MODULE - II

Functions, Objects and Classes in C++

MODULE 2

Functions, Objects and Classes in C++

Module Description

The main goal of studying Functions in C++ is to understand the functions of syntax and to write larger programs into smaller modules to achieve some specific task. This module explains the function definition, declaration and also how to call those functions.

By the end of this module, students will learn user-defined functions aid library functions. They learn to write some sample programs using functions. They will also learn function overloading, default arguments, inline functions and storage classes. In addition to these skills students are also able to intelligently compile and execute the program and debug the errors.

Chapter 2.1

Functions in C++

Chapter 2.2

Classes and Objects

Chapter Table of Contents

Chapter 2.1

Functions in C++

Aim

To equip the students with the concept of Functions in C++

Instructional Objectives

After completing this chapter, you should be able to:

- Explain User defined functions and built in functions
- Illustrate different methods of calling the function*
- Illustrate function overloading with example
- Elaborate Default arguments and inline functions
- Explain storage classes in C++

Learning Outcomes

At the end of this chapter, you are expected to:

- Differentiate between user defined functions and built-in functions
- Compare formal and actual parameters
- Analyse call by value and call by reference with program
- Analyse function overloading in C++
- Design a program to demonstrate inline function.
- Explain a code containing pointer and reference as a parameter in function
- Write a code to demonstrate storage classes

2.1.1 Introduction

A function groups a number of program statements into a unit and gives it a name. This unit can then be invoked from other parts of the program. The most important reason to use functions is to aid in the conceptual organization of a program. Another reason to use functions is to reduce program size. Any sequence of instructions that appears in a program more than once is a candidate for being made into a function. The function's code is stored in only one place in memory, even though the function is executed many times in the course of the program. Figure shows how a function is invoked from different sections of a program.

Every C++ program has at least one function, which is **main () and** all the most trivial programs can define additional functions. Functions in C++ (and C) are similar to subroutines and procedures in various other languages.

A function **declaration** instructs or tells the compiler about a name of the function, return type and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call.

For example, function **strcat ()** to concatenate two strings, function **memcpy ()** to copy one memory location to another location and many more functions.

A function is knows as with various names like a method or a sub-routine or a procedure etc.

2.1.2 Functions in C++

Functions are used to provide modularity to a program. Creating an application using function makes it easier to understand, edit, check errors etc. A C++ program can contain any number of functions according to the needs.

(i) Function Declaration

Function declarations are also called prototypes, since they provide a model or blueprint for the function. They tell the compiler, "A function that looks like this is coming up later in the program". The actual body of the function can be defined separately. Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file before main (). A function declaration has the following syntax:

```
return_type function_name (parameter list);
```

- **Return-type:** This suggests what the function will return. It can be int, char, some pointer or even a class object. There can be a function which does not return anything. They are mentioned with void keyword.

- **Function Name:** It is the name of the function. A function is called using the function name.

- **Parameter list:** They are variables to hold values of arguments passed while function is called. A function may or may not contain parameter list, only their type is required.

  For example, void star ();

The declaration tells the compiler that at some later point we plan to present a function called star. The keyword void specifies that the function has no return value and the empty parentheses indicate that it takes no arguments. (One can also use the keyword void in parentheses to indicate that the function takes no arguments, as is often done in C. Leaving them empty is the more common practice in C++.)

The function declaration is terminated with a semicolon. It is a complete statement in itself.

(ii) Function Definition

The function can be defined using the following syntax

```
Return-type function-name (parameters)
{
   // function-body
}
```

- **Return-type:** Return-type suggests what the function will return. It can be int, char, some pointer or even a class object. There can be functions which do not return anything; they are mentioned with void keyword.

- **Function Name:** Function Name is the name of the function.

- **Parameters:** Parameters are variables to hold values of arguments passed while function is called. A function may or may not contain parameter list.

```
void sum(int p, int q)
{
   int r;
   r = p + q;
   cout << r;
```

```
}
int main()
{
   int x = 10;
   int y = 20;
   sum (x, y);
}
```

Here, **x** and **y** are sent as arguments and **p** and **q** are parameters which will hold values of x and y to perform required operation inside function.

- **Function body:** It is the part where the code statements are written.

