5.1 What is Perl

Perl is the **Practical Extraction and Report Language** and is freely available for downloading from the Comprehensive Perl Archive Network (www.perl.com/CPAN/). 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. The current version of Perl was **Perl 5.20**.

- Perl was created by **Larry Wall**.
- Perl is a stable, cross platform programming language.
- Perl is a portable, command line driven, interpreted programming/scripting language.
- It is used for mission critical projects in the public and private sectors.
- Perl is ***Open Source software***, licensed under the *GNU General Public License(GPL).*

A Perl program (or script) consists of a sequence of commands and the source code file can be named arbitrarily but usually uses the ***.pl*** suffix. A Perl interpreter reads the source file and executes the commands in the order given. You may use any text editor to create Perl scripts. These scripts will work on any platform where the Perl interpreter has been installed.

The Perl scripting language is usually used in the following applications areas:

- Web CGI programming
- DOS and UNIX shell command scripts
- Text input parsing
- Report generation
- Text file transformations, conversions
- System administration
- Network programming
- GUI development

Perl Features

Perl has many of the features. Some of them are

Perl is free

Perl's source code is open and free. Anybody can download the C source that constitutes a Perl interpreter. We can easily extend the core functionality of Perl by modifying the Perl source code.

Perl is simple to learn, concise and easy to read

Perl has the syntax similar to C and shell script but with less restrictive format. Most programs are quicker to write in Perl because of its use of built-in functions and a huge standard and contributed library. Perl can be easy to read because the code can be written in a clear and concise format that almost like English sentences.

Perl is fast

Perl is not an interpreter in the strictest sense – when we execute a Perl program it is actually compiled into a highly optimized language before it is executed. Compared to most scripting languages, this makes execution almost as fast as compiled C code. But because the code is still interpreted, there is no compilation process and applications can be written and edited much faster than with other languages without any performance problems.

Perl is extensible

We can write Perl based packages and modules that extend the functionality of the language. We can also call external C code directly from Perl to extend the functionality further.

Perl has flexible data types

We can create simple variables that contain text and numbers. We can also handle arrays of values as simple lists. We can create associative arrays as hashes. Perl also has the feature of references which allows us to create a complex data structure.

Perl is object oriented

Perl supports all of the object oriented features – inheritance, polymorphism, and encapsulation. There is no boundary as there is with C and C++.

Perl is collaborative

Most programmers supply and use the modules and scripts via **CPAN** (Comprehensive Perl Archive Network). This is a repository of the best modules and scripts available.

5.2 The Perl History

Perl was created in the UNIX tradition of open source software. Perl 1.0 was released 18 December 1987 (the Perl birthday) by Larry Hall with the following description:

"Perl is an interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). It combines some of the best features of C, sed, awk, and sh. Expression syntax corresponds quite closely to C expression syntax".

Version	Date	Details
Perl 0		Introduced Perl to Larry Wall's office associates
Perl 1	Jan 1988	Introduced Perl to the world
Perl 2	Jun 1988	Introduced Harry Spencer's regular expression package
Perl 3.0	Oct 1989	Introduced the ability to handle binary data
Perl 4	Mar 1991	Introduced the first "Camel" book (*Programming Perl*, by Larry Wall, Tom Christiansen, and Randal L Schwartz; O'Reilly & Associates). The book drove the name change, just so it could refer to Perl 4, instead of Perl 3.0
Perl 4.036	Feb 1993	The last stable release of Perl 4
Perl 5.0	Oct 1994	A complete rewrite of Perl adding objects and a modular organization. The modular structure makes it easy for everyone to develop Perl modules to extend the functionalities of Perl.
Perl 5.005_02	Aug 1998	The next major stable release.
Perl 5.005_03	Mar 1999	The last stable release before 5.6
Perl 5.6	Mar 2000	Introduced unified **fork** support, better threading, an updated Perl compiler, and the **our** keyword.

5.3 Perl Programming ABC

To create a Perl program, you may simply use your favorite text editor. The very first line, before any other characters in the source code file, indicates the command to invoke the Perl interpreter. For example,

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

It indicates the location of the Perl interpreter which will execute the rest of the file. The Perl program file must be executable. On UNIX do:

```
chmod a+rx program name
```

On Windows, run Perl programs from the MS-DOS prompt. To run a Perl program use either one of

```
perl –w program name arg1 arg2 ...
program name arg1 arg2 ...
```

As a Web CGI program, a Perl script must be placed in special *cgi-bin* directories configured by the Web server.

In a Perl script:

- Comments start with the # character and continue to the end of the line.
- Each Perl statement ends with a semicolon (;).
- Statements are executed sequentially.
- The statement exit(0); (exit(1);) terminates the program normally (abnormally).

Here is a short Perl program (**Ex: cmdLine.pl**) that simply displays the command-line arguments:

```
#!/usr/bin/perl
print "@ARGV\n";            ## displays all the command line arguments
print "First arg: $ARGV[0]\n";          ## display first argument
print "Second arg: $ARGV[1]\n";         ## display second argument
print "Third arg: $ARGV[2]\n";          ## display third argument
```

On a UNIX system, enter this program into the file ***cmdLine.pl*** and do ***"chmod a+rx cmdLine.pl"*** to make it executable.

