.com rtapadia@barionsolutions.com
www.genesisinsoft.com www.genesisbrains.com www.barionsolutions.com

PERL

PERL (Practical Extraction & Report Language) is an interpreted high-level programming language developed by Larry Wall in 1987.

We write source code in some programming language, which include a set of instructions for the computer to perform some operations dictated by the programmer. There are two ways as to how the source code can be executed by the CPU. The first way is to go through two processes, compilation and linking, to transform the source code into machine code, which is a file consisting of a series of numbers only. This file is in a format that can be recognized by the CPU readily, and does not require any external programs for execution. Syntax errors are detected when the program is being compiled. We describe this executable file as a compiled program. Most software programs (e.g. most EXEs for MS-DOS/Windows) installed in computer fall within this type.

Another way is to leave the program uncompiled (or translate the source code to an intermediate level between machine code and source code, e.g. Java). However, the program cannot be executed on its own. Instead, an external program has to be used to execute the source code. This external program is known as an interpreter, because it acts as an intermediary to interpret the source code in a way the CPU can understand. Compilation is carried out by the interpreter before execution to check for syntax errors and convert the program into certain internal form for execution. Therefore, the main difference between compiled programs and interpreted languages is largely only the time of compilation phase. Compilation of compiled programs is performed early, while for interpreted programs it is usually performed just before the execution phase.

Every approach has its respective merits. Usually, a compiled program has to be compiled once, and thus syntax checking is only performed once. What the operating system only needs to do is to read the compiled program and the instructions encoded can be arranged for execution by the CPU directly. However, interpreted programs usually have to perform syntax check every time the program is executed, and a further compilation step is needed. Therefore, startup time of compiled programs is usually shorter and execution of the program is usually faster. However, there are a number of factors, e.g. optimization, that influence the actual performance. In addition, the end user of a compiled program does not need to have any interpreters installed in order to run the program. This convenience factor is important to some users. On the other hand, interpreters have to be installed in order to execute a program that is interpreted. There are some drawbacks for a compiled program. For example, every time you would like to test software to see if it works properly, you have to compile and link the program. This makes it rather annoying for programmers to fix the errors in the program (debug), although the use of makefiles alleviates most of this hassle. Because compilation translates the source code to machine code that can be executed by the CPU, this process creates a file in machine code that depends on the instruction set of the computer (machine-dependent). On the other hand, interpreted programs are usually platform-independent.

Perl does not create standalone programs and Perl programs have to be executed by a Perl interpreter. Perl interpreters are now available for virtually any operating system, including Microsoft Windows and many flavors of UNIX.

PERL

Perl is a language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. This precise description best summarizes the strength of Perl, mainly because Perl has a powerful set of regular expressions with which programmers can specify search criteria (patterns) precisely

Perl is installed on many Web servers nowadays for dynamic Web Common Gateway Interface (CGI) scripting. Perl programs written as CGI applications are executed on servers where the Perl source files are placed. Discussion forums and many powerful applications for the Web can be developed using Perl.

There is one point that makes Perl very flexible - there is always more than one approach to accomplish a certain task, and programmers can pick whatever approach best suits the purpose. Perl is written in C programming language.

Windows users should download the Activestate distribution of Perl (**http://www.activestate.com/activeperl**). To check if Perl is installed, open a command prompt (on windows) and type **perl -v**. If Perl is installed it would display the version information of Perl on screen.

After you install Perl, make sure the path setting includes the Perl/Bin folder.

Let us look at the steps to create a Perl script.

1) Using any editor create a source file with extension .pl or .plx. Let us call this file hello.pl
2) Add the perl code
3) Execute the perl code by entering the following at the command prompt (-w option enable many useful warnings)

```
Perl –w hello.pl
```

Even if you are using Unix/Linux, it is not absolutely needed to chmod perl source files. In fact, you only need to make those source files executable if you want them to be directly invoked without specifying the name of the interpreter (i.e. Perl). In this case, the shell will look at the first line of the file to determine which interpreter is used, so the #!/usr/bin/perl line must exist. Otherwise, the file cannot be executed. If you only intend to execute the file in UNIX or Linux using **perl -w** filename.pl, then filename.pl need not be given an executable permission. You may have some Perl source files that are not invoked directly. Instead, they are being sourced from another source file and do not need to be executable themselves. For these files, you do not need to chmod them and the default permission is adequate.

Code 1 File: hello.pl

```
#!/usr/bin/perl -w
# print the text to the screen
print "Hello, World!\n";
print "Hello", " ", "World", "\n";
print ("Five plus three: \n", 5+3);
```

Three major types of Perl variables

1. Scalars - single values
2. Arrays - contain lists of values
3. Hashes - "associative arrays – key/value pair of information"

There are others (filehandles, subroutines etc)

A Scalar variable contains a single value.

In Perl, it is not required to declare a variable before it is used. However, before a variable is used in program it should have a value. To assign a value to the variable we use the assignment operator (=). The following are valid.

```
$num = 10;
$a = $b + $c;
$a = $b - 10;
$str = “Genesis”
```

A variable need not be explicitly declared; its “declaration” consists of its first usage. For example, if the statement

```
$x = 5;
```

were the first reference to $x, then this would both declare $x and assign 5 to it.

If you wish to have protection against accidentally using a variable that has not been previously defined, say due to a misspelling, include a line at the top of source code.

```
use strict;
```

Variables in Perl are global by default. To make a variable local to a subroutine or block, the my construct is used.

```
my $val = 3;
```

A Perl script consists of statements (instruction), and each statement is terminated with a semicolon (;). Lines preceded by a # (sharp) sign are comments and are ignored by the perl interpreter. Comments are helpful to help understand what the code does. Line 1 is also a comment as it is of interest to the shell only instead of the perl interpreter itself.

An identifier (variable) should start with a letter (A-Z, a-z) or underscore (_). Subsequent letters may be alphanumeric characters (A-Z, a-z, 0-9) or an underscore (_). No spaces are allowed in the middle of an identifier. Perl is case-sensitive. That means it differentiates lowercase and uppercase characters. The print() function we saw earlier cannot be replaced by Print, PRINT or anything else. Similarly, $var and $Var are two different variables. The last point to note is that identifiers cannot be longer than 255 characters Another point to note is that the name of a scalar, array or hash is formed by a symbol ($ or @ or %) and the identifier. Therefore, $Var, @Var and %Var can coexist. Although they have the same identifier, they are still unique names because the symbols are different. Also, you may use reserved words for the identifier, e.g. $print because the symbol before the identifier tells Perl that this is a variable.

Code 2 File: variables.pl

```perl
$num = 10;
$grade = 'A';
$company = 'Genesis';
$print="value";
$sum = $num + 10;
$prod = $num * 10;

print "Num = $num Grade = $grade company = $company print = $print";
print "\nsum = $sum prod = $prod";
print "\ncompany = $company ", "InSoft";
print "\n", sqrt(25), " ", 5 % 3, " ", 2 ** 4, " ", int(12.34), " ",
abs(-123);

print "\n", $num++;
print "\n", ++$num;
```

Code 3 File: cf.pl

```perl
# A Celsius->Fahrenheit Converter
# Print the prompt
print "Please enter a Celsius degree: ";
# Chop off the trailing newline character
chomp($cel = <STDIN>);
$fah = ($cel * 1.8) + 32;

# print value using variable interpolation
print "The Fahrenheit equivalent of $cel degrees Celsius is $fah\n";
```

Predefined functions:

chomp:	Deletes only new line character at end of string
chop:	Delete and returns the last character of string
index:	Searches for a substring within another string and returns the index position of the substring found
rindex:	Searches a string from the back and returns the index position of the substring found
join:	Merge multiple strings together
lc:	Turn all CAPS to lower case.
lcfirst:	Turn first char to lower case.
length:	Returns the length of the string.
split:	Break string into elements (array)
substr:	Returns a substring of a string substr
uc:	Turn all lower case to CAPS.
ucfirst:	Turn first char to CAPS

The split statement is used for breaking up a scalar value based on a particular character (or characters). The split statement returns what is split as a list (array) of scalar values. This list is normally assigned to an array.

If the outcome of split is used in a scalar context, it automatically sends its output to the `@_` array.

Code 4 File: Split.pl

```
my $passwd = "name=ravi&company=genesis&id=200";
my @fields = split /&/, $passwd;
print "Login name : $fields[0]\n";
print "Company ID : $fields[1]\n";
print "Id : $fields[2]\n\n";

my $i;
foreach $i (@fields) {
   my ($key, $value) = split /=/, $i;
   print "key = $key, value = $value \n";
}

my $data = "Ravi:200:Hyderabad:SE:BE:First class";
my $num = split(/:/, $data, 3);

print "num = $num\n";

for $item (@_) {
   print $i++, " $item\n";
}

my $data = "Ravi::200:Hyderabad:SE:::BE:First class";

$i = 0;
# split using regular expression (more than one occurence of :
split(/:+/ , $data);

for $item (@_) {
   print $i++, " $item\n";
}
```

The join statement is used to combine scalar values (or scalar variables, or elements in an array) into a single scalar value (or variable).

Code 5 File: join.pl

```
@months = qw(Jan Feb Mar Apr May June);
$str1 = join(":", @months);
print $str1, "\n";
$str2 = join(":", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec");
print $str2, "\n";
```

```
$str = join(":", $str1, $str2);
print $str, "\n";
```

The substr statement returns the character(s) in a string when given a position range.

