Chapter -3

Functions in C++

Objectives :

- **to analyze how modularity is implemented in a program at its lowest level.**
- **To appreciate the use and importance of function in C++**
- **to program different types of functions and implement them practically.**
- **To understand difference between User Define Function and Library function.**

3.1: Why to use functions ?

Who likes unnecessary repetition of task ? No body in this world. Every body thinks of re usability these days in every aspect of life. What will you do if your cycle wheel rim breaks down on fine day? Do you will throw the whole cycle or sell the cycle as scrap? No exactly not because cycles are designed in such a way that each of its parts can be repaired or replaced. So you will get a new cycle rim from market and will get it fitted in your cycle! This design of cycle where each of its parts have its own unique functionality and could be reassembled together to form a complete cycle is known as **Modular approach of designing**. Each of the cycles part can thought as a Module which serves some purpose in the whole cycle but is very essential for proper functioning of the cycle. The whole concept is nothing but based on **"Divide and Rule philosophy".** A bigger system is divided into smaller components so that each of these smaller parts could handled easily and effectively. These smaller parts when integrated gives rise to the bigger system.

Just think GOD has also created human beings as a modular entity ! We humans have a body which is integration of organ system and each of these organ system is again integration of some organs. So here organs are acting as modules. These modules (organs) could be taken care of individually when we often fall ill.

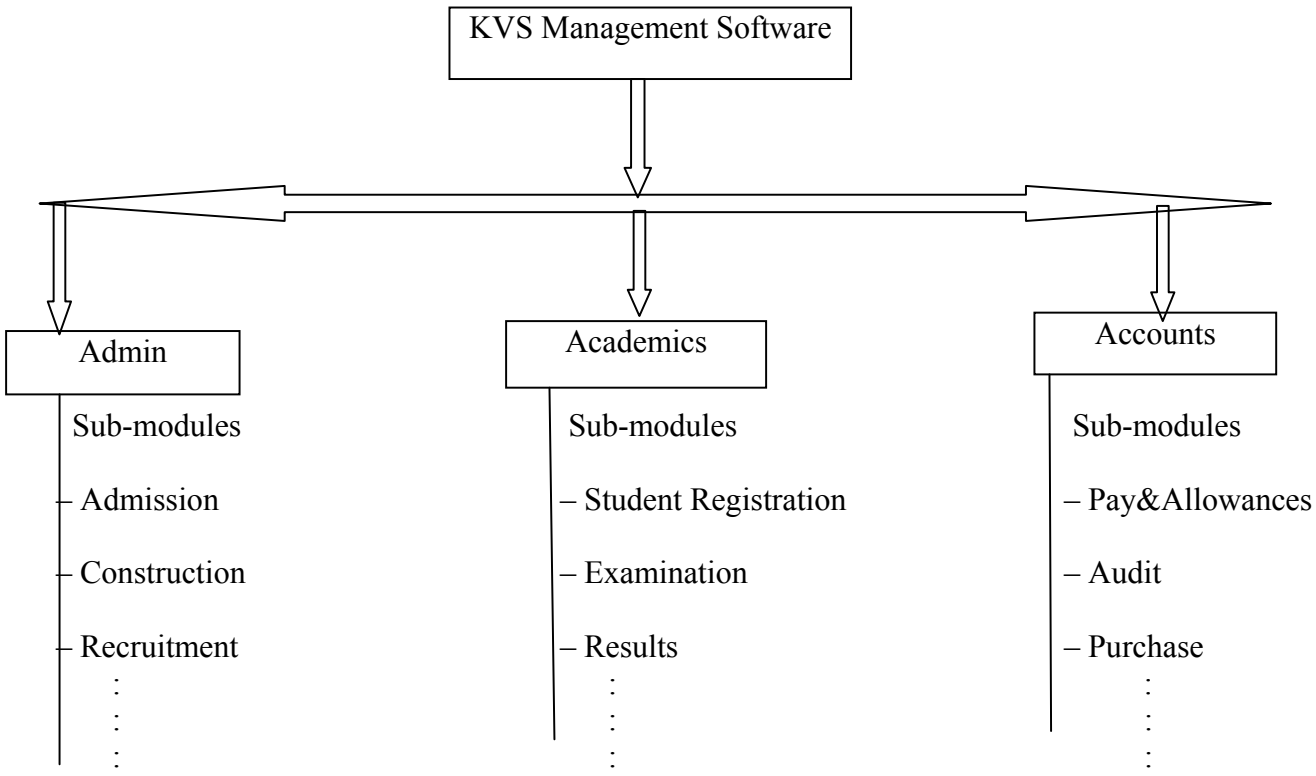
Can you rightly describe what is opposite word for modularity ? Any system which is not modular is known as monolithic (अखंड) or indivisible. A monolithic system does not have any parts or modules, from top to bottom it is one piece.