Then issue the command ***./cmdline.pl a b c d e*** to run the program which is in the current directory (./). You should see the display

```
a b c d e
First arg: a
Second arg: b
Third arg: c
```

You can run Perl programs similarly under MS-DOS.

```
C:/CSE540>perl –w cmdLine.pl a b c d e
```

5.4 Perl Variables

Perl provides three types of variables:

1. Scalar
2. Array (list), and
3. Association array (hash)

5.4.1 Scalars

A scalar variable has a ***'$'*** prefix and can take on any string or numerical values. A Scalar variable always holds one value at a time.

For example,

```
$var = 'a string';              ## a quoted string
$n = length $var;               ## is 8
$x = 12;
$abc = "$var$x";                ## a string12
```

Characters enclosed in single quotes are taken literally while variables are meaningful inside double quotes.

5.4.2 Arrays (Lists)

Array is a one-dimensional list of scalars. Perl uses the **"@"** prefix and parentheses with respect to the name of an array as a whole, whereas individual elements within an array are referred to as scalars and the index is placed in square brackets.

Functions associating with arrays:

push ( ): appends a new element to the end of the array

Syntax:

push (array, value)

pop ( ): The pop function to remove the last element from the array

Syntax:

pop (array)

unshift ( ): The function *unshift* adds an element at the beginning of an array.

Syntax:

unshift (array, value)

shift ( ): The function shift deletes an element at the beginning of an array.

Syntax:

shift (array)

For example

```
#!/usr/bin/perl
@arr = ("aa", "bb", "cc", "dd");         ## creating an array
print "$arr[0]\n";                       ## first array element is aa
$arr[2]=7;                               ## third element set to 7
$m = $#arr;                              ## 3, last index of @arr
$n = @arr;                               ## n is 4 length of @arr
print "@arr\n";                          ## aa bb 7 dd
push(@arr, "xyz");                       ## put on end of array
print "@arr\n";                          ## aa bb 7 dd xyz
$last = pop(@arr);                       ## pop off end of array
```

```
print "@arr\n";                    ## aa bb 7 dd
```

Executing this program produces the following output

```
aa
aa bb 7 dd
aa bb 7 dd xyz
aa bb 7 dd
```

We use the scalar notation to retrieve or set values on an array using indexing. The special prefix $# returns the index of the last array element and -1 if the array has no elements. Hence, $#arr+1 is the array length. Assigning an array to a scalar produces its length. Displaying the entire array is as easy as printing it.

5.4.3 Association Arrays (Hashes)

An association array, also known as a hash array, is an array with even number of elements. Elements come in pairs, a key and a value. Perl association array variables use the % prefix.

You can create an hash arrays with the notation:

```
( key1 => value1, key2 => value2, ... )
```

The keys serve as symbolic indices for the corresponding values on the association array.

For example:

```
%asso = ( "a" => 7, "b" => 11 );    ## creating a hash
print "$asso{'a'}\n";               ## displays 7
print "$asso{'b'}\n";               ## displays 11
print "@asso{'a', 'b'}\n";          ## displays 7, 11
```

The symbol => makes the association perfectly clear.

To retrieve a value from an association array, use its key. Note the $ prefix is used with the key enclosed in curly braces ({}). To obtain a list of values from an association list, the @ prefix can be used. Use a non-existent key or a value as a key get an undefined value (*undef*).

Assign a new value with a similar assignment where the key may or may not already be on the association array:

```
$asso{'c'} = 13;
```

To remove key-value pairs from a hash, use calls like:

```
delete( $asso{'c'} ); (deletes one pair)
delete( @asso{'a', 'c'} ); (deletes a list of pairs)
```

The keys function produces an array of keys of a given hash:

```
@all_keys = keys( %asso )           ## ('a', 'b', 'c')
```

The values function produces an array of values of a given hash:

```
@all_values = values ( %asso )      ## (7, 11, 13)
```

5.5.4 True or False

In Perl Boolean values are scalar values interpreted in the Boolean context.

False values:

- The numerical zero(0)
- The string zero(“0”)
- The Empty string(“”), and
- undefined value are Boolean false.

True values:

All other scalar values are Boolean true. Also any non-zero number and any non-empty string are considered true.

5.5 Automatic Data Context

Perl makes programming easier by detecting the context within which a variable is used and automatically converts its value appropriately. For example you can use strings as numbers and vice versa.

```
$str1 = "12.5";
$str2 = "2.5";
$sum = $str1 + $str2;       ## adding as numbers (3)
```

```
        print "$sum\n";                ## displaying (4)
```

We used the strings as numbers on line 3 and the $sum, which is a number, as a string on line 4.