The syntax of the substr() function is as follows:

```
substr STRING, OFFSET
substr STRING, OFFSET, LENGTH
substr STRING, OFFSET, LENGTH, REPLACEMENT
```

Function substr() takes 2-4 parameters. STRING is the string from which extraction is performed. OFFSET is a zero-based offset which indicates the position from which to start extraction. The first character of any string has an OFFSET 0, and 1 for the second character etc. OFFSET can be negative, which counts from the end. For example, the last character of the string can be represented by the OFFSET -1. LENGTH is the number of characters to extract. If it is not specified, it extracts till the end of the string. The extracted substring is returned upon evaluation. If REPLACEMENT is specified, the substring is replaced by the string obtained by evaluating REPLACEMENT, and the substring being replaced is returned. Alternatively substr() can be placed on the left hand side of an assignment operator, and REPLACEMENT on the right.

The only parameter for the length() function is the string itself. It returns the number of characters in the string.

There are several differences between using single quotes and double quotes for strings.

1. The double-quoted string performs variable interpolation on its contents. That is, any variable references inside the quotes will be replaced by the actual values.

2. The single-quoted string will print just like it is. It doesn't honor the dollar sign.

3. The double-quoted string can contain the escape characters like newline, tab, carriage return, etc.

4. The single-quoted string can contain the escape sequences, like single quote, backward slash, etc.

Code 6 File: substr.pl

```
$str = "Genesis InSoft Limited";

print "\n\tcompany: $str";
print "\n", 'company: $str', "\n";

print substr($str, 8), "\n";
print substr($str, 8, 6), "\n";
substr($str, 8, 0, "Barion ");
print $str, "\n";

substr($str, 8, 0) = "Barion ";
```

```
print $str, "\n";
print length($str);

print "\n", "Genesis"."Insoft";
print "\n", "Genesis"x3;

print "\n", "Genesis".123.450;

print "\n", 'Ravi\'s computer crashed';
print "\n", 'Ends with \\ character';
```

The index statement searches for a substring within another string and return the index position of the substring found. You can skip past characters and start the search later in the string. If the substring is not found in the string, index returns -1.

The grep statement looks at each element of an array and return those that match the expression.

The sleep statement is useful for "pausing" the program for a set amount of time (in seconds).

Code 7 File: utilities.pl

```
$str="A Name you can trust";
print index($str, "Name"),"\n";

# Skipping past characters and start the search later in the string
print index($str, "a", 5),"\n";
print index($str, "cant"),"\n";
print rindex($str, "u"),"\n";

@months = qw(Jan Feb Mar Apr May June July);
@match = grep (/^J/i, @months);
print "@match \n";

$val = 1;
for (; $val <= 5; $val++) {
   print "$val \n";
   sleep 1;
}
```

Conditional Expressions

Nonzero values are treated as true, zero as false. Here are the boolean condition and operator symbols:

condition	numbers	strings
=	==	eq
≠	!=	ne
<	<	lt
≤	<=	le
>	>	gt
≥	>=	ge

Note: Braces cannot be omitted in single-statement blocks.

Control flow statements continued

The if statement is used to determine if something is true or false and to take action based upon this outcome.

The unless statement is the logical opposite of the if statement (negation of if). If the outcome of the condition is false, then the unless statements will be executed. If the outcome of the condition is true then the else statements will be executed.

```
unless(condition) {
   unless_statements;
}
else {
   else_statements;
}
```

The while will continue to execute its statement(s) as long as the condition holds true.

```
while ($line = <INFILE>) {
   print $line unless $line eq "\n";
}
```

would print out all the nonblank lines read from the input file.

The until loop is the logical opposite of the while loop. It will continue to execute its statement(s) as long as the condition holds false. The syntax of until loop is:

```
until(condition) {
   statement(s)
}
```

The idea behind a do statement is to execute the statements first and then perform the conditional check.

The for loop is a generic (not specifically for arrays) statement, which has the following syntax:

```
for(initial statement; conditional evaluation; post statement) {
   statement(s);
}
```

The foreach loop is really just a modified form of the for loop. The syntax for the foreach loop is:

```
foreach $var(@array) {
   statement(s);
}
```

Use the last statement to immediately exit from a loop.
The next statement is used to execute the next iteration of the loop.

Code 8   File: condeval.pl

```
print "Please enter num1: ";
$num1 = <STDIN>;
print "Please enter num2: ";
$num2 = <STDIN>;
print "Please enter num3: ";
$num3 = <STDIN>;

chomp($num1, $num2, $num3);

$max = $num1;

if($num1 > $num2)
{
   if($num1 > $num3)
   {
      $max = $num1;
   }
   else
   {
      $max = $num3;
   }
}
elsif($num2 > $num3)
{
   $max = $num2;
   print "\nval = $val";
}
else
{
   $max = $num3;
}

print "Max is $max";
```

Code 9 File: conditions.pl

```
print "Please enter num: ";
chomp($num = <STDIN>);

if($num > 10) {
   print "\nYou entered a value greater than 10";
}

print "\nYou entered a value less than 10" if($num < 10);

unless($num > 10) {
   print "\nunless num > 10";
}
else {
   print "\nelse num > 10";
}

until($num > 10) {
   print "\nUntil Please enter num: ";
   chomp($num = <STDIN>);
}

while($num > 15) {
   print "\nWhile num is $num";
   print "\n Please enter num: ";
   chomp($num = <STDIN>);
   if($num == 20) {
      next;
   }
   print "\nafter next";
   if($num == 15) {
      last;
   }
}

do {
   print "\nDo while: Please enter num: ";
   chomp($num = <STDIN>);
} while($num > 20)
```

Basic Input/Output

File testing is used to determine the "status" of a file. File test conditions:

-r	Returns "true" if the file is readable by the user
-w	Returns "true" if the file is writeable by the user
-x	Returns "true" if the file is executable by the user
-o	Returns "true" if the file is owned by the user
-e	Returns "true" if the file exists

-z Returns "true" if the file exists and is empty
-s Returns the size of the file (0 if the file is empty or doesn't exist)
-f Returns "true" if the file is a "plain file"
-d Returns "true" if the file is a directory
-T Returns "true" if the file contains text data
-B Returns "true" if the file contains binary data

Logical and: To find if two (or more) conditional statements, use the "&&" operator.
Logical or: To find if one of several conditional statements is true, use the "||" operator.
Logical not: To reverse the outcome of a conditional statement, use the "!" operator.

The data is stored in the special variable $_ (default variable). Instead of assigning <STDIN> or <> to a variable, we can just specify the operator.

1. There is no need to create a new variable (a little less typing)
2. Many Perl statements (chomp, chop, print, regular expression pattern matching, etc.) operate on the default variable unless you specify something else.

A filehandle is a connection between the script and a "port". There are four standard filehandles in Perl by default:

<STDIN> Standard input
<> Standard input or files listed on command line
STDOUT Standard output
STDERR Standard error

Code 10 File: io1.pl

```
print "Please enter the name of a file or directory: ";
chomp($name=<STDIN>);

if (-d $name) {
   print "$name is a directory\n";
} else {
   print "$name is not a directory\n";
}

if (-r $name and -T $name) {
   print "$name is readable and contains text data\n";
} else {
   print  "$name  is  either  not  readable  or  does  not  contain  text
data\n";
}

#Print the file size if it is not a directory
if (! -d $name) {
   print "size of file $name is ";
   print -s $name, " bytes\n";
}
```

Arrays and Lists

Arrays are named entities, lists are not. The relationship between a list and an array is very much similar to that between a literal and a scalar variable. A list is merely an ordered set of elements. An array is just like a list but with a name thus can be referenced through an array variable. Array sizes don't have to be declared; Perl takes care of the size of the array.

Each item in the list is called an element. To create a list, simply delimit (separate) the elements with commas (,) and surround the list with a pair of parentheses. For example a list containing the names of some colors can be written as

```
("red", "green", "blue")
```

An array can be created by assigning a list to an array variable. An array variable starts with the symbol @. Therefore, an array can be set up containing the list above, e.g.

```
@colors = ("red", "green", "blue");
```

Alternatively, you may use the equivalent method of per-item assignment:

```
$colors[0] = "red";
$colors[1] = "green";
$colors[2] = "blue";
```

A useful operator here is the range operator `(..)`. If you would like to generate an array of consecutive integers this operator comes in handy, as you no longer have to use a loop. However, the numbers must be in ascending order. If you would like to have an array of consecutive integers in descending order, you may construct it in ascending order using the range operator, and then reverse the position of the items using the reverse() function:

Two arrays can be merged together by this operation:

```
@CombinedArray = (@Array1, @Array2);
```

The resulting array contains all the elements in @Array1, followed by that of @Array2.

```
@MyArray = (@MyArray, $NewElement);
```

The resulting array contains all the elements in @MyArray, followed by a scalar.

We can also append a list of scalar data to the end of an array by using the push() function.

```
push ARRAY, LIST;
```

where ARRAY is the array to which the list data are to be appended. LIST is a list specifying the elements to be appended to ARRAY. A function returns some values (not necessary scalar, can be list data as well for certain functions) after the operation is finished.

unshift is a function that inserts a list at the beginning of an array, and returns the number of elements after the operation.

```
unshift ARRAY, LIST;
```

Perl predefined variables, `$,` is known as the output field separator.

There are two ways in which you could obtain the number of elements stored in an array.