  In programming, function refers to a segment that group's code to perform a specific task. Depending on whether a function is predefined or created by programmer. There are **two** types of function:

  1. Library Function or Built-in functions
  2. User-defined Function

(iii) Built-in Functions

Library functions are the built-in function in C++ programming. Programmer can use library function by invoking function directly. They don't need to write it themselves.

For example, Library Function

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    double n, squareRoot;
    cout<<"Enter a number: ";
    cin>>n;
  /* sqrt() is a library function to calculate square root */
    squareRoot = sqrt(n);
    cout<<"Square root of "<<n<<" = "<<squareRoot;
    return 0;
}
```

Output:

Enter a number

25

Square root of 25 = 5.00

(iv) User Defined Functions

C++ allows programmer to define their own function. A user-defined function group's code to perform a specific task. That group of code is given a name (identifier). When that function is invoked from any part of program, it executes the codes defined in the body of function.

The working of a user-defined function is depicted in the Figure 2.1.1 below.

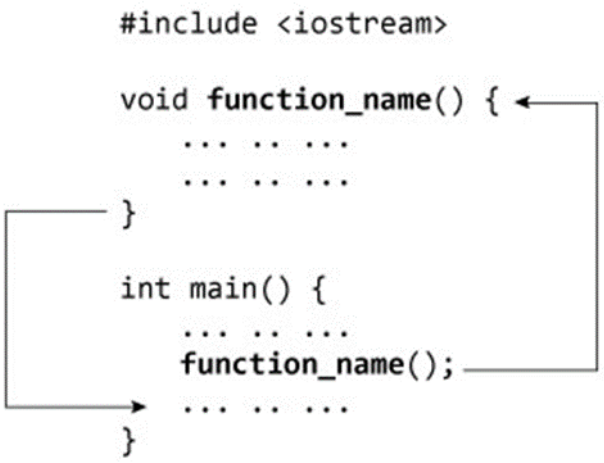

Figure 2.1.1: Working of Functions

Consider the figure above. When a program begins running, the system calls the main () function. The system starts by executing the codes from the main () function. When control of program reaches function_name () inside main (), it moves to void function_name (). All codes inside the void function_name () is executed. Control of the program moves to code right after function_name () inside main () as shown in the figure above.

For example,

Consider a C++ program to add two integers. Make a function add () to add integers and display sum in main () function.

```
# include <iostream>
using namespace std;

int add(int, int);            //Function prototype(declaration)

int main()
{
    int n1, n2, sum;
    cout<<"Enters two numbers to add: ";
    cin>>n1>>n2;
```

```
    sum = add(n1,n2);         //Function call
    cout<<"Sum = "<<sum;
    return 0;
}
int add(int a,int b)
{           //Function declarator
    int add;
    add = a+b;
    return add;                 //Return statement
}
```

Output:

Enters two integers:

8

-4

Sum = 4

(v) Calling the function

Functions are called by their names. If the function is without argument, it can be called directly using its name. But for functions with arguments. Function body is written in its definition. Let us understand this with the help of an ***example***.

```
#include < iostream>
using namespace std;
int sum (int x, int y);   //declaring function
int main()
{
   int a = 10;
   int b = 20;
   int c = sum (a, b);   //calling function
   cout << c;
}
int sum (int x, int y)   //defining function
{
   return (x + y);
}
```

Here, initially the function is **declared**, without body. Then inside the main () function it is called and the summation of two values is returned by the function. Thereby z stores the total value. Then, at last, the function is **defined**, where the body of function is mentioned. We can also, declare and define the function together, but then it should be done so before it is called.

There are two types of parameters in a function:

1. Actual Parameter
2. Formal Parameter

The below table highlights the difference between Actual and Formal Parameters

Table 2.1.1: Actual Parameter vs. Formal Parameter

Actual Parameters	Formal Parameters
Actual parameters are the actual information or exact values of variable or expression	Formal parameters access the information using formal parameters
Calling functions pass actual values or arguments to called functions	Called functions access the information using formal parameters
Actual parameters are in calling function	Formal parameters are in function header
Actual parameters should be same as you declare in the function.	Formal arguments must be declared by the same name or by different names in function definition but the data types should be the same in both calling function and called function

Example:
`#include <iostream.h>`
`void main ()`
`{`
`int a,b;`
`void output (int a, int b) ;`
`/ / function declaration`
`_____`
`_____`
`output (a , b)` `;/ / a and b are the` **`actual parameters`**
	Example:
	`#include <iostream.h>`
	`void main ()`
	`{`
	`int a,b;`
	`void output (int a, int b) ;`
	`output (a , b) ;`
	`}`
	`void output (int x, int y) / / formal parameters`
	`{`
	`/ /body of the function`
	`}`

(vi) Different Methods of Calling the Function

Two different methods of calling the function are:

1. Call by Value
2. Call by Reference

Call by value:

The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function swap () definition as follows.