Software industry has also adopted the modular approach of design where a big software is divided into several modules. Each of the modules are designed for performing specialized task in the whole software. These modules interact with other modules of the system to carry out essential functionality of the whole system.

Each module during its course of execution repeats same type of task, so whenever the whole system requires a specific type of task , for which a particular module is responsible , it calls or invokes that module and the task is done. This calling of module to perform a certain action , can be done several number of times while the software as a whole executes.

Let us understand the above concept with the help of a real life example. Suppose our KVS is going to develop a centralized software for managing all Kvs across the country. While designing such a software KVS has to divide the whole operation of the software into three big modules called as : Admin , Academic and Accounts, each of these modules could be again broken down into many simple and small sub-modules like Admin Module can have Admission , Construction , Recruitment, etc. whereas the Academics can again have sub-modules like Student Registration, Examination , Results etc.

The following diagram describes the whole concept very easily :



When the whole software is divided into modules as seen in the above case scenario then the following benefits could be harvested :

i) Each module can be tracked individually and separately without taking much care of other modules.

ii) A module is a reusable piece of program which could used again and again. Suppose that Navodaya Vidyalaya now wants to make a software like KVS then they can re-use the same modules of KVS with some changes (customization).

iii) If error is found in one module the functionality of that particular module and its associated modules would be disturbed, the whole software will not suffer. Thus errors can be tracked easily and debugged in much less time, because programmer will know which module is causing error, so he will debug that particular module only not the whole software (much like

when you visit doctor suffering from common cold , the doctor does not checks your brain!)

iv) System up gradation (the process of changing from older system to newer one) becomes much easier because only those modules which needs up gradation will be dealt leaving other things as it is.

So we see that modularization of a system gives us much independence and flexibility in terms of fast program development , easier debugging , and re-usability.

How Functions in C++ are related to program modules :

Just as a software is divided into modules , each modules into sub-modules , a sub-module is further divided into several functions. So we may call a **function as a micro-level module** of a bigger software.

A function in C++ :

- is smaller section of code of bigger module/program.
- is re-usable piece of code.
- is very specific in nature as it performs a specific task.
- is often called many times in a program.

Thus a C++ function have all the advantages which a module has in a software.

3.2: Types of function :

Functions in C++ are of two basic types :

a) User Defined : written by a programmer as per his/her requirement domain.
b) Library Function : already available with C++ compiler and stored as Library, from where they can be called and used in any C++ program.

3.2.1 : User Defined Functions :

A user define function in C++ is always created by programmer according to his/her program requirements. Suppose, if some programmer is making software for school management then his list of user defined functions may have functions such as : getFee( ) , calcResult( ) , setExam( ) , all these functions will be used only in school management software not any where else, as they are specially tailored function for school management software.

Let us see the syntax to declare a user defined function :

Function Declaration :

```
<return type> function_name( <parameter list> ) ;
```

where :

return type := is the value which the function returns , if function does not returns any value then we may write there void.

function_name := any valid C++ identifier name

parameter list := declaration of variables of different data types separated by comma these values are inputs passed from outside to the function.

The function declaration as per the syntax given above is also called as **prototype declaration.** In C++ it is compulsory to declare prototype of a function before defining and using it .

The parameter variable in the declaration are also called **Formal parameters.**

Function Definition :

While function definition described about the structure of a function , its inputs and output type , the definition of function actually implements the code of the function. While defining a function we add

C++ code to its block as per requirement.

Syntax :
```
<return type> function_name( <parameter list> )
{  … }
```

Example : Declare and define a function which finds the sum of two integers and returns it.

```
int getSum( int , int );   // declaration of a function

int  getSum( int a  , int b )   // definition of the function
{
    int r = a+b;
    return r;
}
```

The above function declaration has a return type as integer , because the function is meant to return a sum of two numbers. Two numbers to be added are passed to the function as input parameter. The parameter list is having two int separated by a comma (,) it is not compulsory to write a variable names of the parameters in declaration. A semicolon is terminating the declaration of function.