The first method is to employ the concept of context. By evaluating an array in scalar context, we can obtain the number of elements in an array. In the following example, the number of elements in `@colors` is assigned to `$numcolors`.

In Perl, you can find out the subscript of the last item of an array by replacing the symbol @ with $#. For example, the subscript of the last item in @Array is given by $#Array.

Perl provides a facility for users to specify the subscript of the first element of an array. This is specified by assigning an integer to the predefined variable $[. This is 0 by default.

The second method to find number of elements = Last Index - Starting Index + 1
```
= $#Array - $[ + 1
```

An array slice is a subset of elements from all the elements in an array. The subscript operator is not confined to one subscript only. You may specify a list of subscripts using the comma operator (,) and the range operator `(..)`. You use the range operator to specify subscripts in a given range, and the comma operator to specify each subscript individually.

We can use the pop function to remove the last element of an array. It also returns the value of the item being removed.
```
pop ARRAY;
```

On the other hand, the shift function removes the first element of the array, so that the size of the array is reduced by 1 and the element immediately after the item being removed becomes the first element of the array. It also returns the value of the item being removed.
```
shift ARRAY;
```

If ARRAY is empty, undef is returned.

Code 11 File: array1.pl

```
@primarycolors = ("red", "green", "blue");
@othercolors = ("yellow", "black", "white");
@colors = (@primarycolors, @othercolors);

$numcolors = @colors;
print $numcolors, "\n";

$, = " "; # Prints a space in between elements
print @colors, "\n";

@nums = (10 .. 20);
@nums = (@nums, 21, 22, 23);
print @nums, "\n";
```

```
$, = ";";
@rnums = reverse(10 .. 20);
$NumElements = push(@rnums, 9, 8, 7, 6, 5);
print @rnums, "  ", $NumElements, "\n";

@alpha1 = ("a", "b", "c");
@alpha2 = ("d", "e", "f");
unshift @alpha2, @alpha1;
$count = $#alpha2 + 1;   # last element subscript + 1
print @alpha2, "count is ", $count, "\n";
$, = "";
@data = ('a' .. 'z');
@slice = @data[3, 6 .. 10];
print @slice, "\n";

$retval = pop(@slice);
print $retval, "\n";
pop(@slice);
print @slice, "\n";

$retval = shift(@slice);
print $retval, "\n", @slice;
```

There is a generalized function for adding and removing elements from an array. The splice function is so general that it can do what push, pop, shift and unshift does.

```
splice ARRAY, OFFSET [, LENGTH [, LIST]];
```

The following table summarizes how you can use splice in place of other functions.

Function	Equivalent Method
`push(@Array, $x, $y)`	`splice(@Array, @Array, 0, $x, $y)`
`pop(@Array)`	`splice(@Array, -1)`
`shift(@Array)`	`splice(@Array, 0, 1)`
`unshift(@Array, @x)`	`splice(@Array, 0, 0, @x)`
`$Array[$x] = $y`	`splice(@Array, $x, 1, $y)`

`join STRING, LIST`
The join() function concatenates a list of scalars into a single string. It takes a STRING as its first argument which is the separator to be put in between the list elements LIST.

`reverse LIST`
In a list context, the reverse() function returns a list whose elements are identical to that of LIST except the order is reversed.

The map() function iterates over every item in the LIST, sets $ to the item concerned and executes BLOCK or EXPR on each iteration. The return value is a list consisting of the result of evaluation of all iterations. BLOCK is a code block enclosing a sequence of statements to be executed. EXPR can be any expression.

The sort() function can be used to sort a list. By default, the sort() function sorts lexicographically. The items are ordered by comparing the items string wise (using the cmp operator). This comparison is case-sensitive, because it is based on the ASCII values of each character. The sorted list is returned by the sort() function, while the original list remains intact.

In the example below, an array containing the 26 lowercase alphabets is built, and 5 elements starting from the 5th element (i.e. the letter 'e') is converted to uppercase. First @alpha[4 .. 8] contains the 5 letters that are converted to uppercase. The map function calls the uc function (uppercase) for every element in this list, thus converting ("e", "f", "g", "h", "i") to ("E", "F", "G", "H", "I"). We print out each of the names in @countries so that they all start with capital letters while the other characters are in lowercase. This is accomplished by first converting all characters to lowercase by the lc() function, and the first letter is capitalized using the ucfirst() function. This is done for each name in @countries.

Code 12 File: array2.pl

```
@alpha = ('a' .. 'z');
splice @alpha, 4, 5, map(uc, @alpha[4 .. 8]);
$, = ' ';
print @alpha, "\n";

@alpha = ('a' .. 'z');
splice @alpha, 4, 5;
$, = ' ';
print @alpha, "\n";

@countries = ('India', 'USA', 'Malaysia', 'UK');
print join(', ', map {ucfirst(lc($_)); } @countries), "\n";
print sort('India', 'USA', 'Malaysia', 'UK', 'IRAN'), "\n";
```

The srand and rand statements are used to generate random numbers. Since computers can't generate truly random numbers, an algorithm is used to create a random number. This algorithm uses a "seed" (a starting integer number for the algorithm). To set the seed, use the srand statement.

Code 13 File: linearsearch.pl

```
# Linear search of an array. Generating 20 integers
$NUM = 20;
$MAXINT = 500; # 1 to the maximum integer generated

srand(1); # initialize the randomize seed to integer

print "Numbers Generated:\n(";
for $i (1 .. $NUM) {
   push @array, sprintf("%d", rand(1) * $MAXINT);
   print $array[$i-1];
   print ", " unless ($i == $NUM);
}
print ")\n\n";

print "Please enter the number to search for: ";
chomp($toSearch = <STDIN>);

# Linear search here
$counter = 0; $found = 0;
foreach $num (@array) {
   $counter++;
   if ($num == $toSearch) {
      print "\"$toSearch\" found at subscript ", $counter - 1, "\n";
      $found = 1;
      last;
   }
}
if ($found == 0) { print "\"$toSearch\" not found in array.\n"; }
print "Number of comparisons: $counter/", scalar(@array), "\n";
```

Associative array (Hash - %)

Hash is a special kind of data structure. There are several characteristics associated with it. It practically takes very short time to deduce whether any specified data exists. Also, the time it takes does not largely depend on the number of items stored. This is important because hashes are usually used for applications that handle a large amount of data.

An array is simply a contiguous block of memory. In order to support the characteristic stated above, hashes require a slightly more complicated internal structure.

Each item in a hash has a key and a value. The key, which is a string, uniquely identifies an item in the hash. The value is any form of scalar data.

We can assign a list to a hash variable. In this case, the list will be broken up two-by-two, the first one as the key and the second as the value.

If a key already exists, it is assigned the supplied value; otherwise, a new key-value pair is added to the hash.

To delete all key-value pairs in a hash use either method below:

```
%data = ();
or
undef %data;
```

We use the exists() function to check if the key exists in the hash. It returns TRUE if the specified key exists. The defined() function returns true only if the key exists in the hash, and the value is not undefined (i.e. undef).

Code 14 File: hash1.pl

```
$, = " "; # Prints a space in between elements
%data = ('key1', 'value1', 'key2', 'value2');
print $data{'Key1'}, "\n";
$data{'key3'} = 'value3';
print %data, "\n";

$deldata = delete $data{key1};
print $deldata, "\n";

$data{'key1'} = 'newvalue1';

@deldata = delete @data{'key2', 'key3'};

print @deldata, "\n";
print %data, "\n";

undef %data;
print %data, "\n";

%data = (
   key1 => value1,
   key2 => value2
);
print %data, "\n";
%data = ();
print %data, "\n";

%data = qw(k1 v1 k2 v2 k3 v3);
undef $data{'k1'};    # Removes the value, but keeps the key
print "before end", %data, "\n";

if(defined($data{'k1'})) {
   print "\nk1 value exists";
}
else {
   print "\nk1 value does not exist";
}

if(exists($data{'k1'})) {
   print "\nk1 key exists";
}
else {
   print "\nk1 key does not exist";
}
print "\nend";
```

Code 15 File: hash2.pl

```
use strict;
```

```
my ($value, $from, $to, $rate, %rates);
%rates = (
    pounds          => 1,
    dollars         => 1.7,
    rupees          => 82.0,
    "french francs" => 10.0,
    yen             => 170.0,
    euro            => 1.3
);

print "Enter source currency: ";
$from = <STDIN>;
print "Enter target currency: ";
$to = <STDIN>;
print "Enter your amount: ";
$value = <STDIN>;

chomp($from, $to, $value);

exists $rates{$from} || die "I don't know anything about source
currency $from";
exists $rates{$to} || die "I don't know anything about target currency
$to\n";

$rate = $rates{$to} / $rates{$from};

print "$value $from is ",$value*$rate," $to.\n";
```

If the condition defined before the die statement is not met, the script will stop execution at that point, printing out the default error, if a custom error message is not defined.

The output of die will go to the standard error port, STDERR. This will normally be set to the screen, but can be redirected by the user who is running the script.

If a newline character (\n) appears at the end of the die message, just the message is printed on the screen.

If there isn't a newline character at the end of the die message, the message and the line in the script that the die statement appeared on is printed on the screen.

By itself (with no message), die will print "Died" followed by the line number at which the script died.

This Perl sort function uses two special variables $a and $b and they are any two elements from the list that are compared in pairs by sort to determine how to order the list.