```
// function definition to swap the values.
void swap(int x, int y)
{
   int temp;

   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */
     return;
}
```

Now, let us call the function **swap()** by passing actual values as in the following ***example:***

```
#include <iostream>
using namespace std;
 // function declaration
void swap(int x, int y);
 int main ()
{
   // local variable declaration:
   int a = 100;
   int b = 200;
  cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;
    // calling a function to swap the values.
   swap(a, b);
    cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;
    return 0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result:

Before swap, value of a: 100

Before swap, value of b: 200

After swap, value of a: 100

After swap, value of b: 200

Which shows that there is no change in the values though they had been changed inside the function.

Call by reference:

The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the past argument.

To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly one need to declare the function parameters as reference types. In the following function **swap (),** which exchanges the values of the two integer variables pointed to by its arguments.

```
// function definition to swap the values.
void swap(int &x, int &y)
{
   int temp;
   temp = x; /* save the value at address x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */
   return;
}
```

For now, let us call the function swap() by passing values by reference as in the following ***example:***

```
#include <iostream>
using namespace std;

// function declaration
void swap(int &x, int &y);
int main ()
```

```
{
   // local variable declaration:
   int a = 100;
   int b = 200;
   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;
   /* calling a function to swap the values using variable reference.*/
   swap(a, b);
   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;
    return 0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result:

Before swap, value of a: 100

Before swap, value of b: 200

After swap, value of a: 200

After swap, value of b: 100

(vii) Using Reference as Parameter

Reference parameters are useful in two cases:

- **Change values:** Use a reference parameter when you need to change the value of an actual parameter variable in the call. When a function computes only one value it is better to return the value with the return statement. However, if a function produces more than one value, it is common to use reference parameters to return values, or a combination of the return value and reference parameters.

- **Efficiency**: This is used to pass large structures more effectively. This is widely used for passing struct's or class objects. If no changes are made to the parameter, it will be declared const.

When you declare a reference parameter, the function call will pass the memory address location of the actual parameter, instead of copying the parameter value into the formal parameter. To indicate a reference parameter, an ampersand (&) is written in the function prototype and header after the parameter type name.

For example,

```
void assign(int& to, int from)
{
   to = from;  // Will change the actual parameter in the call.
}
```

Has two parameters, **to** is a reference parameter as indicated by the ampersand and **from** is a value parameter. This ampersand must be in both the prototype and the function header.

(viii) Pointer as Parameter

C++ allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following is a simple ***example*** where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:

```
#include <iostream>
#include <ctime>
using namespace std;
void getSeconds(unsigned long *par);
int main ()
{
   unsigned long sec;
   getSeconds( &sec );
   // print the actual value
   cout << "Number of seconds :" << sec << endl;
   return 0;
}
void getSeconds(unsigned long *par)
{
   // get the current number of seconds
   *par = time( NULL );
   return;
}
```

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

Number of seconds: 1294450468

The function which can accept a pointer, can also accept an array as shown in the following ***example:***

