Functions

Learning Objectives

After learning this chapter, the students will be able to

- Understand the Definition of Functions and uses of Functions
- Understand the Types of Functions – pre-defined and user-defined functions
- Apply mathematical functions for solving problems.
- Use String and Character functions for the manipulation of String and Character data
- Implement modular programming by creating functions
- Understand the role of arguments and compare different methods of the arguments
- Recognizes the scope of variables and functions in a program.

11.1 INTRODUCTION

A large program can be split into small sub-programs (blocks) called as functions where each sub-program can perform some specific functionality. Functions reduce the size and complexity of a program, makes it easier to understand, test, and check for errors. The functions which are available by default are known as **"Built-in"** functions and user can create their own functions known as **"User-defined"** functions.

- Built-in functions – Functions which are available in C++ language standard library.
- User-defined functions – Functions created by users.

11.2 Need for Functions

To reduce size and complexity of the program we use Functions. The programmers can make use of sub programs either writing their own functions or calling them from standard library.

1. Divide and Conquer

- Complicated programs can be divided into manageable sub programs called functions.
- A programmer can focus on developing, debugging and testing individual functions.
- Many programmers can work on different functions simultaneously.

2. Reusability

- Few lines of code may be repeatedly used in different contexts. Duplication of the same code can be eliminated by using functions which improves the maintenance and reduce program size.
- Some functions can be called multiple times with different inputs.

11.3 Types of Functions

Functions can be classified into two types,

1. Pre-defined or Built-in or Library Functions
2. User-defined Function.

C++ provides a rich collection of functions ready to be used for various tasks. The tasks to be performed by each of these are already written, debugged and compiled, their definitions alone are grouped and stored in files called **header files**. Such ready-to-use sub programs are called **pre-defined functions or built-in functions.**

C++ also provides the facility to create new functions for specific task as per user requirement. The name of the task and data required (arguments) are decided by the user and hence they are known as **User-defined functions.**

11.4 C++ Header Files and Built-in Functions

Header files provide function prototype and definitions for library functions. Data types and constants used with the library functions are also defined in them. A header file can be identified by their file extension **.h.** A single header file may contain multiple built-in functions.

For example: **stdio.h** is a header file that contains pre-defined **"standard input/ output"** functions.

11.4.1 Standard input/output (stdio.h)

This header file defines the standard I/O predefined functions **getchar(), putchar(), gets(), puts()** and etc.

11.4.1.1 getchar() and putchar() functions

The predefined function **getchar()** is used to get a single character from keyboard and **putchar()** function is used to display it.

Program 11.1 C++ code to accept a character and display it

```
#include<iostream>
#include<stdio.h>
using namespace std;
int main()
{
        cout<<"\n Type a Character : ";
        char ch = getchar();
        cout << "\n The entered Character is: ";
        putchar(ch);
        return 0;
}
```

Output:

```
Type a Character : T
The entered Character is: T
```

11.4.1.2. gets() and puts() functions

Function gets() reads a string from standard input and stores it into the string pointed by the variable. Function **puts()** prints the string read by **gets()** function in a newline.

Program 11.2 C++ code to accept and display a string

```
#include<iostream>
#include<stdio.h>
using namespace std;
int main()
{
        char str[50];
        cout<<"Enter a string : ";
        gets(str);
        cout<<"You entered: "
        puts(str);
        return(0);
}
```

Output :

```
Enter a string : Computer Science
You entered: Computer Science
```

11.4.2 Character functions (ctype.h)

This header file defines various operations on characters. Following are the various character functions available in C++. The header file **ctype.h** is to be included to use these functions in a program.

11.4.2.1.isalnum()

This function is used to check whether a character is **alphanumeric or not**. This function returns non-zero value if c is a digit or a letter, else it returns 0.

