Unit-III

Functions:

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, 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 known with various names like a method or a sub-routine or a procedure etc.

Defining a Function

The general form of a C++ function definition is as follows −

```
return_type function_name( parameter list ) {
   body of the function}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Example

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and return the biggest of both −

```
// function returning the max between two numbers
int max(int num1, int num2) {
   // local variable declaration
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;
   return result;}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

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

For the above defined function max(), following is the function declaration −

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration −

```
int max(int, int);
```

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 calling the function.

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example −

```
#include <iostream>
using namespace std;
// function declaration
int max(int num1, int num2);
int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   int ret;
   // calling a function to get max value.
   ret = max(a, b);
   cout << "Max value is : " << ret << endl;
    return 0;}
 // function returning the max between two numbers
int max(int num1, int num2) {
   // local variable declaration
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;
    return result; }
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result −

Max value is : 200

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function −

SNo	Call Type & Description
1	Call by Value This method 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.
2	Call by Pointer This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
3	Call by Reference This method 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 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 and above mentioned example while calling max() function used the same method.

Default Values for Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example −

```
#include <iostream>
using namespace std;
int sum(int a, int b = 20) {
   int result;
   result = a + b;
    return (result);}
int main () {
   // local variable declaration:
   int a = 100;
```

```
int b = 200;
int result;
 // calling a function to add the values.
result = sum(a, b);
cout << "Total value is :" << result << endl;
// calling a function again as follows.
result = sum(a);
cout << "Total value is :" << result << endl;
return 0;}
```

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

```
Total value is :300
Total value is :120
```

Arrays:

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows −

```
type arrayName [ arraySize ];
```

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement −

```
double balance[10];
```

Initializing Arrays

You can initialize C++ array elements either one by one or using a single statement as follows −

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array −

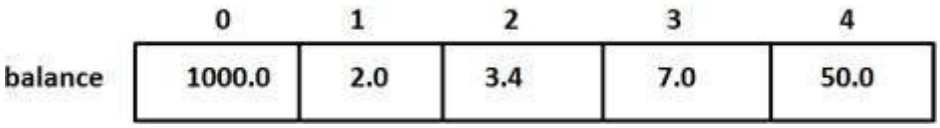
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representaion of the same array we discussed above −



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays −

```
#include <iostream>
using namespace std;
#include <iomanip>
using std::setw;
int main () {
  int n[ 10 ]; // n is an array of 10 integers
   // initialize elements of array n to 0
  for ( int i = 0; i < 10; i++ ) {
     n[ i ] = i + 100; // set element at location i to i + 100   }
  cout << "Element" << setw( 13 ) << "Value" << endl;
  // output each array element's value
  for ( int j = 0; j < 10; j++ ) {
     cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;   }
  return 0;}
```

This program makes use of **setw()** function to format the output. When the above code is compiled and executed, it produces the following result −

```
Element       Value
      0         100
      1         101
      2         102
      3         103
      4         104
      5         105
      6         106
      7         107
      8         108
      9         109
```

Classes and Instances: Defining Classes in Objected Oriented Language

C++ Class Definitions

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows −

```
class Box {
  public:
    double length;   // Length of a box
    double breadth;  // Breadth of a box
    double height;   // Height of a box};
```

The keyword **public** determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box −

```
Box Box1;        // Declare Box1 of type Box
Box Box2;        // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing the Data Members

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear −

```
#include <iostream>
using namespace std;
class Box {
  public:
    double length;   // Length of a box
    double breadth;  // Breadth of a box
    double height;   // Height of a box};
int main() {
  Box Box1;       // Declare Box1 of type Box
  Box Box2;       // Declare Box2 of type Box
  double volume = 0.0;     // Store the volume of a box here
   // box 1 specification
  Box1.height = 5.0;
  Box1.length = 6.0;
  Box1.breadth = 7.0;
  // box 2 specification
  Box2.height = 10.0;
  Box2.length = 12.0;
  Box2.breadth = 13.0;
  // volume of box 1
  volume = Box1.height * Box1.length * Box1.breadth;
  cout << "Volume of Box1 : " << volume <<endl;
```

```
 // volume of box 2
 volume = Box2.height * Box2.length * Box2.breadth;
 cout << "Volume of Box2 : " << volume <<endl;
 return 0;}
```

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

Volume of Box1 : 210
Volume of Box2 : 1560

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

Classes and Objects in Detail

So far, you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects which we will discuss in various sub-sections listed below −

Building and Destroying Instances (Constructors and Destructors):

While programming, sometimes there might be the need for the initialization of data members and member functions of the objects before performing any operations on them. Data members are the variables that are declared in any class by using any fundamental data types (like int, char, float, etc) or derived data types (like class, structure, pointer, etc). The functions which are defined inside the class definition are known as member functions.