Besides this special variables, the Perl sort function uses two operators: cmp (string comparison operator) and <=> (the numerical comparison operator).

cmp returns -1, 0 or 1 depending on whether the left argument is string wise less than, equal to, or greater than the right argument.

<=> returns -1, 0 or 1 depending on whether the left argument is numerically less than, equal to, or greater than the right argument.

Code 16 File: sortarray.pl

```
@countries = qw(India Brazil Argentina Japan USA Korea iran atlanta);

# transform to lowercase
foreach $country (@countries) {
   push(@countries2,  "\L$country");
}

@countries2 = sort(@countries2);
print "@countries2\n";

# Numerical comparison operator
my @array = sort {$a <=> $b} qw(23 101 11 1 102);

print "@array\n";

# String comparison operator
my @array1 = sort {$a <=> $b || $a cmp $b}
            qw(India 32 hyderabad 11 101 Genesis 1 perl 108 training
);

print "@array1\n";
```

A reference is a piece of data that tells us the location of another piece of data. A reference is always a scalar although the data it refers to may not be (it maybe an array or a hash). Main use of reference is in the storage of arrays and hashes (arrays within arrays, hash within hash, arrays within hash and hash within arrays).

Code 17 File: Reference.pl

```
@a1 = (1, 2, 3);
@a2 = (4, 5, 6);
$a1r = \@a1;
$a2r = \@a2;
@a12 = ($a1r, $a2r);
print "a1r $a1r\n";

print "a1 data $a12[0][0], $a12[0][1]\n";
print "@{$a1r} \n";
for (@{$a1r}) {
   print "Element is: $_\n";
}

%h1 = (AP => "Hyderabad", Karnataka => "Bangalore");
```

```
$h1r = \%h1;
print "h1r $h1r\n";

for(keys %{$h1r}) {
   print "Key ", $_, "\t";
   print "Hash ", $h1{$_}, "\t";
   print "Hash Ref ", ${$h1r}{$_}, "\n";
}

@a4 = (10, 20, [30, 40, [50, 60, 65], 70], 80, 90);
print "a4: $a4[0], $a4[2][1], $a4[2][2][1], $a4[4]";
```

Code 17 File: multiarray.pl

```
@id = ("11", "3", "8");
@names = ("ravi", "ram", "ajay");

$id_r = \@id;
$names_r = \@names;
@multi = ($id_r, $names_r);
$, = " ";
print @id, "\n";
print @names, "\n";
print $multi[0][0], " ", $multi[0][1], " ", $multi[1][2], "\n";

# Simple assignment of an array of arrays
@data = (["3", "David"], ["9", "Joe"], ["5", "Pat"]);
print $data[0][0], " ", $data[0][1], " ", $data[2][1], " ", "\n";

my @AoA = ([1, 2, 3], ['MP', 'UP', 'AP'], ['Hyderabad', 20]);

for (my $i = 0; $i <= $#AoA; $i++) {
   print "Data ", $AoA[$i][0], "\n";
}

for (my $x = 0; $x <= 2; $x++) {
   for (my $y = 0; $y <= 2; $y++) {
      $AoA[$x][$y] = $x * $y;
     print "Array [$x][$y] is ", $AoA[$x][$y], "\n";
   }
}
```

Code 18 File: arrayofhash.pl

```
@AoH = ({'key1' => 'value1', 'key2' => 'value2', 'key3' => 'value3'},
      {'newkey1' => 'newvalue1', 'newkey2' => 'newvalue2'});

$AoH[2] = {'key4' => 'val4', 'xkey2' => 'xval2'};

# Adding single key/value pair in hash
$AoH[1]{'NewKey'} = 'NewValue';
```

```perl
# Adding anonymous hash to the array
push(@AoH, {'key3' => 'val3', 'xkey1' => 'xval1'});

print "$#AoH \n";

# Accessing the structure
for (my $i = 0; $i <= $#AoH; $i++)
{
   foreach my $key (keys %{$AoH[$i]})
   {
      print $AoH[$i]{$key}, "\n";
   }
   print "\n";
}
```

Code 19 File: hashofarray.pl

```perl
%HoA = ('Numbers' => [1, 2, 3], 'Names' => ['John', 'Pat', 'Neil']);

$line = "hello world genesis hyderabad";
# Adding an array to the hash
my @tmp = split(' ', $line);
$HoA{'NewKey'} = [@tmp];

# Appending a new element
push(@{$HoA{'NewKey'}}, 'SomeValue');

# Two ways of accessing the structure
print $HoA{'Numbers'}[1];
print "\n";
print $HoA{'Names'}->[1];
print "\n";
print "$HoA{'NewKey'}[2]\t$HoA{'NewKey'}[4]";
```

Code 20 File: hashofhash.pl

```perl
%HoH = ('Hashkey1' => {'Key1'=>'Value1', 'Key2'=>'Value2'},
        'Hashkey2' => {'Key3'=>'Value3', 'Key4'=>'Value4'});

# Adding a key/value pair
$HoH{'Hashkey1'}{'NewKey'} = 'NewValue';

# Adding an anonymous hash to the hash
$HoH{'NewHashKey'} = {'NewKey1'=>'NewValue1', 'NewKey2'=>'NewValue2'};

# Accessing the structure
foreach my $Hashkey (keys %HoH) {
   print "First level: $Hashkey\n";
   foreach my $key (keys %{$HoH{$Hashkey}})
   {
      print "$key => $HoH{$Hashkey}{$key}\n";
   }
}
```

Command Line Arguments

Command-line arguments are stored in the array named @ARGV.

$ARGV[0] contains the first argument, $ARGV[1] contains the second argument, etc.

$#ARGV is the subscript of the last element of the @ARGV array, so the number of arguments on the command line is $#ARGV + 1.

Code 21 File: commandline.pl

```
$numArgs = $#ARGV + 1;
print "Number of command line options are: $numArgs\n";

foreach $argnum (0 .. $#ARGV)
{
   print "$ARGV[$argnum]\n";
}
```

Advanced Input and Output

To open a file to read from, use the open statement.

```
open (HANDLE, "<file_to_open") || die "could not open file";
```

The "<" symbol tells Perl to open the file for reading. This symbol is often omitted as Perl assumes this to be the default behavior.

To open a file to write to, use the open statement:

```
open (HANDLE, ">file_to_open") || die "could not open file";
```

The ">" symbol tells Perl to open the file for writing. If the file already exists, the file contents are overwritten.

To append to the end of the file, use the append symbol: ">>".

Use the close statement to close the filehandle.

The read statement can be used to read a block of a filehandle. The syntax is:

```
read (FILEHANDLE, $var_to_store_read, #_of_bytes_to_read);
```

You can open filehandles that take the output of an OS command and sends it into Perl script. Once again, the open statement creates the filehandle.

```
open (HANDLE, "dir | ");
```

The command "dir" will run the directory listing command on windows. The "|" symbol after the "dir" command tells Perl to run the "dir" command and then send this data into the filehandle. Once the open statement has executed, you can read from it by using the filehandle like <STDIN>.

You can also send output from script into an OS command. For example, suppose you had a large amount of text to display on the screen. You want the user to have the features of the "more" command to control the display of the text.

```
open (MORE, "| more");
```

The "|" symbol before the "more" command tells Perl to send output of the filehandle MORE to the "more" command. Once the open statement has been executed, you can write to it by using the filehandle just like STDOUT.

Note: The "more" command isn't executed until the filehandle is closed. The process of closing the filehandle will close the port and send the data to the OS command:

The special variable `$.` holds the number of the line that is currently being read from the filehandle.

Code 22 File: io2.pl

```
$numArgs = $#ARGV + 1;
if($numArgs < 1)
{
   print "Invalid number of arguments. Atleast 1 argument expected\n";
   exit();
}

open(INFILE, "data.csv");
while ($line = <INFILE>) {
   chomp $line;
   @data = split(",", $line);
   print "@data\n";
}
close(INFILE);

$total = 0;
$filename = "";
while (<>) {
   if (/print/) {
      if($filename ne $ARGV)
      {
         $total = 0;
         print "$ARGV\n";
         $filename = $ARGV;
      }
      $total++;
      print "$total $_";
   }
}
```

Note: Execute the above script as follows:
`perl io2.pl io1.pl io2.pl`

<u>Code 23 File: copy.pl</u>

```
my $source = shift @ARGV;
my $destination = shift @ARGV;

open IN, $source or die "Can't read source file $source: $!\n";
open OUT, ">$destination" or die "Can't write on file $destination:
$!\n";

print "Copying $source to $destination\n";
while (<IN>) {
   print OUT $_;
}

close IN;

open IN, $source or die "Can't read source file $source: $!\n";
my $var = getc IN;
print "character is $var";

read (IN, $data, 10);
print "\nData is $data";

open (PIPE, "dir | ");

while (<PIPE>) {
   print  $_;
}
close PIPE;

open(MORE, "| more") || die "operation failed";
for ($i = 1; $i < 50; $i++) {
   print MORE "$i\n";
}
close MORE
```

readdir DIRHANDLE
Returns the next directory entry for a directory opened by "opendir". If used in list context, returns all the rest of the entries in the directory. If there are no more entries, returns an undefined value in scalar context or a null list in list context.

If you are planning to filetest the return values out of a "readdir", you need to prepend the directory in question. Otherwise, because we didn't "chdir" there, it would have been testing the wrong file (chdir("dirname")).