5.6 Perl Operators

Perl language supports many operator types but following is a list of most frequently used operators:

- **Arithmetic Operators**
- **Relational Operators**
- **Logical Operators**
- **Assignment Operators**
- **Bitwise Operators**
- **Miscellaneous Operators**

Perl Arithmetic Operators

Assume variable $a holds 10 and variable $b holds 20 then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	$a + $b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	$a - $b will give -10
*	Multiplication - Multiplies values on either side of the operator	$a * $b will give 200
/	Division - Divides left hand operand by right hand operand	$b / $a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	$b % $a will give 0
**	Exponent – Performs exponential (power) calculation on operators	$a**$b will give 10 to the power 20

Perl Relational Operators

Assume variable $a holds 10 and variable $b holds 20

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	($a == $b) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	($a != $b) is true.
<=>	Checks if the value of two operands are equal or not, and returns -1, 0, or 1 depending on whether the left argument is numerically less than, equal to, or greater than the right argument.	($a <=> $b) returns -1.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	($a > $b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	($a < $b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	($a >= $b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	($a <= $b) is true.

Perl Assignment Operators

Assume variable $a holds 10 and variable $b holds 20 then:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	$c = $a + $b will assign value of $a + $b into $c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result	$c += $a is equivalent to $c =$c + $a

	to left operand	
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	$c -= $a is equivalent to $c = $c - $a
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	$c *= $a is equivalent to $c = $c * $a
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	$c /= $a is equivalent to $c = $c / $a
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	$c %= $a is equivalent to $c = $c % a
**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand	$c **= $a is equivalent to $c = $c ** $a

Perl Bitwise Operators

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	($a & $b) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in each other operand.	($a | $b) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	($a ^ $b) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~$a) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	$a << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	$a >> 2 will give 15 which is 0000 1111

Perl Logical Operators

Assume variable $a holds true and variable $b holds false then:

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true then then condition becomes true.	($a and $b) is false.
&&	C-style Logical AND operator copies a bit to the result if it exists in both operands.	($a && $b) is false.
or	Called Logical OR Operator. If any of the two operands are non-zero then then condition becomes true.	($a or $b) is true.
||	C-style Logical OR operator copies a bit if it exists in either operand.	($a || $b) is true.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	not($a and $b) is true.

Miscellaneous Operators

Operator	Description	Example
.	Binary operator dot (.) concatenates two strings.	If $a="abc", $b="def" then $a.$b will give "abcdef"
x	The repetition operator x returns a string consisting of the left operand repeated the number of times specified by the right operand.	('-' x 3) will give xxx.

..	The range operator .. returns a list of values counting (up by ones) from the left value to the right value	(2..5) will give (2, 3, 4, 5)
++	Auto Increment operator increases integer value by one	$a++ will give 11
--	Auto Decrement operator decreases integer value by one	$a-- will give 9
->	The arrow operator is mostly used in dereferencing a method or variable from an object or a class name	$obj->$a is an example to access variable $a from object $obj.s

5.7 Perl Conditional and Control Structures

Perl conditional statements helps in decision making which require the programmer specifies one or more conditions to be tested by the program, along with statements to be executed if the condition is determined to be true, and other statements to be executed if the condition is determined to be false.

Perl programming language provides following types of conditional statements.

5.7.1 Conditional Structures

Statement	**Description**
if statement	An **if statement** consists of a Boolean expression followed by one or more statements.
if...else statement	An **if statement** can be followed by an optional **else statement**.
if...elsif...else statement	An **if statement** can be followed by an optional **elsif statement** and then by an optional **else statement**.
unless statement	An **unless statement** consists of a Boolean expression followed by one or more statements.
unless...else statement	An **unless statement** can be followed by an optional **else statement**.
unless...elsif..else statement	An **unless statement** can be followed by an optional **elsif statement** and then by an optional **else statement**.
switch statement	With latest versions of Perl, We can make use of **switch** statement which allows a simple way of comparing a variable value against various conditions.

if – else statement

A Perl **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

Syntax:

The syntax of an **if...else** statement in Perl programming language is:

```
if(boolean_expression) {
        # statement(s) will execute if the given condition is true
}
else {
        # statement(s) will execute if the given condition is false
}
```

If the Boolean expression evaluates to **true** then **if block** of code will be executed otherwise **else block** of code will be executed.

Example:

```
#!/usr/local/bin/perl
$a = 100;
# check the boolean condition using if statement
if( $a < 20 ) {
        printf "a is less than 20\n";
}
else {
        printf "a is greater than 20\n";
}
print "value of a is : $a\n";
```

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

```
a is greater than 20
value of a is : 100
```

if .. elsif Statement

An **if** statement can be followed by an optional **elsif...else** statement, which is very useful to test multiple conditions using single if...elsif statement.

Syntax:

The syntax of an **if...elsif...else** statement in Perl programming language is:

```
if(boolean_expression 1) {
        # Executes when the boolean expression 1 is true
}
elsif( boolean_expression 2) {
        # Executes when the boolean expression 2 is true
}
elsif( boolean_expression 3) {
        # Executes when the boolean expression 3 is true
}
else {
        # Executes when the none of the above condition is true
}
```

Example:

```
#!/usr/local/bin/perl
$a = 100;
# check the boolean condition using if statement
if( $a == 20 ) {
        printf "a has a value which is 20\n";
}
elsif( $a == 30 ) {
        printf "a has a value which is 30\n";
}
else {
        printf "a has a value which is $a\n";
}
```

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

```
a has a value which is 100
```

unless Statement

A Perl **unless** statement consists of a Boolean expression followed by one or more statements.