Suppose you are developing a game. In that game, each time a new player registers, we need to assign their initial location, health, acceleration and certain other quantities to some default value.

This can be done by defining separate functions for each quantity which will assign the quantities to the required default values. For this, we need to call a list of functions every time a new player registers. Now, this process can become lengthy and complicated.

What if we can assign the quantities along with the declaration of the new player automatically? A constructor can help do this in a better and simpler way.

Moreover, when the player deletes his account, we need to deallocate the memory assigned to him. This can be done using a destructor.

What is a Constructor in C++?

Constructor in C++ is a special member function of a class whose task is to initialize the object of the class, it's special because it has the same name as that of the class. It is called a constructor because it constructs the value of data members at the time of object initialization. The compiler invokes the constructor whenever an object is created. Since a constructor defines the value to a data member, it has no return type.

Syntax of Constructor:

```
class scaler {
  public:
    scaler() { //constructor
      // constructor body    }};
```

Characteristics of Constructors in C++

- A constructor can be made public, private, or protected as per the design of our program. Constructors are mostly made public, as public methods are accessible from everywhere, thus allowing us to create the object of the class anywhere in the code. **Note:** When a constructor is made private, other classes cannot create instances of the class. This is used when there is no need for object creation. Such a case arises when the class only contains static member functions(i.e the functions which are independent of any object of the class and can be accessed using class name with scope resolution operator i.e '::').
- A constructor in C++ cannot be inherited. Although a derived class can call the base class constructor. A derived class(i.e child class) contains all members and member functions(including constructors) of the base class.
- Constructor functions are not inherited and their addresses cannot be referenced.
- The constructor in C++ cannot be virtual. A virtual table(also called vtable) is made for each class having one or more virtual functions. Virtual functions ensure that the correct function is called for an object regardless of the type of reference used for function call. Whenever an object is created of such a class it contains a 'virtual-pointer' which points to the base of the corresponding vtable. Whenever there is a virtual function call, the vtable is used to refer to the function address. But when a constructor of a class is executed there is no virtual table created yet in the memory, which means no virtual pointer is defined yet. As a result, a constructor cannot be declared virtual.

Types of Constructors in C++

There are 4 types of constructors in C++:

1. Default Constructors
2. Parameterized Constructors
3. Copy Constructors
4. Dynamic Constructors

1. Default Constructor:

A constructor which does not receive any parameters is called a Default Constructor, or a 'Zero Argument Constructor'.

In the default constructor, every object of the class is initialized with the same set of values.

Even if a constructor is not defined explicitly, the compiler will provide a default constructor implicitly.

Syntax:

```
Employee :: Employee()   //default constructor without any arguments
```

Example:

```
class Employee {
  public:
    int age;
    //default constructor
    Employee() {
      //data member is defined with the help of default constructor
```

```
    age = 50;    }
};
int main() {
  //object of class Employee declared
  Employee e1;
  //prints value assigned by default   constructor
  cout << e1.age;
  return 0;}
```

Output:

```
50
```

Note: Default constructor provided by the compiler will be called which will initialize the object data members to default value, that will be 0 or any random integer value in the example provided below.

Example:

Considering the previous example, the default constructor(i.e Employee() in Employee class definition) defined by the programmer assigns the data member age to a value of 50.

But in the given example here, as the data member age is not assigned to any value, the compiler calls the default constructor and age is initialized to 0 or unpredictable garbage values.

```
class Employee {
  public:
    int age;
    //default constructor not defined.
    //Compiler calls default constructor};
int main() {
  Employee e1; //object e1 declared
  cout << e1.age;
  return 0;}
```

Output:

```
0
```

2. Parameterized Constructor

In a default constructor, it is not possible to initialize different objects with different initial values. What if we need to pass arguments to constructors which are needed to initialize an object? In order to solve this issue, there is a different type of constructor called Parameterized Constructor.

A Parameterized Constructor is a constructor that can accept one or more arguments. This helps programmers to assign varied initial values to an object at the time of creation.

Example:

```
#include <iostream>
using namespace std;
class Employee {
  public:
```

```
   int age;
   Employee(int x) { // parameterized constructor
     age = x; //age assigned to value passed as argument while object declaration    }
};
int main() {
  Employee c1(40); //object c1 declared with argument 40 which gets assigned to age
  Employee c2(30);
  Employee c3(50);
  cout << c1.age << "\n";
  cout << c2.age << "\n";
  cout << c3.age << "\n";
  return 0;}
```

Output:

```
40
30
50
```

3. Copy Constructor

Copy constructor is a type of constructor which is used to create a copy of an already existing object of a class type. The compiler provides a default Copy Constructor to all the classes. Whenever there is a need for an object with the same values for data members as of an already existing object, a copy constructor comes into picture. A copy constructor is invoked when an existing object is passed as a parameter.