General Form:

int isalnum (char c);

Example :

int r = isalnum('5');

cout << isalnum('A') <<'\t'<<r;

But the statements given below assign 0 to the variable n, since the given character is neither an alphabet nor a digit.

char c = '$';

int n = isalnum(c);

cout<<c;

Output:

0

Program 11.3

```
#include<iostream>
#include<stdio.h>
#include<ctype.h>
using namespace std;
int main()
{
        char ch;
        int r;
        cout<<"\n Type a Character :";
        ch = getchar();
        r = isalnum(ch);
        cout<<"\nThe Return Value of isalnum(ch) is :"<<r;
}
```

Output-1:

```
Type a Character :A
The Return Value of isalnum(ch) is :1
```

Output-2:

```
Type a Character :?
The Return Value of isalnum(ch) is :0
```

11.4.2.2. isalpha()

The isalpha() function is used to check whether the given character is an alphabet or not.

General Form:

isalpha(char c);

This function will return 1 if the given character is an alphabet, and 0 otherwise 0. The following statement assigns 0 to the variable n, since the given character is not an alphabet.

int n = isalpha('3');

But, the statement given below displays 1, since the given character is an alphabet.

cout << isalpha('a');

Program 11.4

```
#include<iostream>
#include<stdio.h>
#include<ctype.h>
using namespace std;
int main()
{
        char ch;
        cout << "\n Enter a charater: ";
        ch = getchar();
        cout<<"\n The Return Value of isalpha(ch) is :" << isalpha(ch) ;
}
```

Output-1:

Enter a charater: A
The Return Value of isalpha(ch) is :1

Output-2:

Enter a charater: 7
The Return Value of isalpha(ch) is :0

11.4.2.3 isdigit()

This function is used to check whether a given character is a digit or not. This function will return 1 if the given character is a digit, and 0 otherwise.

General Form:

isdigit(char c);

Program 11.5

```
using namespace std;
#include<iostream>
#include<ctype.h>
int main()
{
        char ch;
        cout << "\n Enter a Character: ";
        cin >> ch;
        cout<<"\n The Return Value of isdigit(ch) is :" << isdigit(ch) ;
}
```

Output-1

Enter a Character: 3
The Return Value of isdigit(ch) is :1

Output-2

Enter a Character: A
The Return Value of isdigit(ch) is :0

***Return 0; (Not Compulsory in latest compilers)**

11.4.2.4. islower()

This function is used to check whether a character is in lower case (small letter) or not. This functions will return a non-zero value, if the given character is a lower case alphabet, and 0 otherwise.

General Form:

islower(char c)

After executing the following statements, the value of the variable **n** will be **1** since the given character is in lower case.

```
char ch = 'n';
int n = islower(ch);
```

But the statement given below will assign 0 to the variable n, since the given character is an uppercase alphabet.

```
int n = islower('P');
```

11.4.2.5. isupper()

This function is used to check the given character is uppercase. This function will return 1 if true otherwise 0.

General Form:

isupper(char c)

For the following examples value **1** will be assigned to **n** and **0** for m.

```
int n=isupper('A');
int m=isupper('a');
```

11.4.2.6. toupper()

This function is used to convert the given character into its uppercase. This function will return the upper case equivalent of the given character. If the given character itself is in upper case, the output will be the same.

General Form:

char toupper(char c);

The following statement will assign the character constant **'K'** to the variable c.

```
char c = toupper('k');
cout<<c;
```

11.4.2.7. tolower()

This function is used to convert the given character into its lowercase. This function will return the lower case equivalent of the given character. If the given character itself is in lower case, the output will be the same.

General Form:

char tolower(char c)

The following statement will assign the character constant **'k'** to the variable c.

```
char c = tolower('K');
cout <<c;
```

11.4.3 String manipulation (string.h)

The library **string.h** (also referred as cstring) has several common functions for dealing with strings stored in array of characters. The string.h header file is to be included before using any string function.

11.4.3.1 strcpy()

General Form:

strcpy(Target String, Source String);

The **strcpy()** function takes two arguments: target and source. It copies the character string pointed by the source to the memory location pointed by the target. The null terminating character **(\0)** attached to the string is also copied.