The definition of function is having code written within its scope where the sum is calculated over the passed parameters a and b and the result is returned using a **keyword return.** It is compulsory that the return data type must be same as that of the datatype of the variable returned using return statement.

Workout yourself :

Declare the prototype of a function which :

i) multiplies three integers and return the product
ii) checks whether a passed integer parameter is even or odd
iii) prints your name 20 times.

Consider the few more definition of functions related to various program :

Function *3.1* :

```
        // function to check whether a given number is prime or not
int isPrime(int );
int isPrime(int num )
{
        int count = 0;
        for( int i = 1 ; i <= num ; i++)
          if( num % I ==  0)
             count++;
```

```
        if (count > 2 )
          return 0  ;       // more than two factors means it is not prime , hence a false value is returned
        else
          return 1 ;      // exactly two factors i.e. 1 and num itself means num is prime , hence return a
                          // true value i.e. 1
}
```

In the above function if the passed parameter to the function i.e. num would be a prime it will have exactly two factors counted out in variable count and if not would have more than 2 factors. After we conduct a looping over the num to check its divisibility by every value from 1 to num we get count incremented whenever num is divisible by i (looping-variable). So on the termination of loop the variable count stores the total number of times num gets divisible in the loop.

We check this count value to find whether it is more than two or not, if it is more than two it means num has more than two factors and hence it does not satisfies to be a prime , hence we return an integer 0 to designate that it is not prime , other wise we return 1.

Instead of returning 1 and 0 from function you directly print using cout that num is prime or not, but then don't forget to change the return type of the function **isPrime( ) to void.**

Function 3.2 :

```
// function to print all even numbers found between two given integer parameters

void findSum( int , int);
void findSum(int i , int j )
{
        int sum = 0;
        if( i <= j)
        {
          for( int k = i ; k<= j ; k++)
                if( k % 2 == 0)
                   cout<< k << " ,";
        }
        else
            cout<< "Range not valid";
}
```

Explanation :

In the above function there was no need of returning any value from its scope, we have to print the even numbers found within the for loop. Since a function can return only one value at a time, and once its returns a value the scope of the function finishes all the variables declared in its scope dies off and are no more available to be used again. So instead of returning we printed the multiple outputs using a cout. Look how we have declared the function return type as void when we are not returning any value from its scope.

Function *3.3 :*

```
// program to find the HCF  of two  inputted numbers :
void getHCF( );
```

```
void getHCF( )
{
        int n1 = 0 , n2 = 0;
        int hcf = 0;
        cout<< "Input two numbers whose hcf is to be found";
        cin>>n1>>n2;
        if( n1 == n2 )
          cout<< n1;
        else if( n1 < n2)
        {
            for( int d = 1  ; d <= n2 ; d++)
            {
                if( n1 % d == 0 && n2 % d == 0)
                {
                        hcf = d;
                }
            }
            cout<< "The hcf is " << hcf ;
        }
        else
        {

            for( int d = 1  ; d <= n1 ; d++)
            {
                if( n1 % d == 0 && n2 % d == 0)
                {
                        hcf = d;
                }
            }
            cout<< "The hcf is " << hcf ;
        }
} // end of function
```

Explanation :

Look at the function definition of the above function , since the function is taking the two required inputs within its scope using cout and cin, we had not passed any parameters to function thus we have kept the paranthesis ( ) empty. Also as the function directly prints the output there is no need to return any value from its scope, hence return type is void.

The logic of the function is simple : we are finding the highest of the two numbers and then running a loop from 1 to the highest number and in each iteration we are finding out whether the iterator value divides both the numbers , if it then we put that iterator's value into variable hcf , changing the old value of variable hcf. Thus at the end of the loop the hcf is left out with the Highest Common Factor and we print it with a cout.