```
#include <iostream>
using namespace std;
// function declaration:
```

```
double getAverage(int *arr, int size);
int main ()
{
   // an int array with 5 elements.
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;
   // pass pointer to the array as an argument.
   avg = getAverage( balance, 5 ) ;
   // output the returned value
   cout << "Average value is: " << avg << endl;
  return 0;
}
double getAverage(int *arr, int size)
{
  int    i, sum = 0;
  double avg;
  for (i = 0; i < size; ++i)
  {
    sum += arr[i];
   }
  avg = double(sum) / size;
  return avg;
}
```

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

Average value is: 214.4

Did you know?

Visualization cannot be done for static member function.

Self-assessment Questions

1) After the function is executed, the control passes back to the _____.

   a) calling function b) called function

   c) main function d) library function

2) A function that uses another function is known as_____.

3) The calling function must pass parameters to the called function.

   a) TRUE b) FALSE

4) Function header is used to identify the function.

   a) TRUE b) FALSE

2.1.3 Overloaded Functions

If any class has multiple functions with same names but different parameters, then they are said to be overloaded. Function overloading allows you to use the same name for different functions, to perform, either same or different functions in the same class.

Function overloading is usually used to enhance the readability of the program. If you have to perform one single operation with different number or types of arguments, then you can simply overload the function. There are two ways to overload a function by changing.

- Different number of arguments.
- Different types of argument.

(i) Different Number of Arguments

In this type of function overloading, we define two functions with same names but different number of parameters of the same type.

For example, in the below mentioned program we have made two sum () functions to return sum of two and three integers.

```
int sum (int x, int y)
{
   cout << x+y;
}
int sum(int x, int y, int z)
{
   cout << x+y+z;
}
```

Here sum() function is overloaded, to have two and three arguments based on which sum() function will be called, depending on the number of arguments.

```
int main()
{
    sum (10,20);  // sum() with 2 parameter will be called
    sum(10,20,30);  //sum() with 3 parameter will be called
}
```

(ii) Different Types of Arguments

In this type of overloading we define two or more functions with same name and same number of parameters, where in the type of parameter will be different.

For example, in this program, we have two sum() function. First one gets two integer arguments and the second one gets two double arguments.

```
int sum(int x,int y)
{
   cout<< x+y;
}
double sum(double x,double y)
{
   cout << x+y;
}
int main()
{
   sum (10,20);
   sum(10.5,20.5);
}
```

Self-assessment Questions

5) What is the output of the program?

```
#include <iostream>
   using namespace std;
    void print(int i)
    {
        cout << i;
    }
    void print(double  f)
    {
        cout << f;
    }
    int main(void)
    {
        print(5);
        print(500.263);
        return 0;
    }
```

a) 5500.263
b) 500.2635
c) 500.263
d) 500.222

6) Overloaded functions are

a) Very long functions that can hardly run
b) One function containing another one or more functions inside it.
c) Two or more functions with the same name but different number of parameters or type.
d) Very small functions

7) We cannot overload a function by the _____.

2.1.4 Inline Functions

C++ **inline** function is a powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called during a compilation.

Any change to an inline function could require all the clients of the function to be recompiled because compiler would need to replace all the codes once again. Otherwise, it will continue with old functionalities.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier if the defined function is more than a single line.

A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

```
// inliner.cpp
// demonstrates inline functions
#include <iostream>
using namespace std;
// lbstokg()
// converts pounds to kilograms
inline float lbstokg(float pounds)
{
   return 0.453592 * pounds;
}
//-----------------------------------------------------------
int main()
{
   float lbs;
   cout << "\nEnter your weight in pounds: ";
   cin >> lbs;
   cout << "Your weight in kilograms is " << lbstokg(lbs)
   << endl;
   return 0;
}
```

It is easy to make a function inline by including the keyword **inline** in the function definition. ***For example,*** Inline float lbstokg (float pounds). You should be aware that the inline keyword is actually just a request to the compiler. Sometimes the compiler will ignore the request and compile the function as a normal function. The compiler might also decide the function as lengthy to be inline.

Advantages of Inline functions

- It increases performance.
- It is better than the macros because it does type checking.

Disadvantages of Inline functions

- Compiler often generates a lot of low level code to implement inline functions.
- It is necessary to write a definition for an inline function (compared to normal function) in every module (compilation unit) that uses it. Otherwise it is not possible to compile a single module independently of all the other modules.
- If we use many inline functions then executable size will also get increased.
- Breaks the encapsulation slightly, because it exposes the internal of the objects.

Self-assessment Questions

8) What does the inline keyword do?
   a) Indicates a function declaration.
   b) Tells the compiler to use the function only within the same source code file.
   c) Causes all the function calls to be replaced by the code from the function.
   d) Allows one-line function declarations

9) Why would you want to use inline functions?
   a) To decrease the size of the resulting program.
   b) To increase the speed of the resulting program.
   c) To simplify the source code file.
   d) To remove unnecessary functions.

10) Which of the following is a limitation of inline functions?
   a) Inline functions cannot return a value.
   b) Inline functions must return a value.
   c) Inline functions must be less than ten lines.
   d) The compiler may choose to ignore an inline directive.

2.1.5 Default Arguments

When we mention a default value for a parameter while declaring a function, it is called a default argument. In this case, even if we make a call to the function without passing any value for that parameter, the function will take the default value specified.