Code 24 File: dirlisting.pl

```
print "Contents of the current directory:\n";
opendir DL, "." or die "Couldn't open the current directory: $!";
while ($_ = readdir(DL)) {
   next if $_ eq "." or $_ eq "..";
   print $_, " " x (30 - length($_));
   print "d" if -d $_;
   print "r" if -r $_;
   print "w" if -w $_;
   print "x" if -x $_;
   print "o" if -o $_;
   print "\t";
   print -s $_ if -r $_ and -f $_;
   print "\n";
}
```

Code 25 File: io3.pl

```
my $target;
print "Program started: ", scalar localtime, "\n";
while (1) {
   print "Enter file name to write data to?";
   $target = <STDIN>;
   chomp $target;
   if (-d $target) {
      print "No, $target is a directory.\n";
      next;
   }
   if (-e $target) {
      print "File already exists?\n";
      print "(Enter 'r' to write to a different name, ";
      print "'o' to overwrite or 'b' to back up to $target.old)\n";
      my $choice = <STDIN>;
      chomp $choice;
      if ($choice eq "r") {
         next;
      } elsif ($choice eq "o") {
         unless (-o $target) {
            print "No owenership permission $target\n";
            next;
         }
         unless (-w $target) {
            print "No write permission $target $!\n";
            next;
         }
      } elsif ($choice eq "b") {
         if ( rename($target,$target.".old") ) {
            print "OK, moved $target to $target.old\n";
         } else {
            print "Cannot rename file: $!\n";
            next;
         }
      } else {
         print "Unknown input.\n";
         next;
      }
   }
   last if open OUTPUT, "> $target";
}
print OUTFILE "Congratulations.\n";
print "Wrote to file $target\n";
print "Program finished: ", scalar localtime, "\n";
```

Perl provides a method of creating formatted output with the format and write statements. The format statement is used to create a template while the write statement is used to send the output to a file handle.

There are many different types of placeholders that can be used with the format statement. These placeholders tell the write command how to place the contents of the variables to the file handle. The following describes some of the basic placeholders:

Placeholder	Meaning
`@<<<`	Left justify the text
`@>>>`	Right justify the text
`@|||`	Center the text
`@##.##`	Numeric output (lines up decimal place)
`^<<<`	Left justify, break up over multiple line if needed
`@*`	Left justify, multi-line output

Each placeholder character represents one character of the variable, so `@<<<` means "four characters, left justified".

If there aren't enough placeholder characters to "fit" all of the variable's characters, the extra characters are truncated. For example, if the contents of variable is "Genesis" and the placeholder is `@<<`, then only "Gen" would be displayed in the placeholder's "space".

To say "repeat this line over and over until the variable is empty", use the `~~` characters at the beginning of the line:

To create a template, use the following syntax:

```
format FILEHANDLE =
Plain text and placeholder: @>>>>>
$var #variable values go in placeholder
```

Code 26 File: format.pl

```
format STDOUT =
@||||||||||||
$title
Name: @<<<<< Age: @<<
$name, $age
code: @>>>>>>>>
$code
Sale #1: @####.##
$sale1
Sale #2: @####.##
$sale2
company: @<<<
$company
Comment: @*
$comment
location: ^<<<<<<<<<
$location
~~^<<<<<<
$location
.
$title="Mr";
$name="Ravi";
$age=25;
$code="R100";
$sale1=123; $sale2=4.567;
$company="Genesis";
$comment="A name\nyou can\ntrust";
$location="Hyderabad 500020, AP, India";

write STDOUT;
```

Pattern matching

Regular expressions (or patterns) in Perl are very much like wildcard at least conceptually. While wildcards are special characters that refer to file names, regular expressions are special characters that refer to text within a string.

Metacharacters are special characters in a pattern that "represent" other strings. The following characters are metacharacters in Perl.

Char	Meaning
*	Represents the previous character repeated zero or more times
+	Represents the previous character repeated one or more times
{x,y}	Represents the previous character repeated x to y times
.	Represents exactly one character (any one character)
[ ]	Represents any single character listed within the bracket. A ^ character in the beginning changes the meaning to represent any single character NOT listed within the brackets.
?	Represents an optional character. The character prior to the "?" is optional.

^	Represents the beginning of the line when it is the first character in the RE
$	Represents the end of the line when it is the last character in the RE
( )	Used to group an expression.
|	Represents an "or" operator
\	Used to "escape" the special meaning of the above characters.

Refer to following samples:

abc*	"ab" followed by zero or more c's
c*enter	zero or more c's followed by "enter"
abc+	"ab" followed by at least one (or more) c
abc{3,5}	"ab" followed by three to five "c's"
abc{3,}	"ab" followed by three or more "c's"
a**.**c	An "a" followed by any single character followed by a "c"
abc**.**	A "abc" followed by any single character
[a-z]xyz	Any lower case character followed by "xyz"
[A-Z]xyz	Any upper case character followed by "xyz"

[A-Za-z]xyz	Any lower case or upper case character followed by "xyz"
[A-Z][a-z]	A upper case character followed by a lower case character
[^A-Z]xyz	Any non-upper case character followed by "xyz"
[abc^]xyz	Any character is either "a", "b", "c" or "^" followed by "xyz"
abc?	Either "ab" or "abc"
colou?r	Either "color" or "colour"
^abc	"abc" found at the beginning of the string
abc$	"abc" found at the end of the string
^abc$	A line that just contains "abc"
^$	A blank line
(abc)*xyz	"abc" zero or more times followed by xyz
(abc)+xyz	"abc" one or more times followed by xyz
^(abc)+$	A line that contains one or more groups of "abc"
a|bxyz	Either an “a” or "bxyz"
(ab|xyz)123	Either "ab" or "xyz" followed by “123”

Perl has some other built-in regular expressions, often called classes:

Class	Matches
\w	Alphanumeric characters and ([a-zA-Z0-9])
\W	Neither alphanumeric characters nor ([^a-zA-Z0-9])
\s	Whitespace characters ([\t \n \r \f])
\S	Non whitespace characters ([^ \t \n \r \f])
\d	Numeric digits ([0-9])
\D	Non numeric digits ([^0-9])

Grouping (parentheses) can be used to group characters. This is primarily used to have a regular expression affect a string of characters instead of just one.

Grouping can also be used to "back reference" patterns that have been matched. When Perl makes a match of characters within parentheses, what was matched can be referred back to by a designator “\” followed by a numeric value.

Code 27 File: RegularExpression1.pl

```
$str="A Name you can trust";

if($str =~ m/Trust/i) {
   print "found match\n";
}

$str =~ tr/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/;
print $str,"\n";

$str =~ y/[A-Z]/[a-z]/;
print $str,"\n";

$str = $str." you";

$str =~ s/you/we/g;
print $str,"\n";

if($str =~ /NAME/i) {
   print "found name\n";
}

$str = 'Telephone: 040-42030013';
if ($str =~ m/Telephone:\s+(\d{3}-\d{8})$/) {
   print "Telephone number is '$1' \n";
}

$str = 'Phone: (91) 040-42030013';
if ($str =~ m/Phone:\s*(\((\d+)\)\s*(\d{3})-(\d{8}))$/) {
   print "Phone number is '$1'\n";
   print "Country code is '$2'\n";
   print "City code is '$3'\n";
   print "Phone number is '$4'\n";
}
```

When the `e` modifier is used, the right-hand (replacement) side of the substitution is evaluated as a Perl statement. The result of the statement is used as the replacement value.

The d modifier means, "if something is matched and we don't specify what to replace it with, then remove it".

When the s modifier is used with the tr operator. It tells tr to delete duplicated characters that are replaced.

To find the Nth occurrence of a match, we use pattern matching with the g modifier in a while loop.

By default, Perl patterns are "greedy". This means that when matching a pattern, Perl will attempt to "grab" as many characters that will possibly match.

To make patterns Non-Greedy (match the minimal amount), use the "?" after the metacharacter.

You can use the following Non-Greedy patterns:
*?   +?   ??   {n}?   {n,}?   {n,m}?

There are many variables that are set as the result of a pattern match:

Variable	Meaning
$`	String preceding what was last matched
$'	String following what was last matched
$&	Last pattern match
$1..$9	Subpattern matches of last pattern match

Code 28 File: RegularExpression2.pl

```
$var = "Hello";
$code = "Genesis";
$code =~ s/G/chop $var/e;
print $code, "\n";

$phone = "Genesis Phone number is 42030013";
$phone =~ tr/[a-z][0-9]/[A-Z]/d;
print $phone, "\n";

$var = "Tempppple tree";
$var =~ tr/p/P/s;
print $var, "\n";

# To find the Nth occurrence of a match
$data = "Code: A12Z Code: B34Y Code: C56X Code: D78W";
while ($data =~ /(Code: [A-Z][0-9]{2}[A-Z])/g) {
   $count++;
   print "The $count match is $1\n";
}

# greedy match ("grab" as many characters that will possibly match)
$line="What is the best time to call you, time now is 10:30 AM";
$line =~ s/the.*time/tested/;
print $line, "\n";
```

```
# Non greedy match ("grab" as few that will possibly match)
$line="What is the best time to call you, time now is 10:30 AM";
$line =~ s/the.*?time/tested/;
print $line, "\n";

$line="What is the best time to call you, time now is 10:30 AM";
$line =~ s/the.+?time/tested/;
print $line, "\n";

# replace with $`, $', $+ to see the results.
$data = "Ge12nesis3";
$data =~ m/[0-9]/;
print "The number was $&\n";
```

We have seen before assertions like ^ and $. Assertions are used to match certain conditions within a string (such as beginning and end of a line).