Function 3.4 :

```
// function to find the Simple Interest on a given principal , rate , and time where the default value of the
//  time parameter is kept 1 yr.

float   getSimpleInt(float , float , float=1 );
float getSimpleInt(float p , float rate , float time)
```

```
{
        float si  = (p * rate * time)/100;
        return si;
}
```

The above program is having a new type of parameter known as **default parameter.** A C++ function parameter can be made default if that parameter is assigned any constant either at the time of

declaring it or at the time of its definition. When we associate a default value to a parameter as we have associated 1 with last parameter, then this value will be used if the function is getting called without passing this parameter in the call. This we will again see ahead while discussing a function call. The logic of the program is quite easy to understand. Anyway default parameters must be always declared from right to left, otherwise will cause error. i.e you can't make principal alone as default. Following declaration is Invalid.

```
        float SimpleInt(float p = 1000 , float rate , float t=1);
or
        float SimpleInt(float p , float rate = 2 , float t);
```

If you want to make principal as default parameter you have to make all the parameters on its right side default as well.

So the valid declaration in such case would be :

```
float SimpleInt(float p = 1000 , float rate=2 , float t=1);
```

3.2.2 : Calling a C++ Function from a program :

You might be wondering that err!! in the above few function examples , the author has forgot to write a main( ) function (**a main is also a function!) ,** then **how the code written within the scope of the function would be executed?** You might be also wondering that in The general syntax of calling a function would be in above functions 3.1 , 3.2 and 3.4 the **author has not asked the user to input values for variables used in these functions?**

So students it done knowingly , the codes within the scope of a function never gets executed if it not called from the scope of other scope. Just like your badly damaged non-functional land-line is not going to repaired of its own until unless you are not calling a mechanic! . **So a function has to be called from the scope of another function so that the code within the scope of the function works.** Usually we make this call from the scope of main( ) function, **but we can call a function from the scope of any other function if the called function is defined globally.**

The other doubt that author has not provided the input values in few function is also reasonable because a **user should input the parameter values, if not error will be given.** A **user inputs the parameter values from the same scope where from it calls the function.**

Let us know see the syntax of calling a function :

A function call for a non-returning type function (return type is void) :

```
function_name( <parameter_list_if_it is defined_or_leave_it_blank);
```

A function call for a non-returning type function (return type is some **datatype**) :

```
return_datatype <var_name> = function_name( <parameter_list_if_it is
                                                    defined_or_leave_it_blank);
```

While calling a function the value of the parameter are also called **actual parameter**.

Now let us call all the four functions which we have defined earlier i.e. Finction 3.1 to 3.4

//calling of function 3.1 : int isPrime(int); from main( )

```
void main ( )
{
   int val = 0;
   cout<<"input a integer to be checked";
   cin>> val;


   int r = isPrime( val );    //second type of calling

   if ( r == 1 )
     cout<< val << " is prime" ;   // function is returning 1 only when it found the num be
                                   // a prime value.
   else
     cout<< val << " is not a prime" ;
}
```

//calling of function 3.2 : void findSum(int , int); from main( )

```
void main ( )
{
   int v1 = 0 , v2 = 0;
   cout<<"input two integer to be checked";
   cin>>v1>>v2;

   int s = findSum( v1 , v2);  //second type of calling

     cout<< "The sum is"<< s ;   // function returns the sum which is caught by s and
                                 // shown!!
}
```

//calling of function 3.3 : void getHCF( ); from main( )

```
void main ( )
{
```

```
        getHCF( );  //first type of calling
                    // though it appears that this type of calling is an easier method, but it is
                    // rarely used in actual software making industries.
    }
```

//calling of function 3.4 : float getSimpleInt(float , float , float=1);; from main( ) ; we may call this //function a different ways let us see

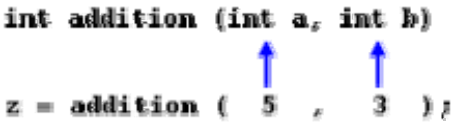
```
    void main()
    {
        float  prnc = 5000, rt = 10 , tm = 5;
        float si = 0.0;
        // first call produces si :  Rs 2500

        si = getSimpleInt(prnc , rt , tm);

        // second call produces si = Rs. 500

        si = getSimpleInt(prnc , rt );  // last parameter is omitted as it was declared as
                                        // as optional type with a default value = 1
    }
```