Syntax:

```
Employee :: Employee(Employee &ptr)       //copy constructor
```

Example:

```
#include<iostream>
using namespace std;
class Employee {
  private:
    int salary, experience; //data members
  public:
    Employee(int x1, int y1) { //parameterized constructor
      salary = x1;
      experience = y1;    }
    /* Copy constructor */
    Employee(const Employee &new_employee) {
      salary = new_employee.salary;
      experience = new_employee.experience;    }
    void display() {
      cout << "Salary: " << salary << endl;
      cout << "Years of experience: " << experience << endl;    }
};
/* main function */
int main() {
  Employee employee1(34000, 2); // Parameterized constructor
  Employee employee2 = employee1; // Copy constructor
```

```
  cout << "Employee1 using parameterized constructor : \n";
  employee1.display();
  cout << "Employee2 using copy constructor : \n";
  employee2.display();
  return 0;}
```

Output:

```
Employee1 using parameterized constructor :
Salary: 34000
Years of experience: 2
Employee2 using copy constructor :
Salary: 34000
Years of experience: 2
```

Explanation:

In this example, object employee1 of class Employee is declared in the first line of main() function. The data members salary, experience for object employee1 are assigned 34000 and 2 respectively with the help of a parameterized constructor which is invoked automatically. When object employee2 is declared in the second line of main() function, object employee1 is assigned to employee2 which invokes the copy constructor as the argument here is an object itself. As a result, the data members salary,experience of object employee2 are assigned to values possessed by salary,experience data members of object employee1(i.e. 34000, 2) respectively.

4. Dynamic Constructor

When allocation of memory is done dynamically(i.e. Memory is allocated to variables at run-time of the program rather than at compile-time) using dynamic memory allocator new in a constructor, it is known as dynamic constructor. By using this, we can dynamically initialize the objects.

```
#include <iostream>
using namespace std;
class Employee {
  int* due_projects;
  public:
    // default constructor
    Employee() {
      // allocating memory at run time
      due_projects = new int;
      *due_projects = 0;    }
    // parameterized constructor
    Employee(int x) {
      due_projects = new int;
      *due_projects = x;    }
      void display() {
      cout << *due_projects << endl;    }
};
//main function
int main() {
  // default constructor would be called
  Employee employee1 = Employee();
  cout << "Due projects for employee1:\n";
  employee1.display();
```

```
 // parameterized constructor would be called
 Employee employee2 = Employee(10);
 cout << "Due projects for employee2:\n";
 employee2.display();
 return 0;}
```

Output:

```
Due projects for employee1:
0
Due projects for employee2:
10
```

Explanation:

Here, integer type pointer variable is declared in class which is assigned memory dynamically when the constructor is called. When we create object employee1 of class Employee in the first line of main() function, the default constructor(i.e. Employee() in class Employee definition) is called automatically and memory is assigned dynamically to the pointer type variable(i.e. *due_projects) and initialized with value 0.

And similarly, when employee2 is created in the third line of main() function, the parameterized constructor(i.e. Employee(int x) in class definition) is called and memory is assigned dynamically.

What is a Destructor in C++?

Destructor is a member function that is instantaneously called whenever an object is destroyed. The destructor is called automatically by the compiler when the object goes out of scope i.e. when a function ends the local objects created within it also gets destroyed with it. The destructor has the same name as the class name, but the name is preceded by a tilde(~). A destructor has no return type and receives no parameters.

Syntax of Destructor:

```
class scaler {
  public:
    scaler(); //constructor
    ~scaler(); //destructor};
```

Characteristics of a Destructor in C++

- A destructor deallocates memory occupied by the object when it's deleted.
- A destructor cannot be overloaded. In function overloading, functions are declared with the same name in the same scope, except that each function has a different number of arguments and different definitions. But in a class, there is always a single destructor and it does not accept any parameters, hence, a destructor cannot be overloaded.
- A destructor is always called in the reverse order of the constructor. In C++, variables and objects are allocated on the Stack. The Stack follows LIFO (Last-In-First-Out) pattern. So, the deallocation of memory and destruction is always carried out in the reverse order of allocation and construction. This can be seen in code below.
- A destructor can be written anywhere in the class definition. But to bring an amount of order to the code, a destructor is always defined at the end of the class definition.

Implementation of Constructors and Destructors in C++

S. No.	Constructors	Destructors

```
#include <iostream>
using namespace std;
class Department {
  public:
    Department() {
      //constructor is defined
      cout << "Constructor Invoked for Department class" << endl;    }
      ~Department() {
      //destructor is defined
      cout << "Destructor Invoked for Department class" << endl;    }
};
class Employee {

  public:
    Employee() {
      //constructor is defined
      cout << "Constructor Invoked for Employee class" << endl;    }

    ~Employee() {
      //destructor is defined
      cout << "Destructor Invoked for Employee class" << endl;    }
};
int main(void) {
  Department d1; //creating an object of Department
  Employee e2; //creating an object of Employee
  return 0;}
```

Output:

```
Constructor Invoked for Department class
Constructor Invoked for Employee class
Destructor Invoked for Employee class
Destructor Invoked for Department class
```

Explanation:

When an object named d1 is created in the first line of main() i.e (Department d1), it's constructor is automatically invoked during the creation of the object. As a result, the first line of output "Constructor Invoked for Department class" is printed. Similarly, when the e2 object of Employee class is created in the second line of main() i.e (Employee e2), the constructor corresponding to e2 is invoked automatically by the compiler and "Constructor Invoked for Employee class" is printed.