Syntax:

The syntax of unless statement in Perl programming language is:

```
unless(boolean_expression)
{
        # statement(s) will execute if the given condition is false
}
```

If the Boolean expression evaluates to **false** then the block of code inside the *unless* statement will be executed. If Boolean expression evaluates to **true** then the first set of code after the end of the *unless* statement (after the closing curly brace) will be executed.

Example:

```
#!/usr/local/bin/perl
$a = 20;
# check the boolean condition using unless statement
unless( $a < 20 ) {
        # if condition is false then print the following
        printf "a is not less than 20\n";
}
print "value of a is : $a\n";
```

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

```
a is not less than 20
value of a is : 20
```

switch Statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

Syntax:

The synopsis for a **switch** statement in Perl programming language is as follows:

```
switch(argument) {
        case 1          {       print "number 1"  }
        case "a"        {       print "string a"     }
        case [1..10]    {       print "number in list"        }
        case (\@array)  {       print "number in list"        }
        case (\%hash)   {       print "entry in hash"         }
        else            {       print "previous case not true"  }
}
```

The following rules apply to a **switch** statement:

- The **switch** statement takes a single scalar argument of any type, specified in parentheses.
- A **switch** statement can have an optional **else** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is matched.
- If a case block executes an untargeted **next**, control is immediately transferred to the statement after the case statement (i.e. usually another case), rather than out of the surrounding switch block.

Example:

```
#!/usr/local/bin/perl
use Switch;
$var = 10;
@array = (10, 20, 30);
%hash = ('key1' => 10, 'key2' => 20);
switch($var){
case 10         { print "number 100\n"; next; }
case "a"        { print "string a" }
case [1..10]    { print "number in list" }
case (\@array)  { print "number in list" }
case (\%hash)   { print "entry in hash" }
else                    { print "previous all cases not true" }
}
```

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

```
number 100
number in list
```

5.7.2 Control Structures

A loop statement allows us to execute a statement or group of statements multiple times. Perl programming language provides following four types of loops to handle looping requirements.

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

while loop

A **while** loop statement in Perl programming language repeatedly executes a target statement as long as a given condition is true.

Syntax:

The syntax of a **while** loop in Perl programming language is:

```
while(condition)
{
        statement(s);
}
```

Example:

```
#!/usr/local/bin/perl
$a = 10;
# while loop execution
while( $a < 15 ) {
        printf "Value of a: $a\n";
        $a++;
}
```

When executed, above code produces following result:

```
Value of a: 10
Value of a: 11
Value of a: 12
Value of a: 13
Value of a: 14
```

until loop

An **until** loop statement in Perl programming language repeatedly executes a target statement as long as a given condition is false.

Syntax:

The syntax of a **until** loop in Perl programming language is:

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

Example:

```
#!/usr/local/bin/perl
$a = 5;
# until loop execution
until( $a > 10 ) {
        printf "Value of a: $a\n";
        $a = $a + 1;
}
```

When executed, above code produces following result:

```
Value of a: 5
Value of a: 6
Value of a: 7
Value of a: 8
Value of a: 9
Value of a: 10
```

for loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

The syntax of a **for** loop in Perl programming language is:

```
for ( initialization; condition; increment or decrement )
{
        statement(s);
}
```

Example:

```
#!/usr/local/bin/perl
# for loop execution
for( $a = 10; $a < 15; $a = $a + 1 ) {
        print "value of a: $a\n";
```

```
    }
```

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

```
    value of a: 10
    value of a: 11
    value of a: 12
    value of a: 13
    value of a: 14
```

The **foreach** loop iterates over a list value and sets the control variable (var) to be each element of the list in turn:

Syntax:

The syntax of a **foreach** loop in Perl programming language is:

```
    foreach var (list) {
        statements;
    }
```

Example:

```
    #!/usr/local/bin/perl
    @list = (2, 20, 30);
    # foreach loop execution
    foreach $a (@list) {
        print "value of a: $a\n";
    }
```

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

```
    value of a: 2
    value of a: 20
    value of a: 30
```

5.8 Perl Functions/Subroutines

A Perl subroutine or function is a group of statements that together perform a task. We can divide up the code into separate subroutines. Perl uses the terms subroutine, method and function interchangeably.

- You define functions with the *sub* keyword.
- A function can be placed anywhere in your Perl source code file. Usually all functions are placed at the end of the file.
- For substantial programming, functions and objects can be placed in separate *packages* or *modules* and then imported into a program with the *use* statement

Define and Call a Function

The general form of a function is:

```
    sub functionName
    {
        body of the functions
    }
```

A call to the above may take any of these forms:

```
    functionName();              ## no arg
    functionName($a);            ## one arg
    functionName($aa, $bb);      ## two args
    functionName(@arr);          ## array elements as args
    functionName($aa, @arr);     ## $aa and array elements as args
```

Example

```
    #!/usr/bin/perl
    # Function definition
    sub Hello {
        print "Hello, World!\n";
    }

    # Function call
```

```
Hello();
```

When above program is executed, it produces following result:

```
Hello, World!
```

Passing Arguments to a Function

We can pass various arguments to a subroutine and they can be accessed inside the function using the special array **@_ .** Thus the first argument to the function is in **$_[0],** the second is in **$_[1],** and so on.

Example

```
#!/usr/bin/perl
# Function definition
sub Sum
{
        $sum = 0;
        my ($a,$b,$c) = @-;
        $sum = $a+$b+$c;
        print "Sum of the given numbers : $sum\n";
}
# Function call
Sum(10, 20, 30);
```

When above program is executed, it produces following result:

```
Sum of the given numbers : 60
```

Returning Value from a Function

We can return a value from subroutine using return statement. If we are not returning a value from a subroutine then last calculation result in a subroutine is the return value. We can return arrays and hashes from the subroutine like any scalar values.

Example

```
#!/usr/bin/perl
# Function definition
sub Average
{
        # get total number of arguments passed.
        $n = scalar(@_);
        $sum = 0;
        foreach $item (@_)
        {
                $sum += $item;
        }
        $average = $sum / $n;
        return $average;
}
# Function call
$num = Average(10, 20, 30);
print "Average for the given numbers : $num\n";
```

When above program is executed, it produces following result:

```
Average for the given numbers : 20
```

5.8.1 Local Variables in Function

By default, all variables in Perl are global variables which means they can be accessed from anywhere in the program. But we can create **private** variables called **lexical variables** at any time with ***my*** or ***local*** keywords.

A variable declared by **my** is known only within the function or source code file, in the same sense as local variables in C, C++, or Java.

my (var1, var2, ...);

A variable declared by **local** is known within the function and other function it calls at run-time,

local (var1, var2, ...);

Example

```
#!/usr/bin/perl
$string = "Hello, World!";          # Global variable
print "$string\n";
printHello();                       # Function call
print "$string\n";

sub printHello                      # Function definition
{
        my $string;                 # Local variable for printHello function
        $string = "Hello, Function!";
        print "$string\n";
}
```

When above program is executed, it produces following result:

```
Hello, World!
Hello, Function!
Hello, World!
```

5.8.2 Temporary Values via local()

The **local** is mostly used when the current value of a variable must be visible to called subroutines. A local just gives temporary values to global (meaning package) variables. This is known as ***dynamic scoping***.

Example

```
#!/usr/bin/perl
$string = "Hello, World!";                  # Global variable
printHello();                               # Function call
print "$string\n";

sub printHello                              # Function definition
{
        local $string;                      # Private variable for printHello function
        $string = "Hello, Function!";
        printMe();                          # Function call
        print "$string\n";
}

sub printMe                                 # Function definition
{
        print "$string\n";
}
```

When above program is executed, it produces following result:

```
Hello, Function!
Hello, Function!
Hello, World!
```

5.8.3 our

The **our** keyword (introduced in Perl 5.6) declares a variable to be global, effectively making it the complete opposite of **my**. For example,

```
our $str = "Hello World";
print "$str\n";
printHello();
print "$str\n";

sub printHello
{
        our $str = "Hello, Function";
        print "$str\n";
}
```

produces

```
Hello World
Hello Function
Hello Function
```

5.9 Perl I/O

5.9.1 Standard I/O

In Perl, the file handles **STDIN, STDOUT,** and **STDERR** stand for the standard input (from keyboard), standard output (to screen), and standard error output (to screen, no buffering) respectively. The notation ***<input-source>*** is handy to read lines from an input source.

For example,

```
$line = <STDIN>;            /* or simply $line = <> */
```

Reads one line from standard input. Repeated execution of this statement will let you read line by line. The value of $line has the line termination character at the end.

A handy function to remove any line terminator is the Perl built-in function chomp:

```
$str = chomp($line);
```

5.9.2 File I/O

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.

Open Function

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

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

Here **IN** 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
$file = $ARGV[0];                              # file name, a string
open(IN, $file) || die("can't open $file:$!");    # opens or fails
@lines = <IN>;                                 # reads into an array
close(IN);                                     # closes input file
print @lines;
```

Following is the syntax to open file.txt in **writing** mode. Here **greater than (>)** sign indicates that file has to be opened in writing mode

open(OUT, ">file.txt") or die "Couldn't open file file.txt, $!";

This example actually truncates (empties) the file before opening it for writing, which may not be the desired effect. If we want to open a file for reading and writing, we can put a **plus sign (+)** before the > or < characters. For example, to open a file for updating without truncating it:

open(OUT, "+<file.txt") or die "Couldn't open file file.txt, $!";

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

```
open(OUT,">>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 we can immediately start appending information.

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

Here is the example which will open a file in append mode and print its content over the screen.

```
$file = $ARGV[0];
open (OUT,">>$file") or die "file is failed to open $!";
print OUT "welcome to perl\n";
open (OUT,"<$file") or die "file is failed to open $!";
@arr = <OUT>;
close(OUT);
print "@arr";
```

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

Close Function

To close a **filehandle**, and therefore disassociate the file handle from the corresponding file, we use the **close** function. This flushes the file handle’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 (IN) or die "Couldn't close file properly";
```

File Checks

Check	**Meaning**
-f	is a plain file
-r	is readable
-x	is executable
-e	is exists
-z	is empty
-s	is file size not 0
-w	is writable
-T	is a text file
-l	is symbolic link
-d	is directory

5.9.3 Inter-process I/O

From a Perl program, you can execute any shell-level command (as another process on the same computer) and obtain its output with

```
$result = `command string`
```

where the command string is enclosed in backquotes (``). For example,

```
$files = `ls –l`;
```

You can open another process for reading or writing. For example,

```
open(MAIL, "| /usr/sbin/sendmail") || die("fork failed") ;
```

gives MAIL for writing to the sendmail process.

5.10 A Form-to-Email Program

For many websites, forms are used to collect information for off-line processing. Such forms can be supported by a well-designed server-side program that takes the form-collected data and sends email to designated persons. Hidden fields in the form can be used to customize and control the behavior of the program to suit diverse needs. As early as 1995, such a CGI program was created and placed on the Web by Matt Wright.

Below figure shows the FormToMail architecture. The form-supplied configuration parameters (dashed arrow) controls how the CGI program works. The form-supplied email content (solid arrow) is sent to the target recipient and reflected in the confirmation response page.

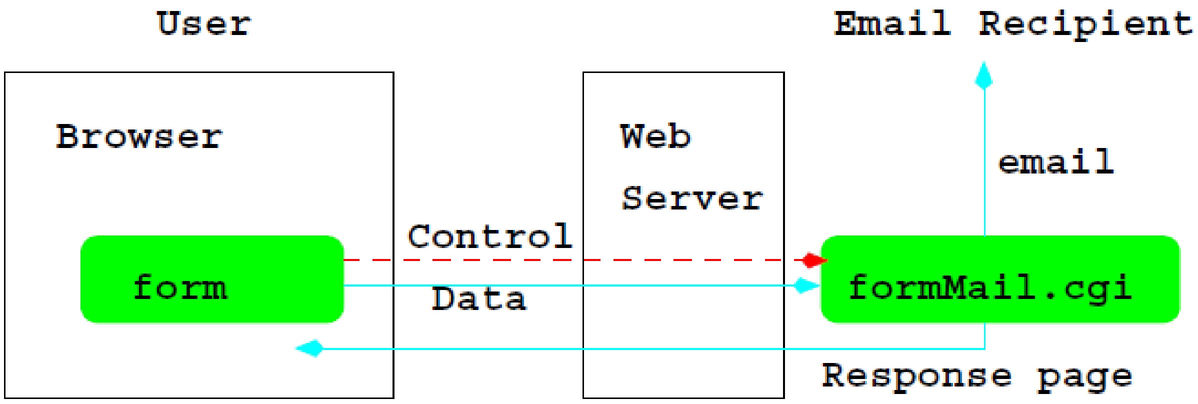


Fig: FormToMail architecture

Form.html

```
<html>
<body>
        <form method="post" action="/cgi-bin/formMail.cgi">
                <input type="hidden" name="formid" value="webtong_hosting" />
                <input type="hidden" name="subject" value="Web Hosting" />
                <input type="hidden" name="page_title" value="Hosting Request Received" />
                <input type="hidden" name="required" value="domain,phone,email,sender_name" />
                Name: <input type="text" name="pname"  size=”25”/><br/>
                E-mail: <input type="text" name="email"  size=”25”/><br/>
                <input type="Submit"  value=”Send Mail”/><br/>
        </form>
</body>
</html>
```

Ex: FormToMail.pl

```
#!/usr/bin/perl

### formMail.cgi
use CGI qw(:standard);                    ## uses CGI
my $formid, $front, $back, $recipient;    ## global vars

#### Customization begin
$mailprog = '/usr/lib/sendmail -t';    ## location of sendmail
@referrers = ();
#### Customization end
my @referrers = ("sacet.ac.in", "131.123.35.90");
my @referrers = ('google.com', 'cs.kent.edu', 'rooster.localdomain');
###
sub checkReferer
{
   if ( $ENV{'REQUEST_METHOD'} eq "GET" ) // disallowed
   {
        error('get request not allowed');
   }
   my $url= $ENV{'HTTP_REFERER'};
   if ( $url )
   {
        foreach $referrer (@referrers)
```

```
                {
                    if ($url =~ m|https?://([^/0-9]*)$referrer|i) # (3)
                    {
                            return;
                    }
              }
         }
         error('bad_referrer');
}
## Configuration values
my %Config=('formid' => '', 'sender_name' => '', 'email' => '', 'subject' => 'Form Email','redirect' => '', 'sort' =>
'','print_blank_fields' =>

'','page_title' => 'Thank You','required' => 'formid,email,sender_name');
my          %Recipient=('webtong_hosting'          =>          'sales@webtong.com','webtong'          =>
'info@webtong.com','wdp_MailForm' => 'test');
my %Cc = ( 'webtong_hosting' => 'DomainMaster@webtong.com' );
?$Config{'formid'}.front
$Config{'formid'}.back