Program 11.6

```
#include <string.h>
#include <iostream>
using namespace std;
int main()
{
        char source[] = "Computer Science";
        char target[20]="target";
        cout<<"\n String in Source Before Copied :"<<source;
        cout<<"\n String in Target Before Copied :"<<target;
        strcpy(target,source);
        cout<<"\n String in Target After strcpy function Executed :"<<target;
        return 0;
}
```

Output:

```
String in Source Before Copied :Computer Science
String in Target Before Copied :target
String in Target After strcpy function Executed :Computer Science
```

11.4.3.2 strlen()

The **strlen()** takes a null terminated string as its argument and returns its length. The length does not include the null(\0) character.

General Form:

strlen(string);

Program 11.7

```
#include <string.h>
#include <iostream>
using namespace std;
int main()
{
        char source[ ] = "Computer Science";
        cout<<"\n Given String is "<<source<<" its Length is "<<strlen(source);
        return 0;
}
```

Output:

```
Given String is Computer Science its Length is 16
```

11.4.3.3 strcmp()

The **strcmp()** function takes two arguments: string1 and string2. It compares the contents of string1 and string2 lexicographically.

General Form:

strcpy(String1, String2);

The strcmp() function returns a:

- Positive value if the first differing character in string1 is greater than the corresponding character in string2. (ASCII values are compared)
- Negative value if the first differing character in string1 is less than the corresponding character in string2.
- 0 if string1 and string2 are equal.

Program 11.8

```
#include <string.h>
#include <iostream>
using namespace std;
int main()
{
        char string1[] = "Computer";
        char string2[] = "Science";
        int result;
        result = strcmp(string1,string2);
        if(result==0)
        {
        cout<<"String1 : "<<string1<<" and String2 : "<<string2 <<"Are Equal";
        }
        if (result<0)
        {
        cout<<"String1 :"<<string1<<" and String2 : "<<string2 <<" Are Not Equal";
        }
}
```

Output

String1 : Computer and String2 : Science Are Not Equal

11.4.3.4 strcat()

The **strcat()** function takes two arguments: target and source. This function appends copy of the character string pointed by the source to the end of string pointed by the target.

General Form:

strcat(Target, source);

Program 11.9

```
#include <string.h>
#include <iostream>
using namespace std;
int main()
{
        char target[50] = "Learning C++ is fun";
        char source[50] = " , easy and Very useful";
        strcat(target, source);
        cout << target ;
        return 0;
}
```

Output

Learning C++ is fun , easy and Very useful

11.4.3.5 strupr()

The **strupr()** function is used to convert the given string into Uppercase letters.

General Form:

strcat(string);

Program 11.10

```
using namespace std;
#include<iostream>
#include<ctype.h>
#include<string.h>
int main()
{
        char str1[50];
        cout<<"\nType any string in Lower case :";
        gets(str1);
        cout<<"\n Converted the Source string “<<str1<<into Upper Case is "<<strupr(str1);
        return 0;
}
```

Output:

Type any string in Lower case : computer science

Converted the Source string computer science into Upper Case is COMPUTER SCIENCE

11.4.3.6 strlwr()

The **strlwr()** function is used to convert the given string into Lowercase letters.

General Form:

strlwr(string);

Program 11.11

```
using namespace std;
#include<iostream>
#include<ctype.h>
#include<string.h>
int main()
{
        char str1[50];
        cout<<"\nType any string in Upper case :";
        gets(str1);
        cout<<"\n Converted the Source string “<<str1<<into Lower Case is "<<strlwr(str1);
}
```

Output:

```
        Type any string in Upper case : COMPUTER SCIENCE
Converted the Source string COMPUTER SCIENCE into lower Case is computer science
```

11.4.4 Mathematical functions (math.h)

Most of the mathematical functions are defined in math.h header file which includes basic mathematical functions.

11.4.4.1 cos() function

The cos() function takes a single argument in radians. The cos() function returns the value in the range of [-1, 1]. The returned value is either in double, float, or long double.

Program 11.12

```
#include <iostream>
#include <math.h>
using namespace std;
int main()
{
        double x = 0.5, result;
        result = cos(x);
        cout << "COS("<<x<<")= "<<result;
}
```

Output:

```
        COS(0.5)= 0.877583
```

11.4.4.2 sqrt() function

The **sqrt()** function returns the square root of the given value. The **sqrt()** function takes a single non-negative argument. If a **negative value** is passed as an argument to **sqrt()** function, a **domain error occurs**.