3.2.3 : Arguments passed by value and by reference

Until now, in all the functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function addition using the following code:

```
int x=5, y=3, z;
z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves.



This way, when the function addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following example:

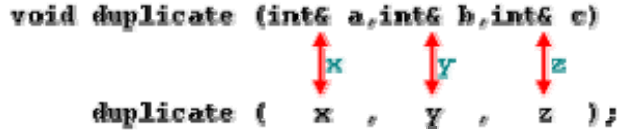
```
// passing parameters by reference
#include <iostream.h>

    void duplicate (int& a, int& b, int& c)
```

```
{
        a*=2;
        b*=2;
        c*=2;
}

        int main ()
        {
                int x=1, y=3, z=7;
                duplicate (x, y, z);
                cout << "x=" << x << ", y=" << y << ", z=" << z;
                return 0;
        }
```

output : x=2, y=6, z=14

The first thing that should call your attention is that in the declaration of duplicate the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.



To explain it in another way, we associate a, b and c with the arguments passed on the function call (x, y and z) and any change that we do on a within the function will affect the value of x outside it. Any change that we do on b will affect y, and the same with c and z.

That is why our program's output, that shows the values stored in x, y and z after the call to duplicate, shows the values of all the three variables of main doubled.

If when declaring the following function:

***void* duplicate (*int*& a, *int*& b, *int*& c)**

we had declared it this way:

***void* duplicate (*int* a, *int* b, *int* c)**

i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of x, y and z without having been modified.

Passing by reference is also an effective way to allow a function to return more than one value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

// more than one returning value

```
#include <iostream.h>
void prevnext (int x, int& prev, int& next)
{
prev = x-1;
```

```
next = x+1;
}

int main ()
{
  int x=100, y, z;
  prevnext (x, y, z);
  cout << "Previous=" << y << ", Next=" << z;
 return 0;
}
```

Ouput :Previous=99, Next=101

3.2.4 : Arguments passed as const parameter

Look at the following protoype :

```
void myFunction( const int x = 10 );
```

The above function is declaring its first parameter as const i.e even if the user tries to pass value to this argument the new value will not change the value assigned to the const parameter, even in the scope of the function also it will remain unchaged.

Let us see one example :

```
void myFunction( const int x = 10 )
{
      cout<< x;
}

main( )
{
         myFunction(3 ); // even if the function is passed with a value 3 it will not accept.
}
```

The output of the code will be 10 instead of 3.
So whenever you don't want to restrict your parameter to a fixed value , declare it as const. Some compiler may produce error.

check your progress :

1. Write a function (waf) to find the largest out of three integers as input parameters.
2. W.a.f to find whether a number either divisible by its predecessor and successor.
3. W.a.f which returns the sum of digits of any integer passed to it as parameter.
4. W.a.f to calculate discount amount on a given rate of item and discount % offered.

3.1.2 : Library Functions :

These functions are ready-made functions available with C++ compiler. They are stored under various header files. A header file is a normal C++ program file with .h extension containing the code for all the C++ functions defined under it. Header files group functions according to its use and common feature.

Following are some important Header files and useful functions within them :

1.	stdio.h (standard I/O function)	:	gets( ) , puts( )
2.	ctype.h (character type function)	:	isalnum( ) , isalpha( ) isdigit ( ), islower ( ),isupper ( ), tolower ( ), toupper( )
3.	string.h (string related function)	:	strcpy ( ), strcat ( ) strlen( ), strcmp( ) , strcmpi( ) , strrev( ) strupr( ) , strlwr( )
4.	math.h (mathematical function)	:	fabs ( ), pow ( ), sqrt ( ), sin ( ), cos ( ), abs ( )
5.	stdlib.h	:	randomize ( ), random ( ), itoa( ) , atoi().

The above list is just few of the header files and functions available under them , but actually there are many more. If you want to learn their use go to the help menu of your turbo C++ compiler and search out function list and learn its prototype.

The calling of library function is just like User defined function , with just few differences as follows:

i) We don't have to declare and define library function.
ii) We must include the appropriate header files , which the function belongs to, in global area so as these functions could be linked with the program and called.

Library functions also may or may not return values. If it is returning some values then the value should be assigned to appropriate variable with valid datatype.

Let us deal with each of these library functions by calling the them from programs :

gets( ) and puts( ) : these functions are used to input and output strings on the console during program run-time.

gets( ) accept a string input from user to be stored in a character array.
puts() displays a string output to user stored in a character array.

program 3.1 :