The default behavior of Perl is to "ignore" new line characters when it comes to matching the end of a string. You can look for a newline character if you want to.

With the s modifier, Perl treats newlines just like normal characters. This means that the "." metacharacter will match a newline character.

While you can place comments before and after regular expressions, sometimes it would be nice to place comments within regular expressions to help explain what the expression does. With the x modifier you can place comments and whitespace within regular expressions. When the x modifier is used, comments (# to end of line) and whitespace (Tabs, spaces, newlines, etc.) are completely ignored. This means if you want to "look for" one of these characters, you need to escape them with a backslash.

Code 29 File: RegularExpression3.pl

```
print "Please enter data:\n";
@words = split (/\s+/, <STDIN>);
print "First word: $words[0]\n";
print "Last word: $words[$#words]\n";
print "Number of words ", $#words+1, "\n";

$_="This is a good day\nto learn Perl";

if(/day.to/) {
   print "found\n";
}
if(/day.to/s) {
   print "found1 $_\n";
}

# commenting regular expression

$_='Area: 040 - \PIN=500020\ ';
```

```
m/
(Area:)     # Look for "Area:"
(\ \d{3})   # match and group " " followed by three numbers
\ - \       # match " - "
\\PIN=      # match "\PIN="
(\d+)       # match and group one or more digits
/x;

print "First match : $1\n";
print "Second match: $2\n";
print "third match: $3\n";
```

Subroutine

A complex program can be broken into smaller tasks, each of which carries out a well-defined function.

Declaring the subroutine makes Perl aware that a subroutine of a particular name exists. Defining means you explicitly describe what the subroutine does by listing the statements to be executed if the subroutine is being called.

In general, subroutine declaration and definition go together. To create a function, use the sub statement.

```
sub name [(prototype)] block
```

To invoke a function, specify the ampersand character (&) followed by the function name.

All functions return a value to the calling program. Although this return value is normally a scalar variable, it can also be an array or an associative array.

Two methods can be used to return a value to the calling program: the explicit method and the more cryptic implied method.

With the explicit method, use the return statement to specify what value to return to the calling program.

If you don't specify what to return with the return statement, Perl will return the outcome of the last statement in the function.

When you pass a parameter into a function, you actually pass a reference. In a sense, the elements in the @_ array share the same memory space as the variables that were passed into the function from the main program. Changing elements in the @_ array will also change the variable that are being passed in.

To avoid this, you should reassign elements in @_ to other variables (either scalar or another array). When variables are assigned, they are assigned by value by default, not by reference.

Code 30 File: subroutine1.pl

```
# read two numbers from the command line
$x = $ARGV[0];
$y = $ARGV[1];

# call subroutine which finds the minimum and prints the value
$z = &min($x, $y);
print $z, "\n";

$total = sum(0 .. 10);
print $total, "\n";

$add = add($x, $y);
print $add, "\n";

$prod = mult($x, $y);
print $prod, "\n";

sub min
{
   if ($_[0] < $_[1]) {
      return $_[0];
   }
   else {
      return $_[1];
   }
}

sub sum(@)
{
   my $sum = 0;
   for my $tmp (@_) {
      $sum += $tmp;
   }

   # implicit method. Returns the value of $sum without any return call
   $sum;
}

sub add
{
   $n1 = shift @_; # get first argument
   $n2 = shift @_; # get second argument
   return $n1 + $n2;
}

sub mult
{
   ($n1, $n2) = @_;
   return $n1 * $n2;
}

sub numSort {
  if ($a < $b) { return -1; }
  elsif ($a == $b) { return 0;}
  elsif ($a > $b) { return 1; }
}

# invoke the Perl sort function with the subroutine
my @array = sort numSort qw(23 101 11 1 102);

print "@array\n";
```

The prototype specification comprises a sequence of symbols indicating the type of each argument. The symbols are used to denote scalar variables, arrays, hashes etc. In front of each symbol you may prepend a backslash to indicate the element is to be passed by reference.

Symbol	Type
$	Scalar variable
@	Array
%	Hash
&	Anonymous subroutine
*	Typeglob (Reference to Symbol Table entry)

For example, if the prototype is ($$), that means the subroutine accepts two scalar variables as parameters. ($$@) implies the first two parameters to be evaluated in scalar context while the remaining parameters would be an array variable.

Note: You cannot have something like (@$ or %S) as the array or hash variable would take up all the input parameters. Always bear in mind that multiple parameters, after evaluating in their respective contexts, are combined together to become one indistinguishable array @_.

The local statement can be used to protect the calling program's variables from being modified from the function.

The my statement is like the local statement in that it protects the calling program from having its variables modified by the function. However, my also protects the function from having its variable changed by another function.

Code 31 File: subroutine2.pl

```
sub sum {
   my(@temp)=@_; # replace my with local and test
   $temp[0]++;
   foreach $num (@temp) {
      $total += $num;
   }
   &modify;
   print "temp in sum is @temp\n";
   return ($total);
}

sub modify {
   @temp=(20, 30);
   print "temp in modify is @temp\n";
}

# main program
@temp=(5, 10);
print &sum(@temp), "\n";
print "temp in main @temp \n";
```

Modules

Perl modules (libraries) are files that contain reusable code. These libraries can either be created by you, built-in to Perl, or downloaded.

Typically, modules declare generic functions that can be used within script. To make use of these functions, use the use statement to tell Perl to "import" the functions into script.

Difference between use and require:

use :
1. The method is used only for the modules (only to include .pm type file).
2. The included objects are verified at the time of compilation.
3. No need to give file extension.

require:
Require is a call to an external essential script or module, without which the current script/program will not proceed any further. The require function is used to load a separate perl script or module at run time. If the file to be loaded is not in the directories listed in @INC array then full path of the perl file is to be passed in the require function.

```
require "require.pl";
require "Mymodule.pm";
```

Once the perl file is loaded INC variable is updated with latest loaded perl file as key and its location as value. That's why before loading perl file require function checks if the file is already there in INC. If file is there then it will not load the file.

1. The method is used for both libraries and modules.
2. The included objects are verified at the run time.
3. Need to give file Extension.

When you split code into multiple files, Perl provides a nice mechanism to ensure that variables in different parts of program do not clash. The mechanism is to divide program into different namespaces. The idea is very simple - each namespace has a label which uniquely identifies the namespace and we prepend to variable names the label so that we can differentiate in case two variables in two namespaces happen to have the same name. C++ uses the notion of namespace, while in Perl a namespace is called a package instead.

Any variables not explicitly contained in any packages belong to the main package. Therefore, all variables we have been using in fact belong to the main package. By declaring additional packages we create shields so that variables in different packages would not interfere with each other.

To declare the start of a package, put

```
package package name;
```

Usually package declarations are placed at the beginning of source files to ensure that all variables in the file are protected. If a package declaration is placed inside a code block, the package extends to the end of the code block.

my is a way to declare:

non-package variables, that are
private,
new,
non-global variables,
separate from any package. So that the variable cannot be accessed in the form of $package_name::variable.

On the other hand, our variables are:

package variables, and thus automatically
global variables,
definitely not private,
nor are they necessarily new; and they
can be accessed outside the package (or lexical scope) with the qualified namespace, as $package_name::variable.

Code 32 File: module1.pm

```
package module1;
use strict;

use Exporter;
use vars qw($VERSION @ISA @EXPORT @EXPORT_OK);

$VERSION     = 1.00;
@ISA         = qw(Exporter);
```

```
@EXPORT      = qw(reverseList converttoUC);
@EXPORT_OK   = qw();

my $value = 20;

sub reverseList{
   return reverse @_;
}

sub converttoUC
{
   return map{uc}@_;
}

1; # must include dummy return value as last line
```

Code 33 File: module1test.pl

```
use strict;
BEGIN {
   print " Before @INC \n";
   push @INC, 'E://PERL//Apps//module1';
}
print "After @INC \n";

my @list = qw (1 3 5 8 9 2);

#use module1 qw(&reverselist &converttoUC);

use module1;
$,= " ";
print reverseList(@list),"\n";
print converttoUC(@list),"\n";

print $module1::value;
```

Advanced Subroutine handling

Passing reference to a subroutine allows us to modify the original value when we dereference it. We can use prototypes to take a reference behind the scenes. If in a prototype, instead of a dollar sign, we give a type symbol followed by a backslash, Perl will automatically take a reference to that type of variable. Hence sub add(\$) looks for a single scalar variable and take a reference to it. Sub add($\%$) looks for a scalar, a hash, and a scalar and will take a reference to the hash.

The prototype of a subroutine can take a subroutine, we can pass arrays, hashes etc.

Code 34 File: subroutine3.pl

```
use strict;

sub funcref
{
   my $reference = shift;
   $$reference++;
}

sub funcref2(\$)
{
   my $reference2 = shift;
   $$reference2++;
}

my $var = 5;
funcref(\$var);
print "$var\n";

funcref2($var);
print $var;
```

In the following example, the prototype takes references to two arrays. What @_ contains are two array references. We then check if the array sizes are same. If they are, we compare each element of the array and stop when there is a difference.