sub formData
{
   my ($name, $value);
   foreach $name ( param() )                    ## for each name-value pair
   {
            $value = param($name);
            if (defined($Config{$name}))        ## set Config values
            {
               if ($name eq 'required')  {
                        $Config{$name} = $Config{$name} . "," . $value;
               }
               else {
                        $Config{$name} = $value;
               }
               if ($name eq 'email' || $name eq 'sender_name')
               {
                        push(@Field_Order,$name);
                        $Form{$name} = $value;
               }
   }
   else                                    ## set Form values
   {
            if ($Form{$name} && $value)
            {
               $Form{$name} = "$Form{$name}, $value";
            }
            elsif ($value)
            {
               push(@Field_Order,$name);
               $Form{$name} = $value;
            }
   }
}
            ## removes white spaces and obtains required fields
            $Config{'required'} =~ s/(\s+|\n)?,(\s+|\n)?/,/g;
       $Config{'required'} =~ s/(\s+)?\n+(\s+)?//g;
            @Required = split(/,/,$Config{'required'});
}

sub checkData
{
   my ($require, @error, $formid);
   $formid = $Config{'formid'};      ## formid
   $recipient=$Recipient{$formid};     ## mail recipient
   if ( (-e "$formid.front") && (-e "$formid.back") )
   {
           $front="$formid.front";     ## response front file
           $back ="$formid.back";              ## response back file
   }
   if (!$recipient) { error('no_recipient') }
   foreach $require (@Required)
   {
           if ($require eq 'email' )       ## email address must be valid
           {
              if ( !checkEmail($Config{$require}))
              {        push(@error,$require); }
           }
           elsif (defined($Config{$require}))  ## check required config values
           {
              if (!$Config{$require})
              {        push(@error,$require);      }
           }
           elsif (!$Form{$require})    ## check required form data
           {           push(@error,$require); }
   }
   if (@error) { error('missing_fields', @error) }         ## If error
}

sub sendMail
{
   if ( $recipient eq "test" ) { return; }
   open(MAIL,"|$mailprog") || die("open $mailprog failed") ;                ## opens mail program
   mailHeaders();                                      ## email headers
   print MAIL "This is a message from the " . "$Config{'site'}. It was submitted by\n";
   print MAIL "$Config{'sender_name'} " . "($Config{'email'}) on $date\n";
   print MAIL "-" x 75 . "\n\n";
   if ($Config{'sort'} eq 'alphabetic') {          ## alphabetical order
           mailFields(sort keys %Form);
   }
   elsif( getOrder() ) {                   ## specific order
           mailFields(@sorted_fields);
   }
   else  {                                 ## no ordering
      mailFields(@Field_Order);
   }
   print MAIL "-" x 75 . "\n\n";
   close (MAIL);
}

checkReferer();                ## checks referring URL
$date = getDate();             ## retrieves current date
formData();                    ## obtains data sent from form
checkData();                   ## checks data for required fields etc.
response();                    ## returns response or redirects
sendMail();                    ## sends email
exit(0);                       ## terminates program
```

5.11 Pattern Matching in Perl

A regular expression is a string of characters that defines 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 Perl relational operators **=~** (match) and ***!~*** (non-match) are used for pattern matching. In Perl, patterns are specified as extended regular expressions. Patterns are given inside /'s (the pattern delimiter).

Symbol	Meaning
/ /	Represents a pattern
^	At beginning of the string
$	At the end of the string
i	ignore case
g	global search

The following are some simple matching examples involving the string $line:

```
$line=”A big fat hen”;
if ( $line =~ /hen/ )               ## contains hen
if ( $line =~ /Hen/ )               ## does not contain Hen
if ( $line =~ /Hent/i )             ## contains Hen, ignoring case
if ( $line =~ /^Hen/ )              ## Hen at beginning of line
if ( $line =~ /hen$/ )              ## hen at end of line
```

If the pattern contains / then it is convenient to use a leading m which allows you to use any non-alphanumeric character as the pattern delimiter:

```
if ( $url =~ m |http://| )
```

Or you can use \ to escape the / in the pattern:

```
if ( $url =~ /http:\/\// )          ##Same as Above
```

Special characters in Perl patterns include:

Pattern	Meaning
\n	A newline
\t	A tab
\w	Any alphanumeric (word) character, same as [a-zA-Z0-9_]
\W	Any non-word character, same as [^a-zA-Z0-9_]
\d	Any digit. The same as [0-9]
\D	Any non-digit. The same as [^0-9]
\s	Any whitespace character: space, \t, \n, etc
\S	Any non-whitespace character

Substitutions

Often we look for a pattern in a string for the purpose of replacing it. This can be done easily with the string matching operator =~:

- $line =~ s/HTTP/http/;   ##replaces first occurrence of HTTP in $line by http.
- $line =~ s/HTTP/http/g;   ##|replaces all occurrences globally in $line.
- $line =~ s/HTTP/http/gi;   ##ignores case in global matching.
- $line =~ s/pattern/cmd/e;   ##uses the replacement string obtained by executing cmd.

5.12 Simple Page Search

The website SymbolicNet.org has an email directory for people in Symbolic Computation, an area of research. The email directory page offers a simple page-search function **(Ex: PageSearch)** allowing users to enter a text string to obtain all email listings matching the given string. Figure 13.4 shows the form at the beginning of the email directory page **(listing.html)**.

The HTML source code for the form is (**listing.html**):

```
<html>
<head>
        <title> Simple Page Search</title>
<head>
<body>
<p>To look for email entries in this page, please enter text to find. </p>
<form method="post" action="/cgi-bin/pagesearch.pl">
         <input type="hidden" name="page"     value="/email/listing.html" />
         <p>Find text:<br /><input name="pattern" size="20" />&nbsp;&nbsp;
        <input type="submit" value="Find All" /> </p>
</form>
```