```
// program to use gets( ) and puts( )
    #include<iostream.h>
    #include<stdio.h>   // must include this line so that gets( ) , puts( ) could be linked and called
    void main( )
    {
        char myname[25] ;   //declaring a character array of size 25

        cout<<"input your name : ";
        gets(myname)   ;   // just pass the array name into the parameter of the function.
        cout<< "You have inputted your name as : " ;
        puts(myname);
    }
```

isalnum( ) , isalpha( ), isdigit( ) : checks whether the character which is passed as parameter to them are alphanumeric or alphabetic or a digit ('0' to '9') . If checking is true functions returns 1.

program 3.2 :

```
// program to use isalnum( ) , isalpha( ), isdigit( )
        #include<iostream.h>
        #include<ctype.h>
        void main( )
        {
                char ch;
                cout<<"Input a character";
                cin>>ch;
                if( isdigit(ch) == 1)
                  cout<<"The inputed character is a digit";
                else if(isalnum(ch) == 1)
                  cout<<"The inputed character is an alphanumeric";
                else if(isalpha(ch) == 1)
                  cout<<"The inputed character is an alphabet.
        }
```

islower ( ),isupper ( ), tolower ( ), toupper( ) : islower( ) checks whether a character has lower case , isupper( ) does opposite. tolower( ) converts any character passed to it in its lower case and the toupper( ) does opposite.

program 3.3:

```
// program to use islower ( ),isupper ( ), tolower ( ), toupper( )
        #include<iostream.h>
        #include<ctype.h>
        void main( )
        {
                char ch;
                cout<<"Input a character";
                cin>>ch;
                if( isupper(ch) == 1)    // checks if character is in upper case converts the character to lowercase
                {
                  tolower(ch);
                  cout<<ch;
                }
                else if(islower(ch) == 1)  // checks if character is in lower case converts the character to
                {                          // uppercase
                  toupper(ch);
                  cout<<ch;
                }
        }
```

fabs ( ), pow ( ), sqrt ( ), sin ( ), cos ( ), abs ( ) :

Program 3.4

```
#include <iostream.h>
#include <math.h>
#define PI 3.14159265 // macro definition PI will always hold 3.14159265

int main ()
```

```
{

 cout<<"The absolute value of 3.1416 is : "<<fabs (3.1416) ;  // abs( ) also acts similarly but only on int data
  cout<<"The absolute value of -10.6 is "<< fabs (-10.6) ;
  cout<<"7.0 ^ 3 = " <<pow (7.0,3);
  cout<<"4.73 ^ 12 = " << pow (4.73,12);
  cout<<"32.01 ^ 1.54 = "<<pow (32.01,1.54);

   double param, result;
   param = 1024.0;
   result = sqrt (param);
   cout<<"sqrt() = "<<result ;

   result = sin (param*PI/180);     // in similar way cos( ) , tan() will be called.
   cout<<"The sine of " <<param<<" degrees is : "<< result ;

  return 0;
}
```

randomize ( ), random ( ), itoa( ) , atoi():

The above functions belongs to header file stdlib.h . Let us observe the use of these functions :

randomize( ) : This function provides the seed value and an algorithm to help random( ) function in generating random numbers. The seed value may be taken from current system's time.

random(<int>) : This function accepts an integer parameter say x and then generates a random value between 0 to x-1

for example : random(7) will generate numbers between 0 to 6.

To generate random numbers between a lower and upper limit we can use following formula :

random(U – L +1) + L

where U and L are the Upper limit and Lower limit values between which we want to find out random values.

For example : If we want to find random numbers between 10 to 100 then we have to write code as :

```
random(100 -10 +1) + 10 ;    // generates random number between 10 to 100
```

Check you progress :

1. Name the header files to which the following functions belongs:

i) toupper( ) , ii) fabs( ) , iii) sqrt( ) , iv) strcpy( )

2. Write a program to check whether a string variable is palidrome or not , using only library function.

----------x----------end of chapter 3----------x----------

Summary :

1. Functions provides modularity to programs.
2. Functions can be re-used.
3. Functions can be of two types : User Defined and Library.
4. We have to declare and define user define functions either in program-scope or header files to use them in a program.
5. To use library functions appropriate header files must be included using #include directive
6. A prototype of a function means , the number , data-types and sequence of its parameters.
7. During function call it is very necessary to pass parameter values (actual parameters) to the function in the same order and type as define in its prototype.
8. The Library functions are kept within header files. These header files are made as per the working nature of the function.
9. A big software must use modular approach of programming to which function is as essential programming element.
10. Sometimes instead of passing values as parameters to the function we pass its references to the function. This type of calling is known as call by reference.