A destructor is always called in the reverse order as that of a constructor. When the scope of the main function ends, the destructor corresponding to object e2 is invoked first. This leads to printing "Destructor Invoked for Employee class". Lastly, the destructor corresponding to object d1 is called

1	Constructor is invoked to initialize the object of a class by the compiler.	Destructor is invoked when the instance is destroyed.
2	Declared as Class_Name(arguments if any){//constructor body}	Declared as~Class_Name(){};
3	Constructor can receive parameters.	Destructor cannot accept any parameters.
4	Constructor is used to initialize an object of the class and assign values to data members corresponding to the class.	While destructor is used to deallocate the memory of an object of a class.
5	There can be multiple constructors for the same class.	In a class, there is always a single destructor.
6	Constructor can be overloaded.	Destructor can't be overloaded.

Modifiers:

In C++, special words(called **modifiers**) can be used to modify the meaning of the predefined built-in data types and expand them to a much larger set. There are four datatype modifiers in C++, they are:

1. long
2. short
3. signed
4. unsigned

The above mentioned modifiers can be used along with built in datatypes to make them more precise and even expand their range.

Below mentioned are some important points you must know about the modifiers,

- **long** and **short** modify the maximum and minimum values that a data type will hold.
- A plain int must have a minimum size of **short**.
- Size hierarchy : short int < int < long int
- Size hierarchy for floating point numbers is : float < double < long double
- **long float** is not a legal type and there are no **short floating point** numbers.
- **Signed** types includes both positive and negative numbers and is the default type.
- **Unsigned**, numbers are always without any sign, that is always positive.

Friend Functions:

Friend Class A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example, a LinkedList class may be allowed to access private members of Node.

```
class Node {
private:
   int key;
   Node* next;
   /* Other members of Node Class */
    // Now class  LinkedList can
   // access private members of Node
```

```
    friend class LinkedList;};
```

Friend Function Like friend class, a friend function can be given a special grant to access private and protected members. A friend function can be:
a) A member of another class
b) A global function

```
class Node {
private:
   int key;
   Node* next;

   /* Other members of Node Class */
   friend int LinkedList::search();
   // Only search() of linkedList
   // can access internal members
};
```

Following are some important points about friend functions and classes:
1) Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.
2) Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.
3) Friendship is not inherited (See this for more details)
4) The concept of friends is not there in Java.

A simple and complete C++ program to demonstrate friend Class

```
 #include <iostream>
class A {
private:
   int a;
public:
   A() { a = 0; }
   friend class B; // Friend Class};
class B {
private:
   int b;
public:
   void showA(A& x)    {
      // Since B is friend of A, it can access
      // private members of A
      std::cout << "A::a=" << x.a;    }
};
int main(){
   A a;
   B b;
   b.showA(a);
   return 0;}
```

Output:

A::a=0

Inline Functions:

C++ **inline** function is 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 at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

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 in case defined function is more than a line.

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

Following is an example, which makes use of inline function to return max of two numbers −

```
#include <iostream>
using namespace std;
inline int Max(int x, int y) {
   return (x > y)? x : y;}
// Main function for the program
int main() {
   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;
   return 0;}
```

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

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

String Handling Functions:

C++ provides following two types of string representations −

- The C-style character string.
- The string class type introduced with Standard C++.

The C-Style Character String

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

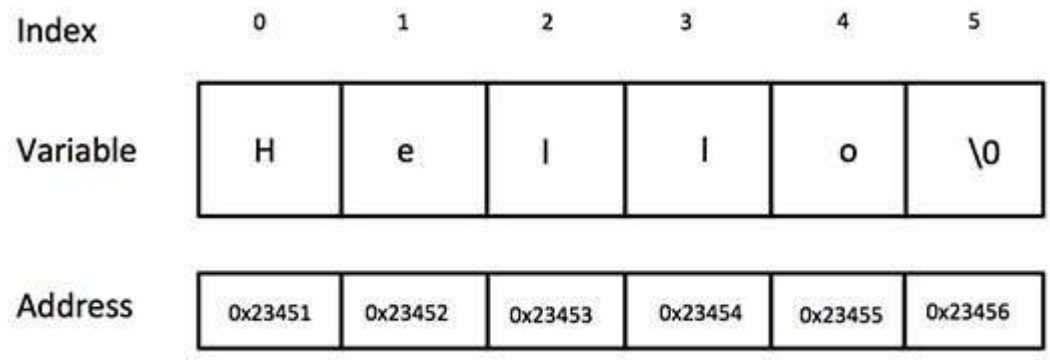
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization, then you can write the above statement as follows −

char greeting[] = "Hello";

Following is the memory presentation of above defined string in C/C++ −



Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string −

```
#include <iostream>
using namespace std;
int main () {
  char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
  cout << "Greeting message: ";
  cout << greeting << endl;
  return 0;}
```

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

Greeting message: Hello

C++ supports a wide range of functions that manipulate null-terminated strings −

Sr.No	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.

5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

Following example makes use of few of the above-mentioned functions −

```
#include <iostream>
#include <cstring>
using namespace std;
int main () {
   char str1[10] = "Hello";
   char str2[10] = "World";
   char str3[10];
   int  len ;
   // copy str1 into str3
   strcpy( str3, str1);
   cout << "strcpy( str3, str1) : " << str3 << endl;
   // concatenates str1 and str2
   strcat( str1, str2);
   cout << "strcat( str1, str2): " << str1 << endl;
   // total lenghth of str1 after concatenation
   len = strlen(str1);
   cout << "strlen(str1) : " << len << endl;
   return 0;}
```

When the above code is compiled and executed, it produces result something as follows −

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

Unit-IV

Data Encapsulation:

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public** members. By default, all items defined in a class are private. For example −

```
class Box {
  public:
    double getVolume(void) {
      return length * breadth * height;
    }
  private:
    double length;      // Length of a box
    double breadth;     // Breadth of a box
    double height;      // Height of a box };
```

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

Data Encapsulation Example

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example −

```
#include <iostream>
using namespace std;
class Adder {
  public:
    // constructor
    Adder(int i = 0) {
      total = i;      }
     // interface to outside world
    void addNum(int number) {
      total += number;      }
    // interface to outside world
    int getTotal() {
      return total;      };
  private:
    // hidden data from outside world
    int total;};
```

```
int main() {
   Adder a;
   a.addNum(10);
   a.addNum(20);
   a.addNum(30);
   cout << "Total " << a.getTotal() <<endl;
   return 0;}
```

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

```
Total 60
```

Polymorphism:

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes −

```
#include <iostream>
using namespace std;
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;      }
      int area() {
         cout << "Parent class area :" <<endl;
         return 0;      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int a = 0, int b = 0):Shape(a, b) { }
        int area () {
         cout << "Rectangle class area :" <<endl;
         return (width * height);       }
};
class Triangle: public Shape {
   public:
      Triangle( int a = 0, int b = 0):Shape(a, b) { }
       int area () {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);       }
};
// Main function for the program
int main() {
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);
   // store the address of Rectangle
   shape = &rec;
   // call rectangle area.
```

```
  shape->area();
  // store the address of Triangle
  shape = &tri;
   // call triangle area.
  shape->area();
   return 0;}
```

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

```
Parent class area :
Parent class area :
```

Operator Overloading:

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

Function Overloading in C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types −

```
#include <iostream>
using namespace std;
 class printData {
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;      }
      void print(double  f) {
        cout << "Printing float: " << f << endl;      }
      void print(char* c) {
        cout << "Printing character: " << c << endl;      }
};
int main(void) {
   printData pd;
    // Call print to print integer
   pd.print(5);
      // Call print to print float
   pd.print(500.263);
      // Call print to print character
   pd.print("Hello C++");
    return 0;
}
```

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

Printing int: 5
Printing float: 500.263
Printing character: Hello C++

Operators Overloading in C++

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Box operator+(const Box&);

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows −

Box operator+(const Box&, const Box&);

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below −

```
#include <iostream>
using namespace std;
class Box {
  public:
    double getVolume(void) {
      return length * breadth * height;     }
    void setLength( double len ) {
      length = len;     }
    void setBreadth( double bre ) {
      breadth = bre;     }
    void setHeight( double hei ) {
      height = hei;     }
    // Overload + operator to add two Box objects.
    Box operator+(const Box& b) {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;     }
  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box};
// Main function for the program
int main() {
  Box Box1;              // Declare Box1 of type Box
  Box Box2;              // Declare Box2 of type Box
  Box Box3;              // Declare Box3 of type Box
  double volume = 0.0;    // Store the volume of a box here
  // box 1 specification
  Box1.setLength(6.0);
```

```
  Box1.setBreadth(7.0);
  Box1.setHeight(5.0);
  // box 2 specification
  Box2.setLength(12.0);
  Box2.setBreadth(13.0);
  Box2.setHeight(10.0);
  // volume of box 1
  volume = Box1.getVolume();
  cout << "Volume of Box1 : " << volume <<endl;
  // volume of box 2
  volume = Box2.getVolume();
  cout << "Volume of Box2 : " << volume <<endl;
  // Add two object as follows:
  Box3 = Box1 + Box2;
  // volume of box 3
  volume = Box3.getVolume();
  cout << "Volume of Box3 : " << volume <<endl;
  return 0;}
```

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

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Overloadable/Non-overloadableOperators

Following is the list of operators which can be overloaded −

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Function Overloading:

Function overloading is a C++ programming feature that allows us to have more than one function having same name but different parameter list, when I say parameter list, it means the data type and sequence of the parameters, for example the parameters list of a function myfuncn(int a, float b) is (int, float) which is different from the function myfuncn(float a, int b) parameter list (float, int). Function overloading is a compile-time polymorphism.
Now that we know what is parameter list lets see the rules of overloading: we can have following functions in the same scope.

```
sum(int num1, int num2)
sum(int num1, int num2, int num3)
sum(int num1, double num2)
```

The easiest way to remember this rule is that the parameters should qualify any one or more of the following conditions, they should have different **type**, **number** or **sequence** of parameters.

For example:
These two functions have different parameter **type**:

```
sum(int num1, int num2)
sum(double num1, double num2)
```

These two have different **number** of parameters:

```
sum(int num1, int num2)
sum(int num1, int num2, int num3)
```

These two have different **sequence** of parameters:

```
sum(int num1, double num2)
sum(double num1, int num2)
```

All of the above three cases are valid case of overloading. We can have any number of functions, just remember that the parameter list should be different. For example:

```
int sum(int, int)
double sum(int, int)
```

This is not allowed as the parameter list is same. Even though they have different return types, its not valid.

Function overloading Example

Lets take an example to understand function overloading in C++.

```
#include <iostream>
using namespace std;
class Addition {
public:
   int sum(int num1,int num2) {
      return num1+num2;    }
   int sum(int num1,int num2, int num3) {
     return num1+num2+num3;    }
};
int main(void) {
   Addition obj;
   cout<<obj.sum(20, 15)<<endl;
   cout<<obj.sum(81, 100, 10);
   return 0;}
```

Output:

```
35
191
```

Virtual Functions:
A virtual function is a member function in the base class that we expect to redefine in derived classes.

Basically, a virtual function is used in the base class in order to ensure that the function is **overridden**. This especially applies to cases where a pointer of base class points to an object of a derived class.

For example, consider the code below:

```
class Base {
  public:
   void print() {
      // code    }
};
class Derived : public Base {
  public:
   void print() {
      // code    }
};
```

Later, if we create a pointer of `Base` type to point to an object of `Derived` class and call the `print()` function, it calls the `print()` function of the `Base` class.

In other words, the member function of `Base` is not overridden.

```
int main() {
   Derived derived1;
   Base* base1 = &derived1;
   // calls function of Base class
   base1->print();
   return 0;}
```

In order to avoid this, we declare the `print()` function of the `Base` class as virtual by using the `virtual` keyword.

```
class Base {
  public:
   virtual void print() {
```

```
        // code    }
};
```

Example 1: C++ virtual Function

```
#include <iostream>
using namespace std;
class Base {
   public:
    virtual void print() {
        cout << "Base Function" << endl;    }
};
class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;    }
};
int main() {
    Derived derived1;
    // pointer of Base type that points to derived1
    Base* base1 = &derived1;
    // calls member function of Derived class
    base1->print();
    return 0;}
```

Output

```
Derived Function
```

Unit-V

Inheritance:

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base and Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form −

class derived-class: access-specifier base-class

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows −

```
#include <iostream>
using namespace std;
// Base class
class Shape {
  public:
    void setWidth(int w) {
      width = w;      }
    void setHeight(int h) {
      height = h;      }
   protected:
    int width;
    int height;};
// Derived class
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height);       }
};
int main(void) {
  Rectangle Rect;
  Rect.setWidth(5);
  Rect.setHeight(7);
  // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;
  return 0;}
```

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

Total area: 35

Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to - who can access them in the following way −

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

Reusability of code through Inheritance:

The main advantages of inheritance are **code reusability** and **readability**. When child class inherits the properties and functionality of parent class, we need not to write the same code again in child class. This makes it easier to reuse the code, makes us write the less code and the code becomes much more readable.
Lets take a **real life example** to understand this: Lets assume that Human is a class that has properties such as height, weight, colour etc and functionality such as eating(), sleeping(), dreaming(), working() etc.
Now we want to create Male and Female class, these classes are different but since both Male and Female are humans they share some common properties and behaviours (functionality) so they can inherit those properties and functionality from Human class and rest can be written in their class separately.
This approach makes us write less code as both the classes inherited several properties and functions from base class thus we didn't need to re-write them. Also, this makes it easier to read the code.

Inheritance Example

Before we discuss the types of inheritance, lets take an example:
Here we have two classes Teacher and MathTeacher, the MathTeacher class inherits the Teacher class which means Teacher is a parent class and MathTeacher is a child class. The child class can use the property collegeName of parent class.
Another important point to note is that when we create the object of child class it calls the constructor of child class and child class constructor automatically calls the constructor of base class.

```
#include <iostream>
using namespace std;
class Teacher {
public:
```

```
  Teacher(){
    cout<<"Hey Guys, I am a teacher"<<endl;  }
  string collegeName = "Beginnersbook";};
//This class inherits Teacher class
class MathTeacher: public Teacher {
public:
  MathTeacher(){
    cout<<"I am a Math Teacher"<<endl;  }
  string mainSub = "Math";
  string name = "Negan";};
int main() {
  MathTeacher obj;
  cout<<"Name: "<<obj.name<<endl;
  cout<<"College Name: "<<obj.collegeName<<endl;
  cout<<"Main Subject: "<<obj.mainSub<<endl;
  return 0;}
```

Output:

```
Hey Guys, I am a teacher
I am a Math Teacher
Name: Negan
College Name: Beginnersbook
Main Subject: Math
```

Type of Inheritance:

1) Single inheritance
2) Multilevel inheritance
3) Multiple inheritance
4) Hierarchical inheritance
5) Hybrid inheritance

Single inheritance

In Single inheritance one class inherits one class exactly.
For example: Lets say we have class A and B

```
B inherits A
```

Example of Single inheritance:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
    cout<<"Constructor of A class"<<endl;  }
};
class B: public A {
public:
  B(){
    cout<<"Constructor of B class";  }
};
int main() {
```

```
  //Creating object of class B
  B obj;
  return 0;}
```

Output:

```
Constructor of A class
Constructor of B class
```

2)Multilevel Inheritance

In this type of inheritance one class inherits another child class.

```
C inherits B and B inherits A
```

Example of Multilevel inheritance:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
    cout<<"Constructor of A class"<<endl;  }
};
class B: public A {
public:
  B(){
    cout<<"Constructor of B class"<<endl;  }
};
class C: public B {
public:
  C(){
    cout<<"Constructor of C class"<<endl;  }
};
int main() {
  //Creating object of class C
  C obj;
  return 0;}
```

Output:

```
Constructor of A class
Constructor of B class
Constructor of C class
```

Multiple Inheritance

In multiple inheritance, a class can inherit more than one class. This means that in this type of inheritance a single child class can have multiple parent classes.
For example:

```
C inherits A and B both
```

Example of Multiple Inheritance:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
    cout<<"Constructor of A class"<<endl;  }
};
```

```
class B {
public:
  B(){
     cout<<"Constructor of B class"<<endl;  }
};
class C: public A, public B {
public:
  C(){
     cout<<"Constructor of C class"<<endl;  }
};
int main() {
   //Creating object of class C
   C obj;
   return 0;}
Constructor of A class
Constructor of B class
Constructor of C class
```

4)Hierarchical Inheritance

In this type of inheritance, one parent class has more than one child class. For example:

```
Class B and C inherits class A
```

Example of Hierarchical inheritance:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
     cout<<"Constructor of A class"<<endl;  }
};
class B: public A {
public:
  B(){
     cout<<"Constructor of B class"<<endl;  }
};
class C: public A{
public:
  C(){
     cout<<"Constructor of C class"<<endl;  }
};
int main() {
   //Creating object of class C
   C obj;
   return 0;}
```

Output:

```
Constructor of A class
Constructor of C class
```

Data Abstraction:

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having any knowledge of its internals.

In C++, classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this −

```
#include <iostream>
using namespace std;
int main() {
  cout << "Hello C++" <<endl;
  return 0;}
```

Here, you don't need to understand how **cout** displays the text on the user's screen. You need to only know the public interface and the underlying implementation of ‘cout’ is free to change.

Access Labels Enforce Abstraction

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels −

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction

Data abstraction provides two important advantages −

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data is public, then any function that directly access the data members of the old representation might be broken.

Data Abstraction Example

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example −

```
#include <iostream>
using namespace std;
class Adder {
  public:
    // constructor
    Adder(int i = 0) {
      total = i;     }
    // interface to outside world
    void addNum(int number) {
      total += number;     }
    // interface to outside world
    int getTotal() {
      return total;     };
  private:
    // hidden data from outside world
    int total;};
int main() {
  Adder a;
  a.addNum(10);
  a.addNum(20);
  a.addNum(30);
  cout << "Total " << a.getTotal() <<endl;
  return 0;}
```

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

```
Total 60
```

Abstract Classes:

The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration as follows −

```
class Box {
  public:
    // pure virtual function
    virtual double getVolume() = 0;
 private:
```

```
   double length;      // Length of a box
   double breadth;     // Breadth of a box
   double height;      // Height of a box
};
```

The purpose of an **abstract class** (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error.

Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

Classes that can be used to instantiate objects are called **concrete classes**.

Abstract Class Example

Consider the following example where parent class provides an interface to the base class to implement a function called **getArea()** −

```
#include <iostream>
using namespace std;
// Base class
class Shape {
  public:
    // pure virtual function providing interface framework.
    virtual int getArea() = 0;
    void setWidth(int w) {
      width = w;      }
     void setHeight(int h) {
      height = h;      }
   protected:
    int width;
    int height;};
// Derived classes
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height);       }
};
class Triangle: public Shape {
  public:
    int getArea() {
      return (width * height)/2;       }
};
int main(void) {
  Rectangle Rect;
  Triangle  Tri;
  Rect.setWidth(5);
  Rect.setHeight(7);
  // Print the area of the object.
  cout << "Total Rectangle area: " << Rect.getArea() << endl;
 Tri.setWidth(5);
 Tri.setHeight(7);
```

```
 // Print the area of the object.
  cout << "Total Triangle area: " << Tri.getArea() << endl;
  return 0;}
```

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

```
Total Rectangle area: 35
Total Triangle area: 17
```

Templates:

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes, let us see how they work −

Function Template

The general form of a template function definition is shown here −

```
template <class type> ret-type func-name(parameter list) {
   // body of function
}
```

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

The following is the example of a function template that returns the maximum of two values −

```
#include <iostream>
#include <string>
using namespace std;
template <typename T>
inline T const& Max (T const& a, T const& b) {
   return a < b ? b:a; }
int main () {
   int i = 39;
   int j = 20;
   cout << "Max(i, j): " << Max(i, j) << endl;
   double f1 = 13.5;
   double f2 = 20.7;
   cout << "Max(f1, f2): " << Max(f1, f2) << endl;
   string s1 = "Hello";
   string s2 = "World";
   cout << "Max(s1, s2): " << Max(s1, s2) << endl;
   return 0;}
```

If we compile and run above code, this would produce the following result −

```
Max(i, j): 39
Max(f1, f2): 20.7
```

Max(s1, s2): World

Class Template

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here −

```
template <class type> class class-name {
  .
  .
  .
}
```

Here, **type** is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack −

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>
using namespace std;
template <class T>
class Stack {
  private:
    vector<T> elems;    // elements
  public:
    void push(T const&);  // push element
    void pop();               // pop element
    T top() const;            // return top element
    bool empty() const {      // return true if empty.
      return elems.empty();      }
};
template <class T>
void Stack<T>::push (T const& elem) {
  // append copy of passed element
  elems.push_back(elem);    }
template <class T>
void Stack<T>::pop () {
  if (elems.empty()) {
    throw out_of_range("Stack<>::pop(): empty stack");    }
   // remove last element
  elems.pop_back(); }
template <class T>
T Stack<T>::top () const {
  if (elems.empty()) {
    throw out_of_range("Stack<>::top(): empty stack");    }
    // return copy of last element
  return elems.back();    }
int main() {
  try {
    Stack<int>         intStack;  // stack of ints
    Stack<string> stringStack;    // stack of strings
```

```
    // manipulate int stack
    intStack.push(7);
    cout << intStack.top() <<endl;
    // manipulate string stack
    stringStack.push("hello");
    cout << stringStack.top() << std::endl;
    stringStack.pop();
    stringStack.pop();
  } catch (exception const& ex) {
    cerr << "Exception: " << ex.what() <<endl;
    return -1;   }
}
```

If we compile and run above code, this would produce the following result −

```
7
hello
Exception: Stack<>::pop(): empty stack
```

Exception Handling:

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **throw** − A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try** − A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows −

```
try {
  // protected code
} catch( ExceptionName e1 ) {
  // catch block
} catch( ExceptionName e2 ) {
  // catch block
} catch( ExceptionName eN ) {
  // catch block }
```

You can list down multiple **catch** statements to catch different type of exceptions in case your **try** block raises more than one exception in different situations.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using **throw** statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs −

```
double division(int a, int b) {
  if( b == 0 ) {
     throw "Division by zero condition!";
  }
  return (a/b); }
```

Catching Exceptions

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {
  // protected code
} catch( ExceptionName e ) {
 // code to handle ExceptionName exception }
```

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows −

```
try {
  // protected code
} catch(...) {
 // code to handle any exception }
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

```
#include <iostream>
using namespace std;
double division(int a, int b) {
  if( b == 0 ) {
     throw "Division by zero condition!";   }
  return (a/b);}
int main () {
  int x = 50;
  int y = 0;
  double z = 0;
   try {
     z = division(x, y);
     cout << z << endl;
  } catch (const char* msg) {
    cerr << msg << endl;   }
  return 0;}
```

Because we are raising an exception of type **const char***, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result −

Division by zero condition!