Code 35 File: subroutine4.pl

```
use strict;

sub is_same(\@\@);

my @arr1 = (2..5);
my @arr2 = (2..5);
my @arr3 = (2..6);

print "Arr1 is same as Arr2 \n" if is_same(@arr1, @arr2);
print "Arr1 is same as Arr3" if is_same(@arr2, @arr3);

sub is_same(\@\@)
{
  my ($a1, $a2) = @_;

  my $len1 = @$a1;
  my $len2 = @$a2;
  print "Len1 = $len1 Len2 = $len2 \n";
  print "a1 = @$a1 a2 = @$a2\n";

  return 0 unless $len1 == $len2;
  for my $elem (0..$#$a1)
  {
    return 0 unless $a1->[$elem] eq $a2->[$elem];
  }
  return 1;
}
```

You can pass filehandles to a subroutine by either passing a glob or a reference to a glob (it does not make any difference). You can collect the filehandle into a glob or place the file handle into a scalar and use that in place of a filehandle.

Code 36 File: subroutine5.pl

```
use strict;

sub display
{
   *target = shift;
   print target "Writing using glob: Hello Perl";
}

open(OUTFILE, ">sample.txt");
display(*OUTFILE);
close(OUTFILE);

sub getdata
{
   my $source = shift;
   my $readline = <$source>;
   chomp $readline;
   return $readline;
}

open(INFILE, "sample.txt");

my $line = getdata(*INFILE);
print "Data is: $line\n";

sub display2
{
   my $target = shift;
   print $target "Writing to stdout";
}

display2(*STDOUT);
```

We can give the parameters for subroutines a default value, that is, give the parameter a value to run through the subroutine with if one is not specified when the subroutine is called. This is done with the `||` operator (logical or).

The logical or operator returns the last thing it sees. If we say `$var = 2 || 4`, then $var will be set to 2, as it is a true value, it has no need to examine anything else, and so 2 is the last thing it saw. If however, we say `$var = 0 || 4`, then $var will be set to 4; 0 is not true, so it looks at the next one, 4, which is the last thing it sees.
Hence, if we get from @_ a value, that is not true, we can give a default with the `||` operator. This would enable us to create a subroutine with variable number of parameters.

Code 37 File: subroutine6.pl

```
sub add
{
   $val = shift || 2;
   $val2 = shift || 3;
   $val3 = shift || 4;
   return $val + $val2 + $val3;
}

$result = add(1, 2, 3);
print "$result \n";

$result = add(5, 6);
print "$result \n";

$result = add();
print $result;
```

In all the subroutines written so far, we had to pass arguments as positional arguments. However we can pass parameters as named arguments, as the parameters can be thought of as a hash.

We can declare a reference to a function (like reference to a variable) by putting a backslash before the name, but include the ampersand.

There are two ways to call subroutine reference. You can call directly as shown below:

```
&{$ref};
&{$ref}(@arguments);
```

or by using the arrow notation:

```
$ref->();
$ref->(@arguments);
```

Code 38 File: subroutine7.pl

```
use warnings;
use strict;

sub formdata
{
   die "Arguments should be even" if @_ % 2;
   my %args = @_;
   print "Name: $args{name}\t";
   print "Location: $args{location}\t";
   print "Phone: $args{phone}\n";
}

formdata(name => "genesis", location => "Hyderabad", phone => "040-
42030013");
formdata(location => "Hyderabad", name => "Barion", phone => "040-
42030013");

sub display
{
   print "Reference to function\n";
}
my $ref = \&display;

&{$ref};
$ref->();
```

File System and Process Control

Perl provides several built-in statements that allow you to control the file system (files and directories). With these statements you can perform the following:

- Change Directories
- List Files
- Make Directories
- Remove Directories
- Remove Files
- Rename Files
- Change Permissions
- Get information regarding a file

You can execute Operating System (OS) commands from within the script. Most OS provide commands that allow you to modify the file system. These commands, such as mkdir and ls, are the methods the OS user uses to manipulate files and directories.

Avoid running OS commands as it can result in several "problems":

- Perl script may run slower. The reason for this is that Perl must spawn another process to run the OS command.
- Script may become "platform dependant" as the OS command that you attempt to run may not be available on another Operating System.
- Script may become "user dependant". A user can modify his/her environment (such as aliases, functions and the PATH variable) to alter the way that OS commands execute. If they do, this may cause problems when you run these OS commands from within Perl script.

Code 39 File: cwd.pl

```perl
# standard Cwd module tells you where you are
# chdir changes directory. opendir/readdir to parse directory

use Cwd;

$now = cwd;
print "We are in $now\n";

# chdir changes directory
$tryfor = $ARGV[0] || "Module1";

if (-d $tryfor) {
   chdir ($tryfor);
   $now = cwd;
   print "We are now in $now\n";
   print "Directory entry count: ", dirsize($now), "\n";

   chdir ("..");
   $now = cwd;
   print "We are now in $now\n";
   print "Directory entry count: ", dirsize($now), "\n";
} else {
   print "No subdirectory $tryfor\n";
}

# Report on entries in directory
sub dirsize {
   if (opendir DH,$_[0]) {
      @stuff = readdir DH;
      return scalar(@stuff);
   } else {
      return -1;
   }
}
```

The stat statement provides an array of useful information regarding a file or directory. According to the Perl documentation, the following describes the fields that the stat statement returns:

```perl
($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size,
 $atime, $mtime, $ctime, $blksize, $blocks) = stat($filepath);
```

Index	Value returned	
0	dev	device number of filesystem
1	ino	inode number
2	mode	file mode (type and permissions)
3	nlink	number of (hard) links to the file
4	uid	numeric user ID of file's owner

5	gid	numeric group ID of file's owner
6	rdev	the device identifier (special files only)
7	size	total size of file, in bytes
8	atime	last access time in seconds since the epoch (00:00 January 1, 1970 GMT)
9	mtime	last modify time in seconds since the epoch
10	ctime	inode change time in seconds since the epoch
11	blksize	preferred block size for file system I/O
12	blocks	actual number of blocks allocated

By using back quotes, we can run an OS command and return the value to a scalar variable.

The system statement sends the output of the OS command to the screen/STDOUT. This is good for tasks like clearing the screen.

Code 40 File: filesystem.pl

```
mkdir ("test");

@info = stat("io1.pl");
for $i (0..12) {
   print "$i\t$info[$i]\n";
}

my $timestamp = $info[8];
my $time       = localtime($timestamp);
print "\n Time is ", $time, "\n\n";

$dir = `dir io*.pl`;
print $dir, "\n";

print "Enter your name: ";
chomp($name = <STDIN>);
system "cls";
print "Hello $name\n";

rmdir("test");
```

Pragmas

Pragmas are specific instructions that can be embedded in Perl code, depending on our needs and preferences, that allow our scripts to be compiled and behave differently than they would otherwise.

integer	Forces integer math instead of floating point or double precision math.
strict	Restricts unsafe constructs

The most common way to invoke a pragma in a Perl script is via the use statement, that allows us to use modules. Pragmas are separate Perl modules that follow the same rules and constructs of their more common counterparts; specifically, they have their own package namespaces, implement (if necessary) their own package variables, and define their own import routines, resulting in the script being somehow altered or enhanced by the pragma's executed code.

The most important similarity between a regularly used module and a pragma is that pragmas are also loaded at compile-time, i.e., they are loaded and executed as the Perl script is being compiled.

```
use strict;
```

To disable the use of a pragma, use the no statement:

```
no strict;
```

Some of the important pragmas:

autouse	Delays the operation of a require statement until one of the specified subroutines is called.
diagnostics	Issues verbose error messages.
integer	Performs integer arithmetic instead of double.
lib	Modifies the @INC variable at compile time.
strict	Prevents unwise statements
subs	Allows you to predeclare subroutines
vars	Allows you to predeclare global variables.
warnings	Outputs messages for common - but syntactically valid scripting errors

use strict ‘subs’ pragma, creates an error message for strings without quotes around them that appear to be subroutine calls, that don't call a valid subroutine.

use strict 'vars', pragma will generate an error if a variable is used that:

- has not been declared as a my variable or
- isn't a fully qualified variable name or
- has not been declared as an our variable or
- has not been declared with a use vars statement

To be able to use a variable prior to having it set, you can use the statement "use vars". Once invoked, you cannot use "no vars" to undo a "use vars" statement.

Use "use subs" to "predefine" subroutines.

The argument to the "use lib" statement will be pre-appended to the @INC variable. Once a module has been loaded, the original location of the module is stored in the %INC associative array.

Use diagnostics is very useful when debugging programs.

Code 41 File: pragma1.pl

```
{
   my($name) = "Genesis";
   print("$name\n");
}

#use strict;
print("$name\n");

use strict 'vars';
our $val;
sub test {
   print "$val\n";
}
$val = 100;
&test;

use subs qw(hello);
use strict 'subs';
&hello;
sub hello {
   print "hello\n";
}

hello;
#func;
```

Code 42 File: pragma2.pl

```
print("Floating point division: ", 10/3, "\n");

use integer;
print("Integer division: ", 10/3, "\n");

no integer;
print("Floating point division: ", 10/3, "\n");

# The $] variable holds the numeric version of the running interpreter
print $], "\n";
if ($] < 5.008) {
   use utf8;
}

print "\n";
foreach $key (keys %INC) {
   print "$key ->> $INC{$key}\n";
}

print "\n";
use Cwd;
print cwd;

print "\n\n$ENV{PATH} \n";
use Env;
print $PATH, "\n";
```

```
# prints OS
print "\n", "$^O", "\n";
```

Perl style and coding standards

```
perldoc perlstyle
or
man -M /usr/local/man perlstyle
```

The most important thing is to run programs under the -w flag at all times. You should also always run under use strict or know the reason why not.