```
        </body>
        </html>
```

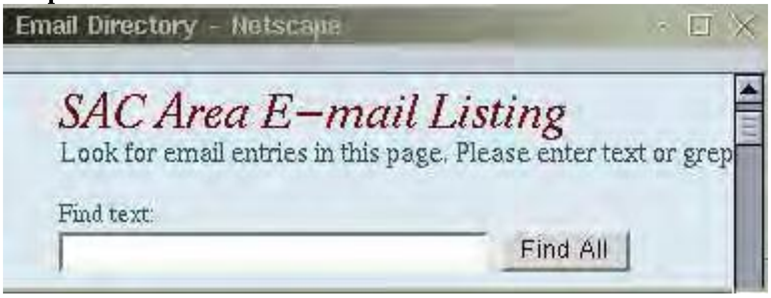
The CGI program **pagesearch.pl** receives the location of the file to search via a hidden form field. The program pagesearch.pl performs these tasks:

- Opens a given email listing page
- Looks for the form supplied pattern in each line of the page
- Remembers all matching lines
- Outputs the count of lines matched followed by all matching lines
- Reports errors when something is wrong

Let's look at the source code of **pagesearch.pl**.

```
        #!/usr/bin/perl
        ## search email listing
        use CGI qw(:standard);
        my $sn_root="/home/httpd/htdocs";
        my $reply="", $error = "", $file;
        my $page = param('page');
        if ( $page eq "" )   {
                $error .= "<p>Page to search not specified!</p>";
        }
        else   {
                $file = $sn_root . $page;
                open(listing, "$file") or $error .= "<p>Can not open $file!</p>";
        }
        $pt = param('pattern');
        if ( $pt eq "" )   {
                $error .= "<p>You didn\'t " . "submit any text to find.</p>";
        }
        if ( $error )     {
                errorReply($error); exit(1);
        }
        ### construct reply
        outputFile("frontfile");
        my $count = 0, $match="";
        while ( defined($line=<listing>))               ## find matching entries
        {
                if ( $line =~ /$pt/i )   {
                        $count++;
                        $match .= $line; ## (15)
                }
        }
        close(listing);
        $reply .= "<h3 style=\"margin-top: 16px\"> Found $count entries matching $pt</h>";
        if ( $count > 0 )   {
                $reply .= "<ol> $match </ol>";
        }
        outputFile("backfile");
        sendPage($reply);
        exit(0);
        sub outputFile
        {
                my($ln, $f);
                $f = $_[0];
                open(FF, $f) || errMail("failed to open the file $f.");
                while ( $line=<FF> )    {
                        $reply .= "$line";
                }
                close(FF);
        }
        sub sendPage
        {
                my $content=$_[0];
                my $length=length($content);
                print "Content-type: text/html\r\n";
                print "Content-length: $length\r\n\r\n";
                print $content;
}
sub errorReply
{
                my $msg=$_[0];
                outputFile("front");
                $reply .= "<h3 style=\"margin-top: 16px\"> Error encountered:</h3> $msg";
                $reply .= "<p>Please go back and submit your request again.</p>";
                outputFile("backfile");
                sendPage($reply);
}
```

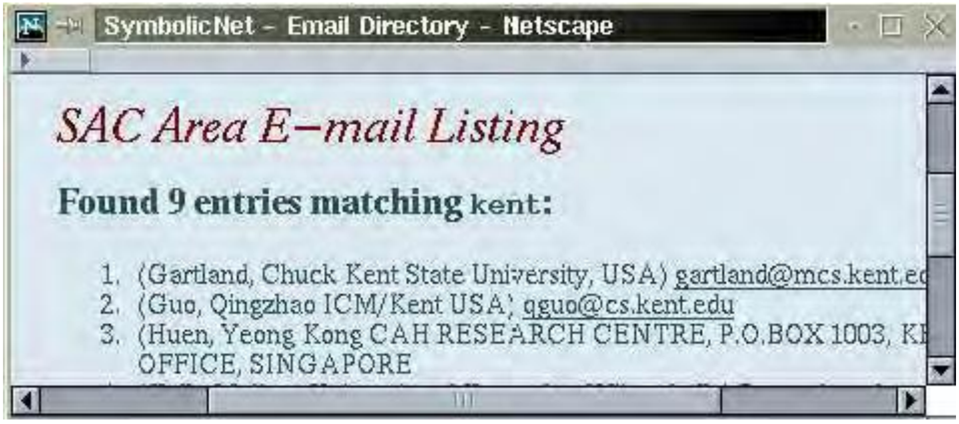
Output:



SAC Area E-mail Listing

Look for email entries in this page. Please enter text or grep

Find text:



SAC Area E-mail Listing

Found 9 entries matching kent:

1. (Gartland, Chuck Kent State University, USA) gartland@mcs.kent.ec
2. (Guo, Qingzhao ICM/Kent USA) qguo@cs.kent.edu
3. (Huen, Yeong Kong CAH RESEARCH CENTRE, P.O.BOX 1003, KI OFFICE, SINGAPORE