Program 11.13

```
#include <iostream>
#include <math.h>
using namespace std;
int main()
{
        double x = 625, result;
        result = sqrt(x);
        cout << "sqrt("<<x<<") = "<<result;
        return 0;
}
```

Output:

```
        sqrt(625) = 25
```

11.4.4.3 sin() function

The **sin()** function takes a single argument in radians. The **sin()** function returns the value in the range of [-1, 1]. The returned value is either in double, float, or long double.

11.4.4.4 pow() function

The **pow()** function returns base raised to the power of an exponent. If any argument passed to **pow()** is long double, the return type is promoted to long double. If not, the return type is double. The **pow()** function takes two arguments:

- **base** - the base value
- **exponent** - exponent of the base

Program 11.14

```
#include <iostream>
#include <math.h>
using namespace std;
int main ()
{
        double base, exponent, result;
        base = 5;
        exponent = 4;
        result = pow(base, exponent);
        cout << "pow("<<base << "^" << exponent << ") = " << result;
        double x = 25;;
        result = sin(x);
        cout << "\nsin("<<x<<")= "<<result;
        return 0;
}
```

Output:

```
        pow(5^4) = 625
        sin(25)= -0.132352
```

11.5 User-defined Functions

11.5.1 Introduction

We can also define new functions to perform a specific task. These are called as **user-defined functions**. User-defined functions are created by the user. A function can optionally define input parameters that enable callers to pass arguments into the function. A function can also optionally return a value as output. Functions are useful for encapsulating common operations in a single reusable block, ideally with a name that clearly describes what the function does.

11.5.2 Function Definition

In C++, a function must be defined before it is used anywhere in the program. The general syntax of a function definition is:

Return_Data_Type Function_name(parameter list)

{

Body of the function

}

Note:

1. The Return_Data_Type is any valid data type of C++.
2. The Function_name is a user-defined identifier.
3. The parameter list, which is optional, is a list of parameters, i.e. a list of variables preceded by data types and separated by commas.
4. The body of the function comprises C++ statements that are required to perform the intended task of this function.

11.5.3 Function Prototype

C++ program can contain any number of functions. But, it must always have **only one main() function** to begin the program execution. We can write the definitions of functions in any order as we wish. We can define the main() function first and all other functions after that or we can define all the needed functions prior to main(). Like a variable declaration, a function must be declared before it is used in the program. The declaration statement may be given outside the main() function.

long fact (int, double)

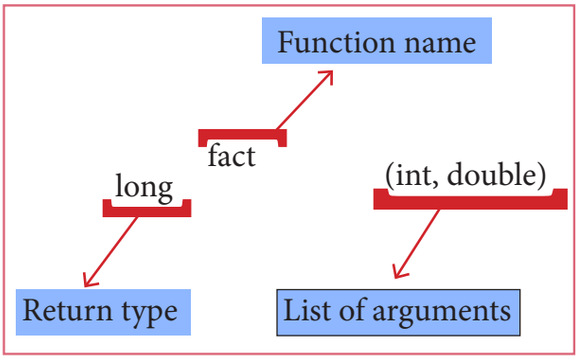

Figure 11.1

The prototype above provides the following information to the compiler:

- The return value of the function is of type long.
- **fact** is the name of the function.
- the function is called with two arguments:
  The first argument is of int **data** type.
  The second argument is of **double** data type.
  int display(int, int) // function prototype//

The above function prototype provides details about the return data type, name of the function and a list of formal parameters or arguments.

11.5.4 Use of void command

void type has two important purposes:

- To indicate the function does not return a value
- To declare a generic pointer.

Notes

void data type indicates the compiler that the function does not return a value, or in a larger context void indicates that it holds nothing.

For Example:

void fun(void)

The above function prototype tells compiler that the function **fun()** neither receives values from calling program nor return a value to the calling program.

11.5.5 Accessing a function

The user-defined function should be called explicitly using its name and the required arguments to be passed. The compiler refers to the function prototype to check whether the function has been called correctly. If the argument type does not match exactly with the data type defined in the prototype, the compiler will perform type conversion, if possible. If type conversion is impossible, the compiler generates an error message.

Example :

1	display()	calling the function without a return value and without any argument
2	display (x, y)	calling the function without a return value and with arguments
3	x = display()	calling the function with a return value and without any argument
4	x = display (x, y)	calling the function with a return value and with arguments

11.5.5.1 Formal Parameters and Actual Parameters or Arguments

Arguments or parameters are the means to pass values from the calling function to the called function. The variables used in the function definition as parameters are known as formal parameters. The constants, variables or expressions used in the function call are known as actual parameters.

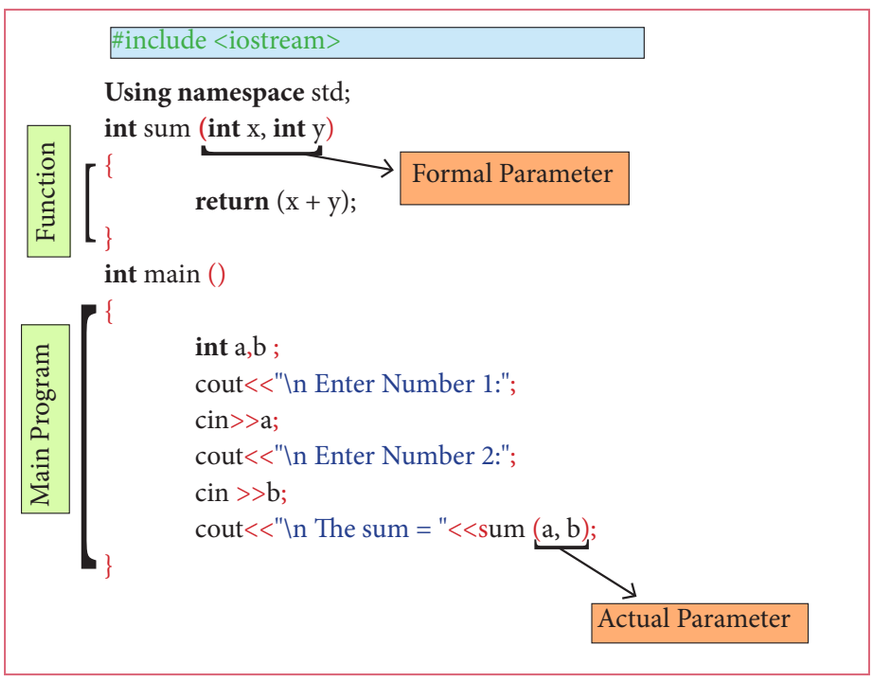

Figure 11.2 Formal and Actual Parameters

11.5.5.2 Default arguments

In C++, one can assign default values to the formal parameters of a function prototype. The Default arguments allows to omit some arguments when calling the function.

When calling a function,

- For any missing arguments, complier uses the values in default arguments for the called function.
- The default value is given in the form of variable initialization.

Example : **void defaultvalue(int n1=10, n2=100);**

- The default arguments facilitate the function call statement with partial or no arguments.

Example : **defaultvalue(x,y);**

defaultvalue(200,150);

defaultvalue(150);

defaultvalue(x,150);

- The default values can be included in the function prototype from right to left, i.e., we cannot have a default value for an argument in between the argument list.

Example : **void defaultvalue(int n1=10, n2);**//invalid prototype

void defaultvalue(int n1, n2 = 10);//valid prototype

11.5.5.3 Constant Arguments

The constant variable can be declared using **const** keyword. The **const** keyword makes variable value stable. The constant variable should be initialized while declaring. The **const** modifier enables to assign an initial value to a variable that cannot be changed later inside the body of the function.

Syntax :

<returntype><functionname> (const <datatype variable=value>)

Example:

- int minimum(const int a=10);
- float area(const float pi=3.14, int r=5);

Program 11.16

```
#include <iostream>
using namespace std;
double area(const double r,const double pi=3.14)
{
        return(pi*r*r);
}
int main ()
{
        double rad,res;
        cout<<"\nEnter Radius :";
        cin>>rad;
        res=area(rad);
        cout << "\nThe Area of Circle ="<<res;
        return 0;
}
```

Output:

```
Enter Radius :5
The Area of Circle =78.5
```

If the variable value "r" is changed as **r=25;** inside the body of the function **"area"** then compiler will throw an error as **"assignment of read-only parameter 'r'"**

```
double area(const double r,const double pi=3.14)
{
        r=25;
        return(pi*r*r);
}
```

11.6 Methods of calling functions

In C++, the arguments can be passed to a function in two ways. Based on the method of passing the arguments, the function calling methods can be classified as **Call by Value method** and **Call by Reference or Address method**.

11.6.1 Call by value Method

This method copies the value of an actual parameter into the formal parameter of the function. In this case, changes made to formal parameter within the function will have no effect on the actual parameter.

Program 11.17

```
#include<iostream>
using namespace std;
void display(int x)
{
        x=x*x;
        cout<<"\n\nThe Value inside display function (x*x):"<<x;
}
int main()
{
          int a;
        cout<<"\nExample : Function call by value:";
          cout<<"\n\nEnter the Value for A :";
          cin>>a;
        display(a);
        cout<<"\n\nThe Value inside main function "<<a;
        return(0);
}
```

Output :

```
Example : Function call by value
Enter the Value for A : 5
The Value inside display function (a * a) : 25
The Value inside main function 5
```

11.6.2 Call by reference or address Method

This method copies the address of the actual argument into the formal parameter. Since the address of the argument is passed ,any change made in the formal parameter will be reflected back in the actual parameter.

Program 11.18

```
#include<iostream>
using namespace std;
void display(int &x) //passing address of a//
{
        x=x*x;
        cout<<"\n\nThe Value inside display function ( x*x) :"<<x ;
 }
int main()
{
int n1;
cout<<"\nEnter the Value for N1 :";
cin>>n1;
cout<<"\nThe Value of N1 is inside main function Before passing : "<< n1;
display(n1);
cout<<"\nThe Value of N1 is inside main function After passing (n1 x n1)  : "<< n1; return(0);
}
```

Output :

```
Enter the Value for N1 :45
The Value of N1 is inside main function Before passing : 45
The Value inside display function (n1 x n1) :2025
The Value of N1 is inside main function After passing (n1 x n1)  : 2025
```

Note that the only change in the **display()** function is in the function header. The **&** symbol in the declaration of the parameter **x** means that the argument is a reference variable and hence the function will be called by passing reference. Hence when the argument **n1** is passed to the **display()** function, the variable **x** gets the address of **n1** so that the location will be shared. In other words, the variables **x** and **n1** refer to the same memory location. We use the name **n1** in the main() function, and the name **x** in the **display()** function to refer the same storage location. So, when we change the value of **x**, we are actually changing the value of **n1**.

11.6.3 Inline function

Normally the call statement to a function makes a compiler to jump to the functions (the definition of the functions are stored in STACKS) and also jump back to the instruction following the call statement. This reduces the speed of program execution. Inline functions can be used to reduce the overheads like STACKS for small function definition.

An **inline** function looks like normal function in the source file but inserts the function's code directly into the calling program. To make a function inline, one has to insert the keyword **inline** in the function header.

Syntax :

inline returntype functionname(datatype parameter 1, … datatype parameter n)

Advantages of inline functions:

- Inline functions execute faster but requires more memory space.
- Reduce the complexity of using STACKS.

Program 11.19

```
#include <iostream>
using namespace std;
inline int add (int a , int b)
{
        int c=a+b;
        return(c);
}
int main ()
{
        int x,y,z;
        cout<<"\nEnter the First Number :";
        cin>>x;
        cout<<"\nEnter the second Number     :";
        cin>>y;
        z=add(x,y);
        cout << "\n sum of "<<x<<"+"<<y<<"="<<z;
        return 0;
}
```

Output:

```
Enter the First Number :10
Enter the second Number     :20
sum of 10+20=30
```

Though the above program is written in the normal function definition format during compilation the function code **a+b** will be directly inserted in the calling statement i.e. **z=add(x,y);** this makes the calling statement to change as **z = a+b;**

11.7 Different forms of User-defined Function declarations

11.7.1 A Function without return value and without parameter

The following program is an example for a function with no return and no arguments passed .

The name of the function is **display()**, its return data type is void and it does not have any argument.

Program 11.20

```
#include<iostream>
using namespace std;
void display()
{       cout<<"First C++ Program with Function"; }
int main()
{       display(); // Function calling statement//
        return(0);
}
```

Output :

```
        First C++ Program with Function
```

11.7.2 A Function with return value and without parameter

The name of the function is **display()**, its return type is int and it does not have any argument. The **return** statement returns a value to the calling function and transfers the program control back to the calling statement.

Program 11.21

```
#include<iostream>
using namespace std;
int display()
{
        int a=10, b=5, s;
        s=a+b;
        return s;
}
int main()
{       int m=display();
        cout<<"\nThe Sum="<<m;
        return(0);
}
```

Output :

```
        The Sum=15
```

11.7.3 A Function without return value and with parameter

The name of the function is **display()**, its return type is void and it has two parameters or arguments **x** and **y** to receive two values. The **return** statement returns the control back to the calling statement.

Program 11 .22

```
#include<iostream>
using namespace std;
void display(int x, int y)
{
        int s=x+y;
        cout<<"The Sum of Passed Values: "<<s;
}
int main()
{
        int a=50,b=45;
          display(a,b);
        return(0);
}
```

Output :

The Sum of Passed Values: 95

11.7.4 A Function with return value and with parameter

The name of the function is display(), its return type is int and it has two parameters or arguments **x** and **y** to receive two values. The return statement returns the control back to the calling statement.

Program 11.23

```
#include<iostream>
using namespace std;
int display(int x, int y)
{
        int s=x+y;
        return s;
}
int main()
{
        int a=45,b=20;
        int s=display(a,b);
        cout<<"\nExample:Function with Return Value and  with Arguments";
        cout<<"\nThe Sum of Passed Values: "<<s;
        return(0);
}
```

Output :

Example: Function with Return Value and with Arguments

The Sum of Passed Values: 65

11.8 Returning from function

Returning from the function is done by using the **return** statement.

The **return** statement stops execution and returns to the calling function. When a **return** statement is executed, the function is terminated immediately at that point.

11.8.1 The return statement

The **return** statement is used to return from a function. It is categorized as a jump statement because it terminates the execution of the function and transfer the control to the called statement. A **return** may or may not have a value associated with it. If return has a value associated with it, that value becomes the return value for the calling statement. Even for void function return statement without parameter can be used to terminate the function.

Syntax:

return expression/variable;

Example : return(a+b); return(a);

return; // to terminate the function

11.8.2 Returning values:

The functions that return no value is declared as void. The data type of a function is treated as **int**, if no data type is explicitly mentioned. For example,

For Example :

int add (int, int);

add (int, int);

In both prototypes, the return value is int, because by default the return value of a function in C++ is of type **int** when no return value is explicitly given. Look at the following examples:

Sl.No	Function Prototype	Return type
1	int sum(int, float)	int
2	float area(float, float)	float
3	char result()	char
4	double fact(int n)	double

Returning Non-integer values

A string can also be returned to a calling statement.

Program 11.24

```
#include<iostream>
#include<string.h>
using namespace std;
char *display()
{       return (“chennai”);   }
int main()
{
        char s[50];
        strcpy(s,display());
        cout<<”\nExample:Function with Non Integer Return”<<s;
        return(0);}
```

Output :

Example: Function with Non Integer Return Chennai

11.9 Recursive Function

A function that calls itself is known as recursive function. And, this technique is known as recursion.

Example 1: Factorial of a Number Using Recursion

Program 11.25

```
#include <iostream>
using namespace std;
int factorial(int); // Function prototype //
int main()
{
        int no;
        cout<<"\nEnter a number to find its factorial: ";
        cin >> no;
        cout << "\nFactorial of Number " << no <<" = " << factorial(no);
        return 0;
}
int factorial(int m)
{
        if (m > 1)
        {
        return m*factorial(m-1);
        }
        else
        {
        return 1;
}
}
```

Output :

Enter a number to find its factorial: 5
Factorial of Number 5 = 120

Note: Function prototype is mandatory since the function factorial() is given after the main() function.

11.10 Scope Rules of Variables

Scope refers to the accessibility of a variable. There are four types of scopes in C++. They are: **Local scope, Function scope, File scope** and **Class scope**.

11.10.1 Introduction

A scope is a region or life of the variable and broadly speaking there are four places, where variables can be declared,

- Inside a block which is called local variables.
- Inside a function is called function variables.
- Outside of all functions which is called global variables.
- Inside a class is called class variable or data members.

11.10.2 Local Scope:

- A local variable is defined within a block. A block of code begins and ends with curly braces { }.
- The scope of a local variable is the block in which it is defined.
- A local variable cannot be accessed from outside the block of its declaration.
- A local variable is created upon entry into its block and destroyed upon exit.

11.10.3 Function Scope:

- The scope of variables declared within a function is extended to the function block, and all sub-blocks therein.
- The life time of a function scope variable, is the life time of the function block. The scope of formal parameters is function scope.

11.10.4 File Scope:

- A variable declared above all blocks and functions (including main ( )) has the scope of a file. The life time of a file scope variable is the life time of a program.
- The file scope variable is also called as **global variable**.

Program 11.26

```
//Demo to test all Scopes//
#include<iostream>
using namespace std;
int file_var=20;          //Declared within File - file scope variable
void add(int x)
{
        int m;            //Declaration of  variable m in add () - Function scope variable
        m=x+30+file_var;
        cout<<"\n The Sum = "<<m;
}
```

```
int main ( )
{
int a =5,b=10;
if(a>b)
{
        int t;          // local to this if block - Local variable
        t=a+20;
 }
cout<<t;
add(a);
cout<<m;
cout<<"\nThe File Variable = "<<file_var;
return(0);
}
```

Error
In function 'int main()':
[Error] 't' was not declared in main()

On compilation the Program 11.28, the compiler prompts an error message: The variable **t** is not accessible. Because the life time of a local variable is the life time of a block in its state of execution.

[Error] 'm' was not declared in this scope

The variable **m i**s not accessible. Because the life time of the function scope variable is the life time of a block in its state of execution.

11.10.5 Class Scope:

- A class is a new way of creating and implementing a user defined data type. Classes provide a method for packing together data of different types.
- Data members are the data variables that represent the features or properties of a class.

class student
{
private :
int mark1, mark2, total;
};
	The class student contains mark1, mark2 and total are data variables. Its scope is within the class student only.

Note: The class scope will be discussed later in chapter **"Classes and Object"**.

11.10.6 Scope resolution operator

The scope operator reveals the hidden scope of a variable. The scope resolution operator (::) is used for the following purposes.

- To access a Global variable when there is a Local variable with same name. An example using Scope Resolution Operator.

Program 11.27

```
// Program to show that we can access a global variable
// using scope resolution operator :: when there is a local
// variable with same name //
#include<iostream>
using namespace std;
int x=45;  // Global Variable x
int main()
{
  int x = 10; // Local Variable x
  cout << "\nValue of global x is " << ::x;
  cout << "\nValue of local x is " << x;
  return 0;
}
```

Output:

```
Value of global x is 45
Value of local x is 10
```

Points to Remember:

- A large program can typically be split into smaller sized blocks called as functions.
- Functions can be classified into Pre-defined or Built-in or Library Functions and User-defined Functions.
- User-defined functions are created by the user.
- The void function tells the compiler that the function returns nothing.
- The return statement returns a value to the calling function and transfers the program control back to the calling function.
- The default return type of a function in C++ is of type int.
- A function that calls itself is known as recursive function.
- Scope refers to the accessibility of a variable.
- There are four types of Scopes. They are: Local scope, Function scope, File scope and Class scope.
- The scope operator (::) reveals the hidden scope of a variable.