Regarding aesthetics of code layout, about the only thing Larry cares strongly about is that the closing curly brace 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:

1) 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";
```

Similarly, just because an operator lets you assume default arguments doesn't mean that you have to make use of the defaults. If you want program to be readable, consider supplying the argument.

Along the same lines, just because you CAN omit parentheses in many places doesn't mean that you ought to:

```
    return print reverse sort num values %array;
    return print(reverse(sort num (values(%array))));
```

When in doubt, parenthesize, to ensure that someone else who is maintaining code doesn’t put parentheses in the wrong place.

2) Don't write complex code 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;
    }
```

3) Don't be afraid to use loop labels: they are there to enhance readability as well as to allow multilevel loop breaks.

4) 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.
5) 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.

6) While short identifiers like $found are probably ok, use underscores to separate words. It is generally easier to read $var_names_like_this than $VarNamesLikeThis.

   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.

7) You may find it helpful to use letter case to indicate the scope or nature of a variable. For example:

$ALL_CAPS_HERE	constants only (beware clashes with perl vars!)
$Some_Caps_Here	package-wide global/static
$no_caps_here	function scope my() or local() variables

Function and method names seem to work best as all lowercase. E.g., $obj->as_string().

You can use a leading underscore to indicate that a variable or function should not be used outside the package that defined it.

8) If you have a complex regular expression, use the /x modifier and put in whitespace to make it easier to read.

9) Use the new ``and'' and ``or'' operators to avoid having to parenthesize list operators so much, and to reduce the incidence of punctuation operators like && and ||. Call subroutines as if they were functions or list operators to avoid excessive ampersands and parentheses.

10) Use here documents instead of repeated print() statements.

11) Line up corresponding things vertically, especially if it'd be too long to fit on one line anyway.

```
$IDX = $ST_MTIME;
$IDX = $ST_ATIME       if $opt_u;
$IDX = $ST_CTIME       if $opt_c;
$IDX = $ST_SIZE        if $opt_s;

mkdir $tmpdir, 0700 or die "can't mkdir $tmpdir: $!";
chdir($tmpdir)      or die "can't chdir $tmpdir: $!";
mkdir 'tmp',   0777 or die "can't mkdir $tmpdir/tmp: $!";
```

12) 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: $!";
```

13) Line up transliterations when it makes sense:

```
tr [abc]
   [xyz];
```

14) Think about reusability. Consider generalizing code. Consider writing a module or object class. Consider making code run cleanly with use strict and -w in effect. Consider giving away code.

15) Be consistent.

Command Line Options

Perl has a large number of command-line options that can help to make programs more concise and open up many new possibilities for one-off command-line scripts using Perl.

To list the options, type
```
perl -help
```

The first of these is -c. This option compiles program without running it. This is a great way to ensure that you haven't introduced any syntax errors while you've been editing a program.

```
$ perl -c <program>
```

This makes sure that the program still compiles

The next safety net is the -w option. This turns on warnings that Perl will then give you if it finds any of a number of problems in code. Each of these warnings is a potential bug in program and should be investigated. In modern versions of Perl (since 5.6.0) the -w option has been replaced by the use warnings pragma, which is more flexible than the command-line option.

```
$ perl -w <program>
```

You can mix multiple command line options as shown below:

```
$ perl -c -w <program>
```

The final safety net is the -T option. This option puts Perl into "taint mode." In this mode, Perl inherently distrusts any data that it receives from outside the program's source - for example, data passed in on the command line, read from a file, or taken from CGI parameters.
One more important option is –d, which puts you into the Perl debugger.

Using the command line approach, you can test simple perl programs. The -e option allows the user to enter the Perl statements on the command line:

For example on windows you can use the following to print Hello Perl on screen.

```
perl -e "print 'Hello Perl\n';"
```

Note: On windows use double quotes around the print, single around the text. On UNIX, The single quotes around the Perl statement are needed to "protect" special characters from the shell and the double quotes are needed around the text that will be printed by the print statement.

You can have as many -e options as you like and they will run in the order that they appear on the command line.

```
perl -e "print 'Hello Perl\n'; -e print 'World\n';"
```

Get count of words in an input file.
If you need to have processing carried out either before or after the main code loop, you can use a BEGIN or END block. Here's a pretty basic way to count the words in a text file:

```
perl -ne "END { print $t } @w = /(\w+)/g; $t += @w" hello.pl
```

Another way to get count using split function.

```
perl -pne "END { print $x } @w = split; $x += @w" hello.pl
```

But there are a couple of command-line options that will make that even simpler. Firstly the -a option turns on autosplit mode. In this mode, each input record is split and the resulting list of elements is stored in an array called @F. This means that we can write our above word-count program like this:

```
perl -ane "END {print $x} $x += @F" hello.pl
```

Debugging Perl

The -w switch (option) is to look for and report unusual code. This code typically includes a logical (not syntax) error and includes the following:

- Variable names that are mentioned only once
- Scalar variables are set before use
- Redefined subroutines
- References to undefined file handles
- References to file handles opened read-only that the script is attempting to write to
- Values used as a number that don't look like numbers
- Arrays used in scalar context
- Subroutines that "recurse" more than 100 deep

Perl provides a built-in debugger that can be invoked when running Perl with the -d option.

```
perl -d join.pl
Loading DB routines from perl5db.pl version 1.32 Editor support
available.

Enter h or `h h' for help, or `perldoc perldebug' for more help.

main::(join.pl:1):      @months = qw(Jan Feb Mar Apr May June);

DB<1>
```

Notes about the debugger:
Perl must first be able to compile the code prior to entering the debugger
main::(join.pl:1) means "Main part of script join.pl, line #1"
At this point, no statements have been executed.
The command above the prompt (DB<1>) is the next command to be execute.

The debugger has many built-in commands. The most common are:

Command	Meaning
!! cmd	Runs the command (cmd) in a separate process (this is typically a shell command)
h	Interactive help
H -num	Prints last "num" commands (excludes single character commands)
l	Lists the next line of code to be executed
n	Step through a statement (if subroutines are called, executes over the subroutine)
q	Quits the debugger
s	Step through a statement (if subroutines are called, executes one subroutine statement at a time)
V	Display all of the variables in package (defaults to main)
w	Lists a window, a few lines of code above and below next statement

Here are some of the debugger operations:

h	Interactive help
b k	set a breakpoint at line/subroutine k
b k c	set a breakpoint at line/subroutine k, with boolean condition c
B k	delete a breakpoint at line/subroutine k
n go	to next line, skipping over subroutine calls
s go	to next line, not skipping over subroutine calls
c	continue until breakpoint
c k	continue until line k/subroutine (one-time breakpoint)
L	list all breakpoints/actions
d k	delete breakpoint at line k
a k c	execute Perl command ("action") c each time hit breakpoint at k
a k	delete action at line k
r	finish this subroutine without single-stepping
p y	print y
x y	examine y (nicer printing of y)
T	stack trace
l	list a few lines of source code
!	re-do last command
H	list all recent commands
!n	re-do command n
R	attempt a restart, retaining breakpoints etc.
q	quit
=	set a variable to a specified value

The “DB <1>” is the prompt, showing that this is the first command that we’ve given.

For an example of the use of = command, the following would set the variable $z to 5:
DB <1> $z = 5
The a command is very useful. E.g.
```
a 10 print "$x, $y\n"
```
would result in printing out $x and $y every time you hit the breakpoint at line 10.

Debugging tools

The Carp module can be used to generate error messages. The module provides functions that act similar to Perl's warn and die commands.

carp function produces error messages similar to warn.
croak function acts similar to the die statement.

The built-in warn statement prints error messages to STDERR. It also displays the line number where the error occurred.

The carp and croak commands works in the same manner in cases in which it is called within the main part of the program. If it is called within a subroutine, it also provides the original line from where the subroutine was called

The built-in die statement prints error messages to STDERR and exits the script. It also displays the line number where the error occurred.

Code 43 File: debug1.pl

```
use Carp;

sub warning3 {
   warn "This is warn with a newline char\n";
   warn "This is what warn look like";
   carp "This is what carp looks like";
}

sub warning2 {
   &warning3;
}

sub warning1 {
   &warning2;
}

&warning1;

print "Will be printed";
```

Code 44 File: debug2.pl

```
use Carp;

sub dieusage {
   die "This is what die looks like";
}

sub croakusage3 {
   croak "This is what croak looks like";
}

sub croakusage2 {
   &croakusage3;
}
sub croakusage1 {
   &croakusage2;
}

#&dieusage;
&croakusage1;

print "Will not be printed";
```

In Perl debugger we use the x command to print data in a "nicer" format.

```
  DB<1> @countries = qw(India USA Japan)
  DB<2> x @countries
0  'India'
1  'USA'
2  'Japan'
```

When you use the x command, it uses the Data::Dumper module to format the output. You can use this module to print data from within script.

Code 45 File: debug3.pl

```
use Data::Dumper;

@countries = qw(India USA Japan);
print Dumper(@countries);
print Dumper(\@countries);

%data = ('k1', 'v1', 'k2', 'v2', 'k3', 'v3');

print Dumper(%data);
print Dumper(\%data);
print Dumper(\%ENV);
```

Note: The "\" before %data is used to make a reference to the %data hash. Similarly the "\" before countries is used to make a reference to the @countries array.