```
sum(int x,int y=0)
{
   cout << x+y;
}
```

Here we have provided a default value for y, during function definition.

```
int main()
{
   sum(10);
   sum(10,0);
   sum(10,10);
}
```

Output: 10 10 20

First two function calls will produce exactly the same value. In the third function call, y will take 10 as value and output will become 20.

By setting a default argument, we are also overloading the function. Default arguments allows to use the same function in different situations just like the function overloading.

Rules for using default arguments

1. Only the last argument must be given a default value. You cannot have a default argument followed by a non-default argument.

   ```
   sum (int x,int y);
   sum (int x,int y=0);
   sum (int x=0,int y);  // This is Incorrect
   ```

2. If you default an argument, then you will have to default all the subsequent arguments.

   ```
   sum (int x,int y=0);
   sum (int x,int y=0,int z);  // This is incorrect
   sum (int x,int y=10,int z=10);  // Correct
   ```

3. You can give any value as a default value to an argument, which is compatible with its datatype.

Self-assessment Questions

11) A function can have a default argument followed by a non-default argument

a) True b) False

12) It will take _____ value when both user and default values are given

13) Which arguments cannot be place before by the non-default arguments?

a) Trailing arguments b) Default arguments

c) Global arguments d) No arguments

2.1.6 Storage Classes

Every variable in C++ has a type which specifies the type of data that can be stored in a variable. ***For example,*** int, float, char etc. Also, variables and objects in C++ have another feature called storage class. Storage class specifiers control two different properties: storage duration (determines how long a variable can exist) and scope (determines which part of the program can access it). There are four different storage classes, based on the storage duration and scope of variables.

- Automatic
- External
- Static
- Register

(i) Automatic

A variable which is defined inside a function is an automatic variable or local variable. Local variables are created when the function containing local variable is called and destroyed when that function returns. Local variables can only be accessed from inside a function in which it exists. Consider this ***example:***

```
#include <iostream>
using namespace std;
void test();
int main()
{
    int var = 5;   // local variable to main()
```

```
    test();
    var1 = 9;      // illegal: var1 not visible inside main()
}
void test()
{
    int var1;      // local variable to test()
    var1 = 6;
    cout<<var;     // illegal: var not visible inside test()
}
```

The variable var cannot be used inside test () and var1 cannot be used inside main () function. Keyword auto could also be used for defining local variables (it was optional but not necessary) before as: auto int var; After C++11, auto has different meaning and should not be used for defining Local variables as variables defined inside a function is a Local variable by default.

(ii) External

If a variable is defined outside any function, then that variable is called a Global variable. Any part of the program can access global variable even after the global variable declaration. If a Global variable is defined at the beginning of the listing, then global variable is visible to all the functions. Consider this ***example:***

```
/* In this example, global variable can be accessed by all functions because
   it is defined at the top of the listing.*/
#include <iostream>
using namespace std;
int c = 12;
void test();
int main()
{
    ++;c
    cout<<c<<endl;   //Output: 13
    test();
    return 0;
}
void test()
{
    ++c;
    cout<<c;     //Output: 14
}
```

In the above program, “c” is a Global variable. This variable is visible to both the functions in the above program. The memory for Global variable is set when a program starts and exists until the program ends.

(iii) Static

Keyword **static** is used for specifying static variable.

For example,

```
      ... .. ...
int main()
{
   static float a;
   ... .. ...
}
```

A **static** Local variable exists only inside a function in which it is declared (similar to local variable).The lifetime of static variable starts when the function containing static variable is called and terminates when the program ends. The main difference between local variable and static variable is that, the value of the static variable persists until the program ends. Consider this ***example:***

```
      #include <iostream>
using namespace std;
void test()
{
    static int var = 0; // var is a static variable;
    ++var;
    cout<<var<<endl;
}
int main()
{
    test();
    test();
    return 0;
}
```

Output:

1

2

In the above program, test () function is invoked three times. During the first call, **variable var** is declared as a static variable and initialized to 0. Then 1 is added to **var** which is displayed in the screen. When the function test () returns, variable **var** still exists because it is a static variable. During the second function call, no new **variable var** is created. Only var is increased by 1 and then displayed on the screen.

Output of the above program if var was not specified as static variable

1

1

(iv) Register

Keyword **register** is used for specifying the register variables. Register variables are similar to an automatic variable and exists inside that particular function only. If a program encounters register variable, it stores that variable in processor's register rather than the memory if available. Processor's register is much faster than the memory. This keyword was deprecated in C++11 and should not be used.

Self-assessment Questions

14) The default storage class of a local variable is

a) Auto
b) Static
c) Register
d) Global

15) Which variable retains its value in between function calls?

a) Auto
b) Static
c) Register
d) Global

16) Memory is allocated to a function when the function is_____.

Summary

- A function is a group of statements that performs a task. A function can be used by invoking a call to the function.
- User defined functions are functions which the users themselves have to write whereas built in functions are the library functions offered by C++.
- A function can be called by two methods; call by value and call by reference.
- A class is considered overloaded when it has multiple functions but with different parameters.
- Function overloading can be performed in two ways; different number of arguments and different types of arguments.
- Default values of the arguments are passed when invoking a function without specifying all the arguments. This is true only when the default argument values are defined for that function.
- An inline function eliminates the overhead of function call, wherein the compiler copies the executable statements of the function in the place of each function call.
- A storage class defines the visibility or scope and the lifetime of variables or functions. There are 4 different storage classes – Automatic, External, Static and Register.

Terminal Questions

1. Define functions and explain the types of functions in C++
2. Differentiate between call-by value and call-by reference with suitable examples.
3. Illustrate function overloading.
4. Explain default arguments and Inline functions
5. Explain the types of storage classes.

Answer Keys

Self-assessment Questions	
Question No.	**Answer**
1	a
2	Calling function
3	b
4	a
5	a
6	c
7	Return type
8	c
9	b
10	b
11	False
12	User
13	b
14	a
15	b
16	Defined

Activity

Activity Type: Online **Duration:** 30 Minutes

Description:

Students should be able to write functions on their own to perform some specific task like

a) To read a matrix.

b) To find the power of a given number.

c) To find the largest of three numbers.

Also, Design and execute the main program to test the above functions.

Bibliography

e-References

- learncpp.com,(2016). *Function Pointers.* Retrieved 7 April, 2016 from http://www.learncpp.com/cpp-tutorial/78-function-pointers/
- cplusplus.com,(2016). *Function Pointers - C++.* Retrieved 7 April, 2016 from http://www.cplusplus.com/forum/beginner/18060/
- tutorialspoint.com,(2016). *Passing pointers to functions in C++* .Retrieved 7 April, 2016 from http://www.tutorialspoint.com/cplusplus/cpp_passing_pointers_to_functions.htm
- studytonight.com,(2016). *Functions in C++ - Declaration, Definition and Call* . Retrieved 7 April, 2016 from http://www.studytonight.com/cpp/functions-in-cpp

External Resources

- Balaguruswamy, E. (2008). *Object Oriented Programming with C++*. Tata McGraw Hill Publications.
- Lippman. (2006). *C++ Primer* (3rd ed.). Pearson Education.
- Robert, L. (2006). *Object Oriented Programming in C++* (3rd ed.). Galgotia Publications References.
- Schildt, H., & Kanetkar, Y. (2010). *C++ completer* (1st ed.). Tata McGraw Hill.
- Strousstrup. (2005). *The C++ Programming Language* (3rd ed.). Pearson Publications.

Video Links

Topic	Link
Functions in C++	https://www.youtube.com/watch?v=VTF0KQga31k
Overloaded function	https://www.youtube.com/watch?v=wQEByoBQG-o
Inline functions	https://www.youtube.com/watch?v=TGwl3tJYFRg
Default arguments	https://www.youtube.com/watch?v=V40tv3LKKjY
Storage classes	https://www.youtube.com/watch?v=1zlvsCeju1I

Notes: