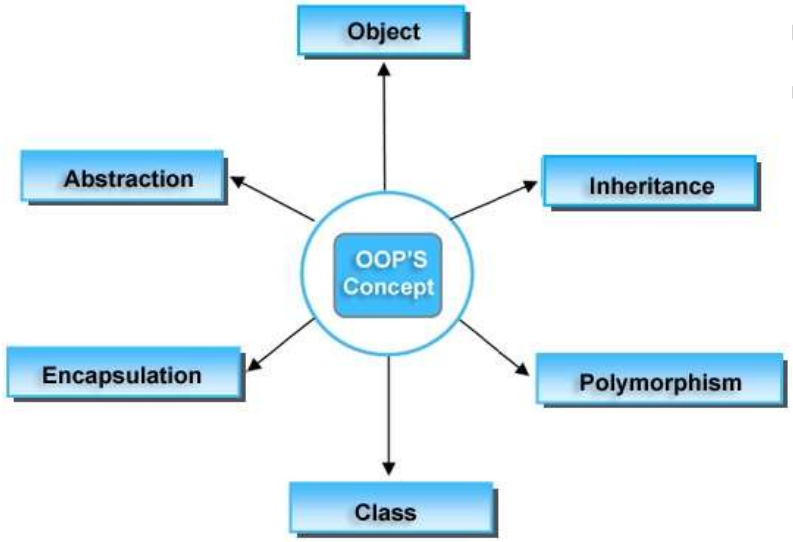
Chapter 01
Principles of Object Oriented Programming

1.1. Introduction

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts such as Object, Class, Inheritance, Polymorphism, Abstraction, and Encapsulation. The programming paradigm where everything is represented as an object is known as truly object-oriented programming language. **Smalltalk** is considered as the first truly object-oriented programming language. The main purpose of C++ programming was to add object oriented extension to the C programming language. If any programming language follows oops concept then that language is called as an object oriented programming language.

A. Features of OOPS:



1. Object

Object is an entity that has state and behavior. For example: chair, pen, table, keyboard, bike etc. It can be physical or logical entity. Objects are basic run-time entities in an object oriented system, objects are instances of a class.

2. Class

Class: Class is a blue print which is containing only list of variables and method and no memory is allocated for them. A class is a group of objects that has common properties. **Collection of objects** is called class. It is a logical entity. Class is also called as user defined data type or template for members such as data and functions.

3. Encapsulation

Encapsulation is a process of wrapping of data and methods in a single unit is called encapsulation. Encapsulation is achieved in C++ language by class concept. The main advantage of using of encapsulation is to secure the data from other methods, when we make a data private then these data only use within the class, but these data not accessible outside the class. Binding (or wrapping) code and data together into a single unit is known as encapsulation.

4. Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing. **Abstraction** is the concept of exposing only the required essential characteristics and behavior with respect to a context.
Hiding of data is known as **data abstraction**. In object oriented programming language this is implemented automatically while writing the code in the form of class and object or abstract classes can be used.

5. Inheritance

When one object acquires all the properties and behaviors of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism. The process of obtaining the data members and methods from one class to another class is known as **inheritance**. It is one of the fundamental features of object-oriented programming.

6. Polymorphism

Ability to represent more than one form is known as **Polymorphism**. Polymorphism means one entity can represent many different behaviors. **One task is performed by different ways.** In C++, we use Function overloading and Function overriding to achieve polymorphism.

7. Dynamic Binding:

Binding means to link a function call with function definition. Binding can be of two types:
(a) Static binding. (b) Dynamic binding

8. Message Passing:

In Object Oriented Programming the objects communicate with each other by sending and receiving message to each another.

B. Characteristics of OOPS:

(1) It follows bottom up approach.
(2) Primary focus is given to data rather than function.
(3) Programs are divided into small parts known as objects.
(4) Data and functions are defined inside the class so it can be accessed only by the object of the class.
(5) Objects of the same class communicate with each other using member functions.
(6) New data and function can be easily added.

C. What is C++:

C++ is an **object oriented programming language**. It was developed by **Bjarne Stroustrup** in early **1980** at AT&T bell laboratories. Bjarne Stroustrup developed C++ by adding the feature of class so initially it was known as "C with Class". In 1983 it was given name C++. Because C++ is an incremented version of c it is called C++ using the concept of increment operator (++).

D. Object Oriented Languages:

C++, Java, VB.net, C#.net, SmallTalk, Objective C, Ruby, Perl, Python, JADE, LAVA, PHP5, Simula, Swift, and UberCode etc……

1.2. Benefits of OOPS

- **Data encapsulation** benefit provided by the OOP using the concept of class to bind data and its associated functions together.
- **Data hiding** benefit provided by the OOP using the concept of public, private and protected visibility mode.
- The problem can be easily divided into smaller part using the concept of **object**.
- Using the concept of **function overloading** we can create same function with different number and type of arguments to perform different task.
- We can give additional meaning to the existing operator using the concept of **operator overloading**.
- **Code Reusability** benefit provided by the OOP using the concept of **inheritance**.
- Object of the same class communicates with each other using the concept of **message passing**.
- Complexity of the program can be easily managed using OOP.

1.3. Applications of OOPS

- User interface design such as windows, menu.
- Real Time Systems
- Simulation and Modeling
- Object oriented databases
- AI and Expert System
- Neural Networks and parallel programming
- Decision support and office automation systems etc.

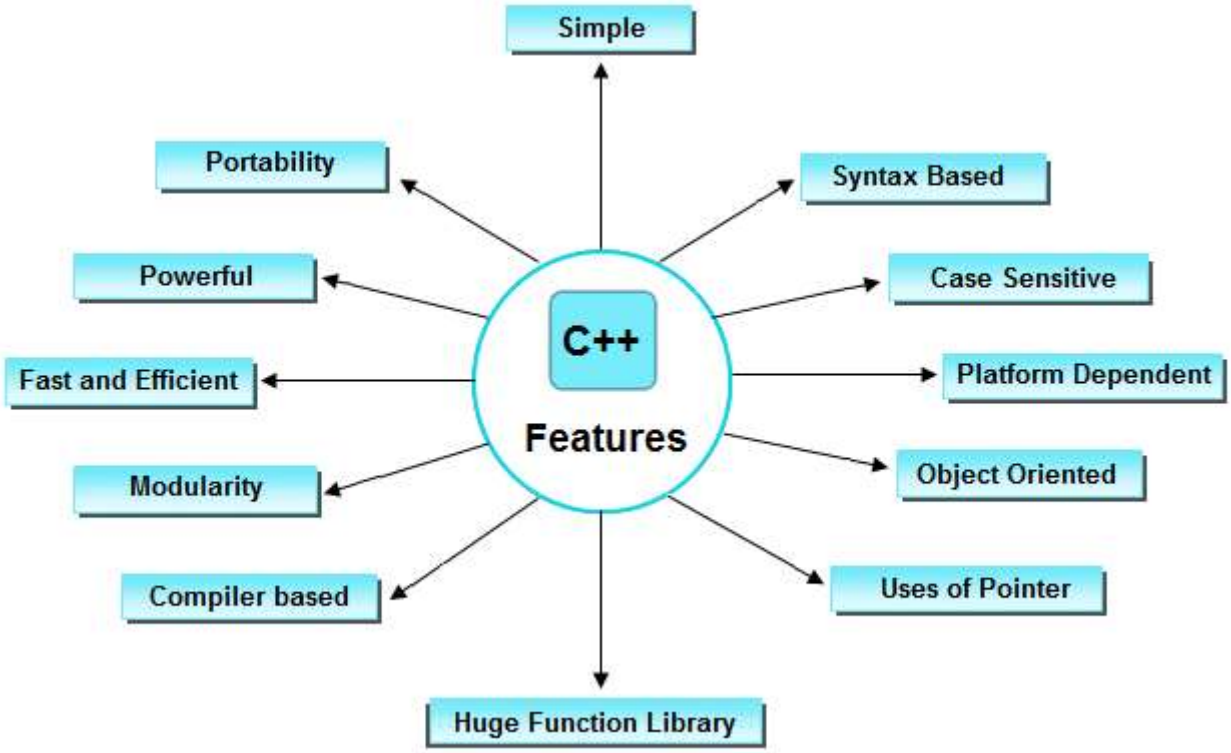
1.4. Differences

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP,Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.

Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Examples of POP are: Algol, C, FORTRAN, and Pascal.	Examples of OOP are: C++, JAVA, VB.NET, C#.NET.

C	**C++**
C is Procedural Language.	C++ is non Procedural i.e Object oriented Language.
No virtual Functions are present in C	The concept of virtual Functions are used in C++.
In C, Polymorphism is not possible.	The concept of polymorphism is used in C++. Polymorphism is the most Important Feature of OOPS.
Operator overloading is not possible in C.	Operator overloading is one of the greatest Feature of C++.
Top down approach is used in Program Design.	Bottom up approach adopted in Program Design.
No namespace Feature is present in C Language.	Namespace Feature is present in C++ for avoiding Name collision.
Multiple Declaration of global variables are allowed.	Multiple Declaration of global varioables are not allowed.
In C • scanf() Function used for Input. • printf() Function used for output.	In C++ • Cin>> Function used for Input. • Cout<< Function used for output.
Mapping between Data and Function is difficult and complicated.	Mapping between Data and Function can be used using "Objects"
In C, we can call main() Function through other Functions	In C++, we cannot call main() Function through other functions.
C requires all the variables to be defined at the starting of a scope.	C++ allows the declaration of variable anywhere in the scope i.e at time of its First use.
No inheritance is possible in C.	Inheritance is possible in C++
In C, malloc() and calloc() Functions are used for Memory Allocation and free() function for memory Deallocating.	In C++, new and delete operators are used for Memory Allocating and Deallocating.
It supports built-in and primitive data types.	It support both built-in and user define data types.
In C, Exception Handling is not present.	In C++, Exception Handling is done with Try and Catch block.

1.5. Features of C++



Simple

Every C++ program can be written in simple English language so that it is very easy to understand and developed by programmer.

Platform dependent

A language is said to be platform dependent whenever the program is execute in the same operating system where that was developed and compiled but not run and execute on other operating system. C++ is platform dependent language.

Portability

It is the concept of carrying the instruction from one system to another system. In C++ Language **.cpp** file contain source code, we can edit also this code. **.exe** file contain application, only we can execute this file. When we write and compile any C++ program on window operating system that program easily run on other window based system.

Powerful

C++ is a very powerful programming language, it have a wide verity of data types, functions, control statements, decision making statements, etc.

Object oriented Programming language

This main advantage of C++ is, it is object oriented programming language. It follow concept of oops like polymorphism, inheritance, encapsulation, abstraction.

Case sensitive

C++ is a case sensitive programming language. In C++ programming 'break and BREAK' both are different. If any language treats lower case latter separately and upper case latter separately than they can be called as case sensitive programming language [Example c, c++, java, .net are sensitive programming languages.] otherwise it is called as **case insensitive** programming language [Example HTML, SQL is case insensitive programming languages].

Compiler based

C++ is a compiler based programming language that means without compilation no C++ program can be executed. First we need compiler to compile our program and then execute.

Syntax based language

C++ is a strongly tight syntax based programming language. If any language follows all rules and regulation very strictly then it is known as strongly syntax based language.

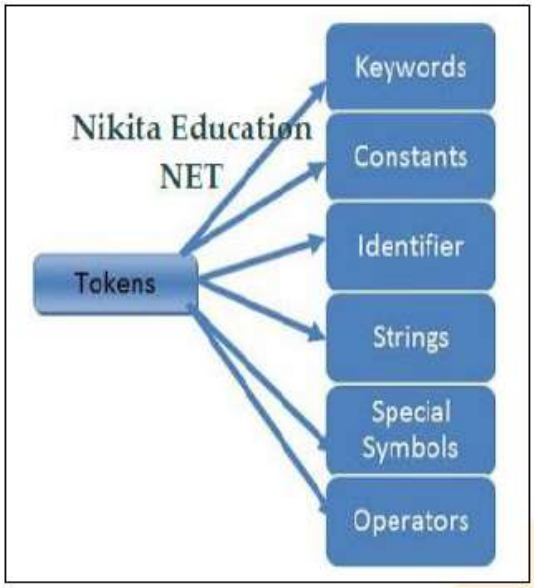
1.6. Expressions and Control Structures

A. Expressions

In programming, an expression is any **legal combination of symbols that represents a value or operation**. C++ Programming provides its own rules of Expression, whether it is legal expression or illegal expression. For example, in the C++ language x+5 is a legal expression. Every expression consists of at least one operand and can have one or more operators. Operands are values and Operators are symbols that represent particular actions. In C++ programming language, expressions are divided into THREE types. They are as follows...

1. Infix Expression
2. Postfix Expression
3. Prefix Expression

B. Tokens or basic building blocks



No	Token Type	Example 1	Example 2
1	Keyword	do	while
2	Constants	05	-14
3	Identifier	var1	sum
4	String	"NET"	"NSPL"
5	Special Symbol	*	@
6	Operators	++	/

Token	Meaning
Keyword	Keywords are reserved words used by compiler.
Constant	Constants are expressions with a fixed value
Identifier	The term identifier is usually used for variable names
String	Sequence of characters
Special Symbol	Symbols other than the Alphabets and Digits and white-spaces
Operators	A symbol that represent a specific mathematical or non mathematical action

C. Data types and variables

C++ Data Types

Data type is a keyword used to identify type of data. It is used for storing the input of the program into the main memory (RAM) of the computer by allocating sufficient amount of memory space in the main memory of the computer. In general every programming language is containing three categories of data types. They are

- Fundamental or primitive data types
- Derived data types
- User defined data types

Types	Data Types
Basic Data Type	int, long, char, float, double, long double etc
Derived Data Type	array, pointer, etc
Enumeration Data Type	enum
User Defined Data Type	structure, class

C++ Variable

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times. It is a way to represent memory location through symbol so that it can be easily identified.

Syntax:

1. type variable_list;

Example:

1. **int** x;
2. **float** y;
3. **char** z;

Scope of Variable in C++

In C++ language, a variable can be either of global or local scope.

1. *Global variable*

Global variables are defined outside of all the functions, generally on top of the program. The global variables will hold their value throughout the life-time of your program.

2. *Local variable*

A local variable is declared within the body of a function or a block. Local variable only use within the function or block where it is declare.

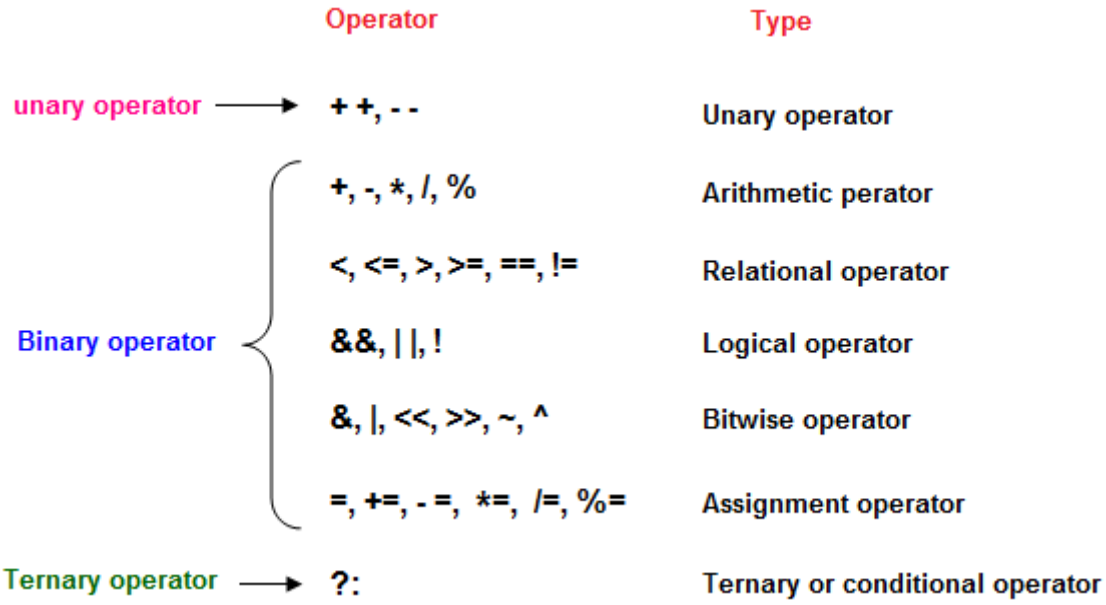
```
#include<iostream.h>
#include<conio.h>

int a;   // global variable
void main()
{
      int b;    // local variable
      a=10, b=20;
      cout<<"Value of a: "<<a;
      cout<<"Value of b: "<<b;
      getch();
}
```

D. Operators

Operator is a special symbol that tells the compiler to perform specific mathematical or logical or Bitwise Operation.

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Ternary or Conditional Operators



E. Basic Structure of C++ Program

Documentation
Header files section
Namespaces [optional]
Global declarations
Class definitions
Main function

```
#include<headerfilename.h>                    //include section
using namespace namespacename;
global variables;
class class_name                             //class definition
{
        Access specifier:                    //private or public
        data members;                        //instance variables
        user defined method;                 //functions
        {
                .....
                .....
        }
};
```

```
returntype main()                                    //main method
{
.....
.....
}
```

Example:

```
#include <iostream>
using namespace std;
int main()
{
   cout << "Welcome to Nikita Education for C++ Programming.";
   return 0;
}
```

F. Input / Output Functions

1. cout

Standard output stream. The cout is a predefined object of ostream class. The cout is used in conjunction with stream insertion operator (<<) to display the output on a console.

```
#include <iostream>
using namespace std;
int main( ) {
   char ary[] = "Welcome to C++ tutorial";
   cout << "Value of ary is: " << ary << endl;
}
```

2. cin

Standard input stream. The cin is a predefined object of istream class. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

```
#include <iostream>
using namespace std;
int main( ) {
   int age;
   cout << "Enter your age: ";
   cin >> age;
   cout << "Your age is: " << age << endl;
}
```

G. Array & String in C++

An array is a collection of similar data type value in a single variable. It is a derived data type in C++, which is constructed from fundamental data type of C++ language.

Advantage of array

- **Code Optimization:** Less code is required; one variable can store numbers of value.

- **Easy to traverse data:** By using array easily retrieve the data of array.
- **Easy to sort data:** Easily short the data using swapping technique
- **Random Access:** With the help of array index you can randomly access any elements from array.

There are 2 types of arrays in C++ programming:

1. Single Dimensional Array
2. Multidimensional Array

Syntax: Datatype arrayName[SIZE]; or Datatype ArrayName[SIZE][SIZE];

Example: int myarray[5]; or int arr[]={10,20,30,40,50};

H. Decision Making Statements

C++ conditional statements allow you to make a decision, based upon the result of a condition. These statements are called as Decision Making Statements or Conditional Statements.

- if statement
- if-else statement
- else-if statement
- switch statement
- Conditional Operator

I. Loop Control Statements

Sometimes it is necessary for the program to execute the statement several times, and C++ loops execute a block of commands a specified number of times until a condition is met.

- while loops
- do while loops
- for loops

J. Jump Control Statements

Jump control statements allow us to jump from one statement to another statement during program execution. C++ supports following jump control statements.

- goto statement
- break statement
- continue statement

K. Functions in C++

Functions are used for divide a large code into module, due to this we can easily debug and maintain the code. Functions are used for Code Re-usability, Develop an application in module format, Easily to debug the program, Code optimization**:** No need to write lot of code. There are two types of functions in C++ i.e. *Library function or pre-define function and User defined function.*

Syntax:

```
return_type  function_name(parameter)
{
        function body;
}
```

Chapter 02
Classes & Objects

2.1 Classes

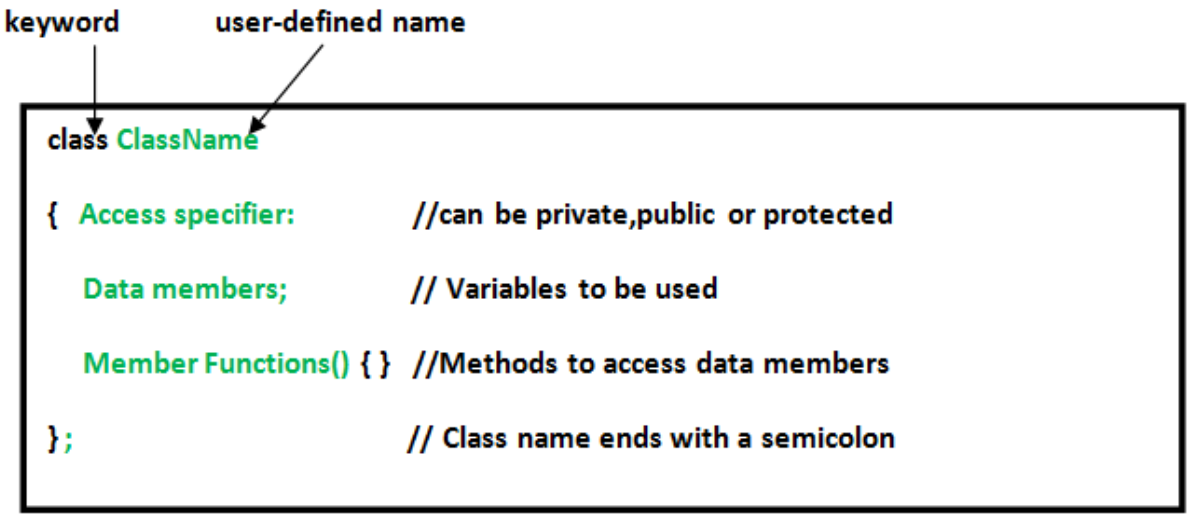
A. Class:

A class is a **user defined data type** that allows us to bind data and its associated functions together as a **single unit**. Thus class provides the facility of data encapsulation. Class provides the facility of data hiding using the concept of visibility mode such as **public, private and protected**. Once a class is defined we can create an object of the class to access variables and functions defined inside the class.

A class is **an abstract data type** similar to 'C structure'. The class is **a container** for data members and methods. Class is **a group of similar objects** or it is a template from which objects are created. A class is a blueprint for the object.

A class in C++ contains, following properties;

- Data Member
- Method
- Constructor
- Block



Syntax:

```
class Class_Name
{
Private:
      Data-Type Variable_Name;
      Function declaration or Function Definition;
Public:
      Data-Type Variable_Name;
      Function declaration or Function Definition;
};
```

Class can be created using the **class keyword**. The class definition starts with curly bracket and ends with curly bracket followed by semicolon. We can declare variables as well as functions inside the curly bracket as shown in the syntax. The variables defined inside class are known as **data member** and the function declared inside the class are known as **member function**. In order to provide **data hiding** facility class provides the concept of **visibility mode** such as **private, public or protected**. If you don't specify any visibility mode for the member of the class then **by default all the members of the class are considered as private**.

Example:

```
class test
{
      int a, b;                                  //data member
public:                                          //access specifier
      void input ();                             //member function
      {
            cout<<"Enter Value of a and b";
            cin>>a>>b;
      }
      void output ()                             //member function
      {
            cout<<"A="<<a<<endl<<"B="<<b;
      }
};
```

2.2 Objects

In C++, Object is a **real world entity**, for example, chair, car, pen, mobile, laptop etc. In other words, object is an entity that **has state and behavior**. Here, **state means data** and **behavior means functionality**. Object is a **runtime entity**, it is created at runtime. Object is **an instance** of a class. All the members of the class can be accessed through object. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

An Object in C++ has three characteristics:

- State: Represents data (value) of an object.
- Behavior: Represents the behavior (functionality).
- Identity: Object identity is typically implemented via a unique ID.

Syntax: `ClassName ObjectName;`

Example: `Employee e1, e2, e3;`

In real world many examples of object and class like dog, cat, and cow are belong to animal's class. Each object has state and behaviors. For example a dog has state:- color, name, height, age as well as behaviors:- barking, eating, and sleeping.

Note: Objects are created like variables using class name as its type.

2.2.1 Create Object in C++

There are two ways to create objects in C++:

a. Create object with class definition.

```
class test
{
      int a,b;
public:
      void input ();
      void ouput ();
}t1,t2,t3;
```

Here, t1, t2 and t3 are the objects of class test. Like creating references of structure in C.

b. Create object inside any function.

```
class test
{
      int a,b;
public:
      void input (){ …… }
      void ouput (){ …… }
};

void main()
{
   test t1, t2, t3;          //object of class test
}
```

2.2.2 Accessing Class Members Using Objects:

The data member and member function declared as a public can be accessed by **dot (.) or pointer (-->)** operator directly using the object of the class. But the data member and member function declared as private cannot be accessed directly using the object of the class.

```
Object_Name . Data_Member = value;

Object_Name . Member_Function (Argument_List);
```

Example:

```
t1. input ();

t1. output ();
```

Example:

```
class Test
{
      int b;                  //by default b is private variable of class
Public:
      int a;                  //a is public variable of class
      void inputb()           //member function of a class
      {
            b=20;
      }
};

int main()
{
      Test T1;
      T1.a=10;                //works because a is public
      T1.b=20;                //error because b is private
      T1.inputb ();           //works because inputb () is public
      return 0;
}
```

2.3 Access Specifiers/Modifiers in C++

Access modifiers are used to implement important feature of Object Oriented Programming known as **Data Hiding**. Access modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions. There are 3 types of access modifiers available in C++:

1. Public
2. Private
3. Protected

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be **Private**.

Examples:

```
class PublicAccess {
      public:           // public access specifier
      int x;            // Data Member Declaration
      void display();   // Member Function decaration
}

class PrivateAccess {
      private:          // private access specifier
      int x;            // Data Member Declaration
      void display();   // Member Function decaration
}
```

```
class ProtectedAccess {
        protected:              // protected access specifier
        int x;                  // Data Member Declaration
        void display();         // Member Function decaration
}
```

2.3.1 Public: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

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

class Circle                // class definition
{
    public:
        double radius;

        double  compute_area()
        {
            return 3.14*radius*radius;
        }

};

int main()      // main function
{
    Circle obj;

    // accessing public datamember outside class
    obj.radius = 5.5;

    cout << "Radius is:" << obj.radius << "\n";
    cout << "Area is:" << obj.compute_area();
    return 0;
}
```

```
Output:
Radius is:5.5
Area is:94.985
```

2.3.2 Private: The class members declared as **private** can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

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

class Circle
{
    // private data member
    private:
        double radius;

    // public member function
    public:
        double  compute_area()
        {   // member function can access private
            // data member radius
            return 3.14*radius*radius;
        }

};
// main function
int main()
{
    // creating object of the class
    Circle obj;

    // trying to access private data member
    // directly outside the class
    obj.radius = 1.5;

    cout << "Area is:" << obj.compute_area();
    return 0;
}
```

Output:

```
In function 'int main()':
11:16: error: 'double Circle::radius' is private
       double radius;
              ^
31:9: error: within this context
    obj.radius = 1.5;
      ^
```

However we can access the private data members of a class indirectly using the public member functions of the class. Below program explains how to do this:

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

class Circle
{
    // private data member
    private:
        double radius;

    // public member function
    public:
        double  compute_area(double r)
        {   // member function can access private
            // data member radius
            radius = r;

            double area = 3.14*radius*radius;

            cout << "Radius is:" << radius << endl;
            cout << "Area is: " << area;
        }

};

// main function
int main()
{
    // creating object of the class
    Circle obj;

    // trying to access private data member
    // directly outside the class
    obj.compute_area(1.5);

    return 0;
}
```

Output:
Radius is:1.5
Area is: 7.065

2.3.3 Protected: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

```
#include <bits/stdc++.h>
using namespace std;

// base class
class Parent
{
    // protected data members
    protected:
    int id_protected;

};

// sub class or derived class
class Child : public Parent
{


    public:
    void setId(int id)
    {

        // Child class is able to access the inherited
        // protected data members of base class

        id_protected = id;

    }

    void displayId()
    {
        cout << "id_protected is:" << id_protected << endl;
    }
};

// main function
int main() {

    Child obj1;

    // member function of derived class can
    // access the protected data members of base class

    obj1.setId(81);
```

```
    obj1.displayId();
    return 0;
}
```

Output:
id_protected is:81

2.4 Member Functions

Function is a set of instructions or self contained block of statements used to perform specific task. Functions are used to divide complex task into smaller manageable units called as modules of the application. The primary aim of functions is to avoid repetitive code in main program.

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

Member function can be defined in two different way:
(1) Inside class
(2) Outside class

(1) Inside Class:

When we declare the function in the class at the same time we can also give the definition of the function in the class as shown below. The function defined inside class becomes **inline** by default.

```
class Test
{
     int a,b;
public:
     //member function declared and defined inside the class
     void input ()
     {
          cout<<"Enter Value of a";
          cin>>a>>b;
     }
};
```

(2) Outside Class:

We can also define the member function outside the class. But at that time we have to instruct compiler this function belongs to which class using scope resolution operator as follow:

Syntax:
```
Return-Type Class_Name :: Function_Name (parameter list)
{
     //Function body
}
```

Example:

```
class Test
{
      int a,b;
Public:
      void input ();
};
void test :: input ()
{
      cout<<"Enter Value of a";
      cin>>a>>b;
}
```

2.4.1 Private Member Function

Like data member of the class a member function can also be declared as private so that it cannot be accessed outside the class. If we declare private member function then it cannot be accessed directly using the object of the class so we have to access it from the public member function of the same class.

Example:

```
class Circle
{
      int r;
      float area ();                    //private member function

public:
      void getRadius ();
      void DisplayArea ();
};

int Circle ::area ()
{
      return (3.14 * r * r);
}

void Circle :: getRadius ()
{
      cout<<"Enter Radius";
      cin>>r;
}

void Circle ::DisplayArea ()
{
      cout<<"Area="<<area ();
}
```

```
int main()
{
      Circle C1;
      C1.getRadius();
      C1.DisplayArea ();
      return 0;
}
```

In the above example we have declared three-member function getRadius (), DisplayArea () and area(). In which area () is defined as private. We have called only two member functions getRadius () and DisplayArea () using the name of the object inside main function. We have called the third member function area () from inside the DisplayArea () member function. This concept is known as private member function.

2.5 Types of Member Functions in Class

1. Simple functions
2. Static functions
3. Const functions
4. Inline functions
5. Friend functions
6. Virtual functions

Simple Member functions

These are the basic member function, which don't have any special keyword like static etc as prefix. All the general member functions, which are of below given form, are termed as simple and basic member functions.

```
return_type functionName(parameter_list)
{
      //function body;
}
```

Static member functions

Static member functions are designed to work with static data members. In order to make a member function as static we need to precede the function declaration with **static keyword**. Static member function can access only static member of the class in which it is declared. Static member function is not a part of class object so it cannot be called using object of the class. Static member function can be called using class name and scope resolution operator as shown below:

```
                  Class_Name :: Function_Name ();
class X
{
      public:
      static void f(){};
};
int main()
{
      X::f();   // calling member function directly with class name
}
```

Const Member functions

When used with member function, such member functions can never modify the object or its related data members.

//Basic Syntax of const Member Function:

```
void fun() const {}
```

Inline functions

All the member functions defined inside the class definition are by default declared as Inline. We will study Inline Functions in details in the next topic.

Friend functions

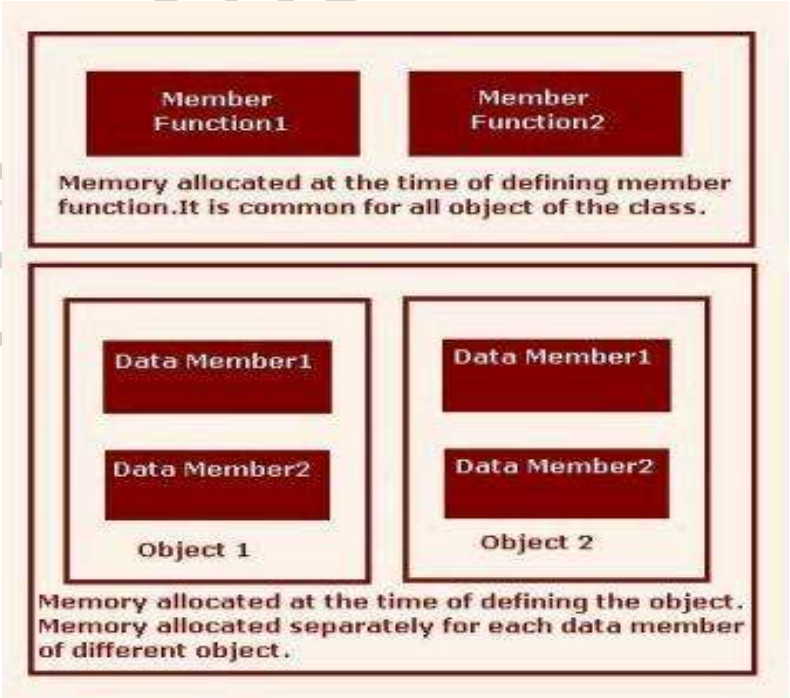
Friend functions are not class member function. Friend functions are made to give **private** access to non-class functions. You can declare a global function as friend, or a member function of other class as friend.

Virtual Functions

Virtual Function is a function in base class, which is overridden in the derived class, and which tells the compiler to perform **Late Binding** on this function. Virtual Keyword is used to make a member function of the base class Virtual.

2.6 Memory Allocation

Once you define class it will not allocate memory space for the data member of the class. The memory allocation for the data member of the class is performed separately each time when an object of the class is created. Since member functions defined inside class remains same for all objects, only memory allocation of member function is performed at the time of defining the class. Thus memory allocation is performed separately for different object of the same class. All the data members of each object will have separate memory space. The memory allocation of class members is shown below:



Hence data member of the class can contain different value for the different object; memory allocation is performed separately for each data member for different object at the time of creating an object. Member function remains common for all objects. Memory allocation is done only once for member function at the time of defining it.

2.6.1 Dynamic Memory Allocation

C uses malloc() and calloc() function to allocate memory dynamically at run time and uses free() function to free dynamically allocated memory. C++ supports these functions and also has two operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way.

Allocation of heap memory using new operator

Syntax: datatype pointername = new datatype

Example: int *new_op = new int;

```
//Allocating block of memory
int *new_op = new int[10];
//allocating memory for object of class Stud
Stud * S = new Stud();
```

Deallocation of memory using delete operator

Syntax: delete pointer_variable

Example: delete new_op; or delete S;

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

class Box
{
public:
   int a,b;
   int * c=new int;
   void show()
   {
      cout<<a<<" , "<<b;
   }
};

int main()
{
   Box * p=new Box();
   p->a=50;
   p->b=10;
   p->show();
   delete p;
}
```

2.7 Arrays

An array is a collection of similar data type value in a single variable. It is a derived data type in C++, which is constructed from fundamental data type of C++ language. Array in C++ is a group of similar types of elements that have contiguous memory location. Arrays can be declared as the data members of a class. The arrays can be declared as private, public or protected members of the class.

Advantages of C++ Array

- Code Optimization (less code)
- Random Access
- Easy to traverse data
- Easy to manipulate data
- Easy to sort data etc.

C++ Array Types

There are 2 types of arrays in C++ programming:

1. Single Dimensional Array
2. Multidimensional Array

2.7.1 Arrays within a Class

- Arrays can be declared as the members of a class.
- The arrays can be declared as private, public or protected members of the class.
- To understand the concept of arrays as members of a class, consider this example.

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

const int size=5;

class Student{
   int roll_no;
   int marks[size];
public:
   void getdata ();
   void tot_marks ();
};

void Student :: getdata()
{
   cout<<"\nEnter roll no: ";
   cin>>roll_no;
   for(int i=0; i<size; i++)
   {
      cout<<"Enter marks in subject"<<(i+1)<<": ";
      cin>>marks[i] ;
   }
}
void Student :: tot_marks() //calculating total marks
{
   int total=0;
   for(int i=0; i<size; i++)
      total+= marks[i];
   cout<<"\n\nTotal marks "<<total;
}
```

```
int main()
{
   Student stu;
   stu.getdata() ;
   stu.tot_marks() ;
   return 0;
}
```

2.7.2 Array of Objects

- Like array of other user-defined data types, an array of type class can also be created.
- The array of type class contains the objects of the class as its individual elements.
- Thus, an array of a class type is also known as an array of objects.
- An array of objects is declared in the same way as an array of any built-in data type.

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

class Student
{
   int rollno;
   char name[20];
public:
   void Input();
   void Output();
};

void Student::Input()
{
   cout<<"Enter Roll Number:";
   cin>>rollno;
   cout<<"Enter Name:";
   cin>>name;
}

void Student::Output()
{
   cout<<"Roll Number:"<<rollno<<endl;
   cout<<"Name:"<<name<<endl;
}
int main()
{
   Student S[3];
   int i;
   for(i=0;i<3;i++)
      S[i].Input();
   for(i=0;i<3;i++)
      S[i].Output();
   return 0;
}
```

2.8 Object as a Function Argument

In C++ we can pass object as a function argument and also return an object from the function. An object can be passed to the function using two methods:
(1) Call By value: In which copy of the object is passed to the function.
(2) Call By reference: In which an address of the object is passed to the function.

2.8.1 Passing object as an argument

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

class Student
{
   int a,b,c;
   int d;
public:
   void add()
   {
      a=b=23;
      c=a+b;
   }
   void mul(Student d1)
   {
      int res;
      d=5;
      res=d*d1.c;
      cout<<"result is :"<<res<<endl;
   }
};

int main()
{
   Student d1;
   d1.add();
   Student d2;
   d2.mul(d1);
   return 0;
}
```

2.8.2 Returning object from function

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

class Demo
{
   int roll;
public:
   Demo setRoll(int r);
```

```
    void show(Demo ob2);
};

Demo Demo::setRoll(int r)
{
    Demo ob2;
    ob2.roll=r;
    return ob2;
}

void Demo::show(Demo ob2)
{
    cout<<ob2.roll;
}

int main()
{
    Demo ob1;
    Demo ob2=ob1.setRoll(175105);
    ob1.show(ob2);
    return 0;
}
```

2.9 Static Members

Static is a keyword in C++ used to give special characteristics to an element. Static elements are allocated storage only once in a program lifetime in static storage area. And they have a scope till the program lifetime. Static Keyword can be used with following:

- Static Data Members
- Static Member Functions

2.9.1 Static data members

- To declare **static** keyword is used.
- Static creates a single common variable means it allocates memory only once to variable
- Static variable is commonly shared by all the objects of same class
- Static data members are initialized before the objects of class are created
- *Static data member is declared in the class but it must be defined outside the class using class name and scope resolution operator (::)*

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

class item
{
    static int count;
public:
    void DisplayCounter()
    {
        count++;
```

```
        cout<<"count:"<<count<<endl;
    }
};

int item::count;          //static data member must be defined outside the class

int main()
{
    item a,b,c;
    a.DisplayCounter();
    b.DisplayCounter();
    c.DisplayCounter();
    return 0;
}
```

2.9.2 Static member functions

- Static member functions are designed to work with static data members.
- In order to make a member function as static we need to precede the function declaration with **static keyword**.
- Static member function can access only static member of the class in which it is declared.
- Static member function is not a part of class object so it can not be called using object of the class.
- Static member function can be called using class name and scope resolution operator as shown below:

Class_Name :: Function_Name ();

```
#include <iostream>
using namespace std;
class test
{
private:
    static int count;
public:
    void setCount(void);
    static void DisplayCounter(void);
};

void test :: setCount(void)
{
    count++;
}

void test :: DisplayCounter(void)
{
    cout<<"Count:"<< count << endl;
}

int test::count;
```

```
int main(void)
{
   test t1, t2;
   t1.setCount();
   test :: DisplayCounter();
   t2.setCount();
   test :: DisplayCounter();
   return(0);
}
```

********************************Difference Between********************************

Object Oriented	Procedure Oriented
In Object Oriented Programming primary focus is on object.	In Procedure Oriented Programming primary focus is on function.
It follows bottom up approach.	It follows top down approach.
Primary importance is given to data instead of function.	Primary importance is given to function instead of data.
The program is divided into small parts Known as object.	The program is divided into small parts Known as function.
Data can not move freely from one function to another function.	Data moves freely from one function to another function.
It provides data hiding facility.	It does not provides data hiding facility.
Data and functions can be added easily Whenever required.	Data and functions cannot be added easily.
C++, JAVA is the example of Object Oriented Programming Language.	C, COBOL, FORTRAN is the example of Procedure Oriented Programming.

Data Abstraction	Data Encapsulation
Data abstraction refers to the process of providing only essential information to the outside word and hiding their background details.	The process of binding data and its associated function into a single unit is known as data encapsulation.

cout	cin
cout is known as an output statement.	cin is known as an input statement.
It is used to display the content of the variable or string on the output screen.	It is used to input data into the program.
cout is a predefined object defined in ostream.h which represents the standard output stream in C++ which is monitor.	cin is a predefined object defined in istream.h file which represents the standard input stream in C++ which is keyboard.
Syntax: cout << Variable_Name or "String"; Example cout << "Welcome To C++";	Syntax: cin >> Variable_Name ; Example cin >>a;

Single Line Comment	Multi Line Comment
It was introduced in C++ so it is also known as C++ Style comment.	It was introduced in C so it is also known as C Style comment.
It is starts with //.	It is starts with /* and ends with */ symbols.

Example:
// This Program Implements Class
// and Objects
	Example:
	/* This Program Implements Class And Objects
	*/

We can not use this comment inside C++ statements.	We can also use this comment inside C++ statements.

While Loop	Do While Loop
In while loop, condition is checked at the beginning of loop.	In do...while loop, condition is checked at the end of loop.
It is known as entry controlled loop.	It is known as exit controlled loop.
If condition is false at the first trial body of the loop never executes.	If condition is false at the first trial body of the loop executes at least once.

Syntax:
while(condition)
{
Body of loop
}
	Syntax:
	do
	{
	Body of loop
	} while(condition);

Break	Continue
Break statement is used to transfer control of the program immediately after the loop or switch case statement.	Continue statement is used to skip some part of the loop and transfer control of the program to the next iteration in the loop.

Normal Function	Inline Function
Normal Function decrease execution speed of the program.	Inline Function increase execution speed of the program.
It occupies less memory during runtime as compared to inline function.	It occupies more memory during runtime as compared to normal function.
Normal function can contain any number of statements.	Inline function works fine if it contains only two or three statements.

Public	Private
Public members of the class can be accessed directly using object of the class.	Private members of the class can not be accessed directly using object of the class.
Inheritance of public members is possible.	Inheritance of private members is not possible.

Call By Value	Call By Reference
In this method of calling the function values of the original variables is passed to the function.	In this method of calling the function addresses of the original variables is passed to the function using concept of pointer.
As you pass the values of original variables it is copied in the formal parameters of the function. Thus function works with the copy of the original variables. Any changes that are made to the variables inside function do not affect the original variables.	As you pass the addresses of original variables, the function works with the original variables. Any changes that are made to the variables inside function also affect the original variables.
Call by value method can not alter the value of the original variable.	Call by reference method alters the value of the original variable.

Compile Time Polymorphism	Run Time Polymorphism
Compile time polymorphism is also known as static binding or early binding.	Run time polymorphism is also known as dynamic binding or late binding.
Function overloading and Operator overloading are the example of compile time polymorphism.	Virtual function is the example of run time polymorphism.
In compile time polymorphism which function to invoke is determined at compile time so it is called static or early binding.	In run time polymorphism which function to invoke is determined at runtime so it is called dynamic binding or late binding.

Default Constructor	Parameterized Constructor
Default Constructor does not accept any arguments.	Parameterized Constructor accept one or more arguments.
It initializes same value for data member of different objects.	It can initialize different value for data member of different objects.

Chapter 03
Functions in C++

3.1 Functions in C++

Function is a group of logically related statements that is used to perform specific task. Functions are used for dividing a large code into module, due to this we can easily debug and maintain the code. There is no need to write same block of statements again and again. Thus it will reduce the length of the program. Error handling is easy because we have to check only function body instead of same block of statement again and again.

Type of Function

There are two type of function in C++ Language. They are;

- Library functions or pre-defined functions.
- User defined functions.

Syntax for User Defined Functions:

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

3.2 Function Prototyping

The declaration of function in the program is known as Function Prototype. A function prototype is a **function declaration** that specifies the **data types** of its **arguments** in the **parameter list. A function prototype** or **function interface** is a declaration of a function that specifies the function's name and type signature, but omits the function body.

Prototype style:

- Functions are declared explicitly with a prototype before they are called.
- Multiple declarations must be compatible, parameter types must agree exactly.
- Arguments to functions are converted to the declared types of the parameters.
- Empty parameter lists are designated using the **void** keyword.
- Ellipses are used in the parameter list of a prototype to indicate that a variable number of parameters are expected.

Syntax: `Return-type Function_ Name (Argument_List);`

Example: `int sum (int a, int b);`

Function Prototype provides following information to the compiler:

- Name of the Function
- Return Type of the Function
- Number of arguments and their Data Type

Whenever compiler founds any function call statement, first it will check function prototype to ensure following things:

- Weather the function that is called is declared or not.
- Weather proper number of arguments is passed while calling the function or not.
- Weather the data type of the arguments that are passed is corresponds to the arguments specified in the function prototype or not.
- If the function returns any value then it is corresponds to the return type specified in the function prototype or not.

If all the above mentioned criteria are satisfied then control of the program is transferred to the function definition otherwise compiler will generate error.

While declaring the function you should keep following points in the mind:

- You must specify data types for each argument separately.
  Example:
  ```
  int sum (int a, int b); // valid
  int sum (int a, b); // Not Valid
  ```
- It is not compulsory to specify name of the arguments in the function declaration.
  Example:
  ```
  int sum (int, int); // valid
  ```
- If function does not accept any argument then you can leave the parenthesis empty.
  Example:
  ```
  int sum (); // valid
  ```

3.3 Function Calling

In C++ you can call the function using two methods:

3.3.1 Call By Value:

In this method when the function is called it will first check function prototype to see whether the specified function is declared or not. If the function is declared then it will match number of arguments, data type and return type. Now compiler will creates new variables and copy the value of the arguments in to newly created variable. Thus function works with the newly created variables. So it will not affect the original variables in the calling program. Using this method we cannot alter the value of the original variable.

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

class CallByValue
{
public:
    void swap(int a, int b)
    {
        int temp;
        temp=a;
        a=b;
        b=temp;
    }
};
```

```
int main()
{
 CallByValue obj;
 int a=100, b=200;
 obj.swap(a, b);  // passing value to function
 cout<<"Value of a : "<<a<<endl;
 cout<<"Value of b : "<<b;
 return 0;
}
```

3.3.2 Call By Reference:

In this method when the function is called it will first check function prototype to see whether the specified function is declared or not. If the function is declared then it will match number of arguments, data type and return type. Now instead of passing value of the variable reference of the variable is passed to the function using the concept of reference variable. Thus function works with the original variable. Using this method we can alter the value of the original variable.

```
#include<iostream>
#include<conio.h>

using namespace std;

class CallByRef
{
public:
    void swap(int *a, int *b)
    {
        int temp;
        temp=*a;
        *a=*b;
        *b=temp;
    }
};

int main()
{
 CallByRef obj;
 int a=100, b=200;
 obj.swap(&a, &b);  // passing value to function
 cout<<"Value of a : "<<a<<endl;
 cout<<"Value of b : "<<b;
 return 0;
}
```

Difference between call by value and call by reference.

	call by value	**call by reference**
1	This method copy original value into function as a arguments.	This method copy address of arguments into function as a arguments.
2	Changes made to the parameter inside the function have no effect on the argument.	Changes made to the parameter affect the argument. Because address is used to access the actual argument.
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

Note: By default, C++ uses call by value to pass arguments.

3.3.3 Return by reference

As we can pass reference of the variable to the function, function can also return a reference of the variable to the calling program.

Return by Reference

- A C++ program can be made easier to read and maintain by using references rather than pointers.
- A C++ function can return a reference in a similar way as it returns a pointer.
- When a function returns a reference, it returns an implicit pointer to its return value.
- This way, a function can be used on the left side of an assignment statement.

```
#include <iostream>

using namespace std;

int n;

int & test();

int main() {
   test() = 5;
   cout<<n;
   return 0;
}

int & test() {
return n;
}
```

3.4 Inline Functions

Inline Function is powerful concept in C++ programming language. 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. To make any function inline function just preceded that function with **inline** keyword. The inline function must be defined before it is called in the program. It means at the starting of the program. The main advantage of inline function is it makes the program faster.

Why use Inline function

Whenever we call any function many time then, it take a lot of extra time in execution of series of instructions such as saving the register, pushing arguments, returning to calling function. For solve this problem in C++ introduce inline function.

Syntax:

```
inline return-type function-name(arguments)
{
        return ( conditions );
}
```

Example:

```
class S
{
     public:
          int square(int s);        // declare the function
};

inline int S::square(int s)          // use inline prefix
{

}
```

Advantage of inline function is that it will increase the speed of program execution. You should declare the function as inline only when the function contains two or three statements. **Disadvantage** of inline function is that each time a function is called it will replace by function definition so extra memory space is occupied at the time of executing the program.

Where inline functions not work?

- If inline function are recursive
- If function contain static variables.
- If return statement are exits but not return any value.
- If function contains loop control statements

Example:

```
#include<iostream>
using namespace std;
class operation
{
    int a,b,add,sub,mul;
    double div;
public:
    void get();
    void sum();
```

```
    void difference();
    void product();
    void division();
};

inline void operation :: get()
{
    cout << "Enter first value:";
    cin >> a;
    cout << "Enter second value:";
    cin >> b;
}

inline void operation :: sum()
{
    add = a+b;
    cout << "Addition of two numbers: " << add << "\n";
}

inline void operation :: difference()
{
    sub = a-b;
    cout << "Difference of two numbers: " << sub << "\n";
}

inline void operation :: product()
{
    mul = a*b;
    cout << "Product of two numbers: " <<mul<< "\n";
}

inline void operation ::division()
{
    div=(double)a/b;
    cout<<"Division of two numbers: "<<div<<"\n" ;
}
int main()
{
    cout << "Program using inline function\n";
    operation s;
    s.get();
    s.sum();
    s.difference();
    s.product();
    s.division();
    return 0;
}
```

3.5 Const Function

- A function becomes const when const keyword is used in function's declaration.
- The idea of const functions is not allow them to modify the object on which they are called.
- It is recommended practice to make as many functions const as possible so that accidental changes to objects are avoided.

```
#include<iostream>
using namespace std;
class Test {
    int value;
public:
    Test(int v = 0) {
        value = v;
    }

    // We get compiler error if we add a line
    // like "value = 100;" in this function.
    int getValue() const {
        return value;
    }
};

int main() {
    Test t(20);
    cout<<t.getValue();
    return 0;
}
```

3.6 Default Arguments

A default argument is a value provided in function declaration that is automatically assigned by the compiler if caller of the function doesn't provide a value for the argument with default value. Default argument is the argument that is assigned value at the time of declaring the function. It is not compulsory to pass value for the default argument at the time of calling the function, it will automatically assign default value that is specified at the time of declaring the function. However if you specify value for the default argument explicitly at the time of calling the function it will override the default value. Default arguments are widely used in the situation where value of some arguments always remains same. Default arguments are always specified from right to left in the function prototype.

- Example:
  - `void demo(int a, int b=0);`
  - `int show(int a, int b, int c=5,int d=2);`
  - `float disp(float a, int b, char c, float f=5.5, int d=0)`
- Invalid examples:
  - `void demo(int a, int b=5, int d);`
  - `int show(float f=5.5, int a, int b);`

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

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
class DefaultArgs
{
public:
    int sum(int x, int y, int z=0, int w=0)
    {
        return (x + y + z + w);
    }
};
/* Drier program to test above function*/
int main()
{
    DefaultArgs obj;
    cout << obj.sum(10, 15) << endl;
    cout << obj.sum(10, 15, 25) << endl;
    cout << obj.sum(10, 15, 25, 30) << endl;
    return 0;
}
```

3.7 Function Overloading

Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as **method overloading**.

- Overloading means using same thing for different purpose.
- The process of using same function for different purpose is known as Function Overloading.
- In Function Overloading you can create more than one function having same name but with
  (1) Different Number of Arguments
  (2) Different Data Types
- When you call the function, appropriate version of the function is invoked at compile time depending upon the number and type of arguments.
- Function overloading is the best example of **compile time polymorphism**
- **Note:** The scope of overloading is within the class only.
- 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.

Why method Overloading

Suppose we have to perform addition of given number but there can be any number of arguments, if we write method such as a(int, int)for two arguments, b(int, int, int) for three arguments then it is very difficult for you and other programmer to understand purpose or behaviors of method they cannot identify purpose of method. So we use method overloading to easily figure out the program. For example above two methods we can write sum(int, int) and sum(int, int, int) using method overloading concept.

Syntax:
```
class  class_Name
      {
           Returntype  method()
           {
                .....
                .....
           }
           Returntype  method(datatype1 variable1)
           {
                .....
                .....
           }
           Returntype  method(datatype1 variable1, datatype2 variable2)
           {
                .....
                .....
           }
       };
```

3.7.1 By changing number of arguments

In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.

```
#include<iostream.h>
#include<conio.h>

class Addition
{
     public:
          void sum(int a, int b)
          {
               cout<<a+b;
          }
          void sum(int a, int b, int c)
          {
               cout<<a+b+c;
          }
};
void main()
{
     clrscr();
     Addition obj;
     obj.sum(10, 20);
     cout<<endl;
     obj.sum(10, 20, 30);
}
```

3.7.2 By changing the data type

In this example, we have created two overloaded methods that differs in data type. The first sum method receives two integer arguments and second sum method receives two float arguments.

```
#include<iostream.h>
#include<conio.h>
class Addition
{
      public:
            void sum(int a, int b)
            {
                  cout<<a+b;
            }
            void sum(float a, float b)
            {
                  cout<<a+b;
            }
};
void main()
{
      clrscr();
      Addition obj;
      obj.sum(10, 20);
      cout<<endl;
      obj.sum(10.5, 20.03);
}
```

Example:

```
#include <iostream>
using namespace std;
class FunctOver
{
public:
    void print(int i) {
     cout << " Here is int " << i << endl;
    }
    void print(double  f) {
      cout << " Here is float " << f << endl;
    }
    void print(string c) {
      cout << " Here is string " << c << endl;
    }
    void print(int i,int j) {
      cout << " Here is i+j " << (i+j) << endl;
    }
};
```

```
int main() {
  FunctOver obj;
  obj.print(10);
  obj.print(10.10);
  obj.print("ten");
  obj.print(5,5);
  return 0;
}
```

3.8 Friend Functions

In C++, a **Friend Function** that is a "friend" of a given class is allowed access to private and protected data in that class. A function can be made a friend function using keyword friend. Any friend function is preceded with **friend** keyword. The declaration of friend function should be made inside the body of class (can be anywhere inside class either in private or public section) starting with keyword friend.

- A friend function is a function that is not a member of a class but it can access private and protected member of the class in which it is declared as friend.
- Since friend function is not a member of class it cannot be accessed using object of the class.
- It is called in the same way as normal external function is called.
- It works same as your real life friend.
- A friend function can be: a) A method of another class or b) A global function

Syntax:

```
class class_name
{
        ......  ....  ........
        friend return_type function_name(arguments);
        ......  ....   ........
}
```

- Now, you can define friend function of that name and that function can access the private and protected data of that function.
- No keywords in used in function definition of friend function.

Friend function having following characteristics:

- A friend function can be declared inside class but it is not member of the class.
- It can be declared either public or private without affecting its meaning.
- A friend function is not a member of class so it is not called using object of the class. It is called like normal external function.
- A friend function accepts object as an argument to access private or public member of the class.
- A friend function can be declared as friend in any number of classes.

```
#include <iostream>

class A
{
    int a;
public:
    A() {a = 170;}
    friend void showA(A); // global friend function
};

void showA(A x) {
    // Since showA() is a friend, it can access
    // private members of A
    std::cout << "A::a=" << x.a;
}

int main()
{
    A a;
    showA(a);
    return 0;
}
```

3.8.1 Friend Class

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.

Syntax:

```
class A
{
        friend class B; // class B is a friend class
        ......
}
class B
{
        ......
}
```

When a class is made a friend class, all the member functions of that class becomes friend function. If B is declared friend class of A then, all member functions of class B can access private and protected data of class A but, member functions of class A cannot private and protected data of class B.

```
#include <iostream>
class A
{
private:
    int a;
public:
    A() { a=50; }
    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;
}
```

3.9 Virtual Functions

A **virtual function** is a member function of class that is declared within a base class and re-defined in derived class. When you want to use same function name in both the base and derived class, then the function in base class is declared as virtual by using the **virtual** keyword and again re-defined this function in derived class without using virtual keyword.

Syntax:

```
virtual return_type function_name()
{
        .......
        .......
}
```

A virtual function is a special form of member function that is declared within a base class and redefined by a derived class. The keyword virtual is used to create a virtual function, precede the function's declaration in the base class. If a class includes a virtual function and if it gets inherited, the virtual class redefines a virtual function to go with its own need. In other words, a virtual function is a function which gets override in the derived class and instructs the C++ compiler for executing late binding on that function. A function call is resolved at runtime in late binding and so compiler determines the type of object at runtime.

```
#include<iostream>
using namespace std;
class A
{
    public:
        virtual void show()
        {
            cout<<"\nHello base class";
        }
};

class B : public A
{
    public:
        void show()
        {
            cout<<"\nHello derive class";
        }
};

int main()
{
    A aobj;
    B bobj;
    A *bptr;

    bptr=&aobj;
    bptr->show();  // call base class function

    bptr=&bobj;
    bptr->show();  // call derive class function
    return 0;
}
```

3.10 Math Library Functions

Header <cmath> or <math.h> declares a set of functions to compute common mathematical operations and transformations. Following are the different functions provided by math.h library, It provides a variety of functions for solving common math problems.

Some Mathematical Functions in math.h			
Function	**Prototype**	**Description**	**Example**
acos	double acos(double arg)	The acos() function returns the arc cosine of arg. The argument to acos() must be in the	double val = -0.5; cout<<acos(val);

		range -1 to 1; otherwise, a domain error occurs.	
asin	double asin(double arg)	The asin() function returns the arc sine of arg. The argument to asin() must be in the range -1 to 1	double val = -10; cout<<asin(val);
atan	double atan(double arg)	The atan() function returns the arc tangent of arg	atan(val); (val is a double type identifier)
atan2	double atan2(double b, double a)	The atan2() function returns the arc tangent of b/a.	double val = -10; cout<<atan2(val, 1.0);
ceil	double ceil(double num)	The ceil() function returns the smallest integer represented as a double not less than num	ceil(1.03) gives 2.0 ceil (-1.03) gives -1.0
cos	double cos(double arg)	The cos() function returns the cosine of arg. The value of arg must be in radians	cos(val) (val is a double type identifier)
cosh	double cosh(double arg)	The cosh() function returns the hyperbolic cosine of arg. The value of arg must be in radians	cosh(val); (val is a double type identifier)
exp	double exp(double arg)	The exp() function returns the natural logarithm e raised to the arg power	exp(2.0) gives the value of e^2
fabs	double fabs(double num)	The fabs() function returns the absolute value of num	fabs(1.0) gives 1.0 fabs(-1.0) gives 1.0
floor	double floor(double num)	The floor() function returns the largest integer (represented by double) not greater than num	floor(1.03) gives 1.0 floor(-1.03) gives -2.0
fmod	double fmod(double x, double y)	The fmod() function returns the remainder of the division x/y	fmod(10.0, 4.0) returns 2.0
log	double log(double num)	The log() function returns the natural logarithm for num. A domain error occurs if num is negative and a range error occurs if the argument num is zero	log(1.0) gives the natural logarithm for 1.0
log10	double log10(double num)	The log10() function returns the base 10 logarithm for num. A domain error occurs if num is negative and a range error occurs if the argument is zero	log10(1.0) gives base 10 logarithm for 1.0

pow	double pow(double base, double exp)	The pow() function returns base raised to exp power i.e., base exp. A domain error occurs if base = 0 and exp <= 0. Also if base < 0 and exp is not integer.	pow(3.0, 0) gives value of 3^0 pow(4.0, 2.0) gives value of 4^2
sin	double sin(double arg)	The sin() function returns the sin of arg. The value of arg must be in radians	sin(val) (val is a double type identifier)
sinh	double sinh(double arg)	The sinh() function returns the hyperbolic sine of arg. The value of arg must be in radians	sinh(val) (val is a double type identifier)
sqrt	double sqrt(double num)	The sqrt() function returns the square root of num. If num < 0, domain error occurs	sqrt(81.0) gives 9.0
tan	double tan(double arg)	The tan() function returns the tangent of arg. The value of arg must be in radians	tan(val)
tanh	double tanh(double arg)	The tanh() function returns the hyperbolic tangent of arg. The value of arg must be in radians	tanh(val)

```
#include<iostream.h>
#include<math.h>

void main()
{
      short int si = 100;
      int i = -1000;
      long int li = 1300;
      float f = 230.47;
      double d = 200.347;

   cout<<"sqrt(si): "<<sqrt(si)<<endl;
   cout<<"pow(li, 3): "<<pow(li, 3)<<endl;
      cout<<"sin(d): "<<sin(d)<<endl;
      cout<<"abs(i) : "<<abs(i)<<endl;
      cout<<"floor(d): "<<floor(d)<<endl;
      cout<<"sqrt(f): "<<sqrt(f)<<endl;
      cout<<"pow(d, 2): "<<pow(d, 2)<<endl;

      getch();
}
```

Chapter 04
Constructors & Destructors

4.1 Constructors in C++

A constructor is a special member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object (instance of class) is created. It is special member function of the class because the name of the constructor is same as name of the class. Constructor is used to construct the values for the data member of the class automatically when the object of class is created. Like other member function there is no need to call constructor explicitly. It is invoked automatically each time the object of its class is created. Every class having at least one constructor defined in it. If you do not define any constructor in the class then compiler will automatically create a constructor inside the class and assigns default value (0) to the data member of the class.

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

Syntax of Constructor:

```
class_name ( parameter list if any )
{
      body of the constructor
}
```

Following are important characteristics of Constructor:

(1) It is called automatically when object of its class is created.
(2) It does not return any value.
(3) It must be defined inside public section of the class.
(4) It can have default arguments.
(5) Inheritance of constructor is not possible.
(6) It cannot be virtual.

1. Constructor can be defined inside class as shown below:

```
class Rectangle
{
      int Height, Width;
public:
      Rectangle ()
      {
            Height = 10;
            Width = 10;
      }
}
```

2. Constructor can be defined outside class as shown below:

```
class Rectangle
{
        int Height, Width;
public:
        Rectangle ();
}

Rectangle :: Rectangle ()
{
        Height = 1;
        Width = 1;
}
```

4.2 Types of Constructors:

Constructors are of three types:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

1. Default Constructor

Default constructor is the constructor which doesn't take any argument. It has no parameters.

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

class construct
{
public:
   int a, b;

   // Default Constructor
   construct()
   {
      a = 10;
      b = 20;
   }
};

int main()
{
   // Default constructor called automatically when the object is created
   construct c;
   cout << "a: "<< c.a << endl << "b: "<< c.b;
   return 1;
}
```

Note: Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly. The default value of variables is 0 in case of automatic initialization.

Disadvantage of default constructor is that each time an object is created it will assign same default values to the data members of the class. It is not possible to assign different values to the data members for the different object of the class using default constructor.

2. Parameterized Constructor:

However sometimes it is required to assign different values to the data members for the different object of the class. This can be accomplished using the concept of parameterized constructor. The constructor that accepts parameters as an argument is called **Parameterized constructor**. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function.

Parameterized Constructor can be defined inside class as shown below:

```
class Rectangle
{
        int Height, Width;
public:
        Rectangle (int h, int w)
        {
                Height = h;
                Width = w;
        }
}
```

Parameterized Constructor can be defined outside class as shown below:

```
class Rectangle
{
        int Height, Width;
public:
        Rectangle (int h, int w);
}

Rectangle :: Rectangle (int h, int w)
{
        Height = h;
        Width = w;
}
```

In order to invoke parameterized constructor we need to pass arguments while creating object. We can pass arguments using two different methods:

(1) Implicit:
Rectangle R1 (10, 20);
It will assign value of 10 to Height and value of 20 to Width.

(2) Explicit:

Rectangle R2 = Rectangle (30, 40);

It will assign value of 30 to Height and value of 40 to Width.

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

class Point
{
    private:
        int x, y;
    public:
        // Parameterized Constructor
        Point(int x1, int y1)
        {
            x = x1;
            y = y1;
        }

        int getX()
        {
            return x;
        }
        int getY()
        {
            return y;
        }
    };

int main()
{
    // Constructor called
    Point p1(10, 15);

    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    return 0;
}
```

Uses of Parameterized constructor:

1. It is used to initialize the various data elements of different objects with different values when they are created.
2. It is used to overload constructors.

Can we have more than one constructors in a class?

Yes, It is called Constructor Overloading.

3. Copy Constructor

A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

ClassName (const ClassName &old_obj);

The constructor that accepts reference to the object as an argument is known as copy constructor.

Copy Constructor can be defined inside class as shown below:

```
class Rectangle
{
        int Height, Width;
public:
        Rectangle (Rectangle &r)
        {
                Height = r.Height;
                Width = r.Width;
        }
}
```

Copy Constructor can be defined outside class as shown below:

```
class Rectangle
{
        int Height, Width;
public:
        Rectangle (Rectangle &r);
}

Rectangle :: Rectangle (Rectangle &r)
{
        Height = r.Height;
        Width = r.Width;
}
```

In order to invoke copy constructor we need to pass object as an arguments while creating object. We can pass arguments using two different methods:

(1) Implicit:
Rectangle R2 (R1);
It will assign value of data member of object R1 into data member of object R2.

(2) Explicit:
Rectangle R2 = Rectangle (R1);
It will assign value of data member of object R1 into data member of object R2.

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

class Point
{
private:
    int x, y;
public:
    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor
    Point(const Point &p1) {x = p1.x; y = p1.y; }

    int getX()          {  return x; }
    int getY()          {  return y; }
};

int main()
{
    Point p1(10, 15);           // Normal constructor is called here
    Point p2 = p1;              // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;
}
```

When is copy constructor called?

In C++, a Copy Constructor may be called in following cases:
1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When compiler generates a temporary object.

4.3 Constructor Overloading:

In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading and is quite similar to function overloading.

- Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

```
#include<iostream.h>
#include<conio.h>

using namespace std;

class Rectangle
{
        int Height, Width;
public:
        Rectangle ();                           // Default Constructor
        Rectangle (int h, int w);               // Parameterized Constructor
        Rectangle (Rectangle &r);               // Copy Constructor

        void Display(void)
        {
                cout<< "Height="<<Height<<"Width="<<Width<<endl;
        }
};

Rectangle :: Rectangle ()
{
        Height = 0;
        Width = 0;
}

Rectangle :: Rectangle (int h, int w)
{
        Height = h;
        Width = w;
}

Rectangle:: Rectangle (Rectangle &r)
{
        Height = r. Height;
        Width = r. Width;
}

int main(void)
{
        clrscr();
        Rectangle R1;
        R1. Display ();
        Rectangle R2 (21, 42);
        R2. Display ();
        Rectangle R3 (R2);
        R3. Display ();
        getch ();
        return (0);
}
```

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

class construct
{

public:
   float area;

   // Constructor with no parameters
   construct()
   {
      area = 0;
   }

   // Constructor with two parameters
   construct(int a, int b)
   {
      area = a * b;
   }

   void disp()
   {
      cout<< area<< endl;
   }
};

int main()
{
   // Constructor Overloading with two different constructors of class name
   construct o;
   construct o2( 10, 20);

   o.disp();
   o2.disp();
   return 1;
}
```

4.4 Destructors

Destructor is a member function which destructs or deletes an object. Destructor is a member function of the class. It is called **special member function** because the name of the destructor is same as name of the class but it is preceded by the **tilde (~)** sign. Destructor is used to **destroy** the object that is created using constructor. Like other member function there is no need to call destructor explicitly. It is invoked automatically each time the object of its class is destroyed. It is invoked when the control of the program exist from the scope in which the object is created. A destructor does not accept any argument and it does not return any value.

When is destructor called?

A destructor function is called automatically when the object goes out of scope:
(1) the function ends
(2) the program ends
(3) a block containing local variables ends
(4) a delete operator is called

How destructors are different from a normal member function?

Destructors have same name as the class preceded by a tilde (~). Destructors don't take any argument and don't return anything.

Destructor can be defined inside class as shown below:

```
class Rectangle
{
        int Height, Width;
public:
        Rectangle ()
        {
                Height = 1;
                Width = 1;
                cout << "Object Created";
        }

        ~Rectangle ()
        {
                cout << "Object Destroyed";
        }
}
```

Destructor can be defined outside class as shown below:

```
class Rectangle
{
        int Height, Width;
public:
        Rectangle ();
        ~Rectangle ();
}

Rectangle :: Rectangle ()
{
        Height = 1;
        Width = 1;
        cout << "Object Created";
}

Rectangle :: ~Rectangle ()
{
        cout<<"Object Destroyed";
}
```

```
class StringDemo
{
private:
   char *s;
   int size;
public:
   StringDemo(char *); // constructor
   ~StringDemo();      // destructor
};

StringDemo::StringDemo(char *c)
{
   size = strlen(c);
   s = new char[size+1];
   strcpy(s,c);
}

StringDemo::~StringDemo()
{
   delete []s;
}
```

Can there be more than one destructor in a class?
No, there can only one destructor in a class with classname preceded by ~, no parameters and no return type.

Can a destructor be virtual?
Yes, In fact, it is always a good idea to make destructors virtual in base class when we have a virtual function. See virtual destructor for more details.

4.5 Constructors with Default Arguments

Function with Default Argument: Default argument is the argument that is assigned value at the time of declaring the function. It is not compulsory to pass value for the default argument at the time of calling the function; it will automatically assign default value that is specified at the time of declaring the function. However if you specify value for the default argument explicitly at the time of calling the function it will override the default value. Default arguments are widely used in the situation where value of some arguments always remains same. Default arguments are always specified from right to left in the function prototype.

Constructor with Default Argument: A constructor like a member function can have default arguments. The default arguments should be declared at the end of the parameters list. Following program demonstrates a constructor with default arguments.

```
#include <iostream>
using namespace std;
class Demo
{
   private:
      int X,Y;
   public:
      Demo()
      {
            X = 0;
            Y = 0;

            cout << endl << "Constructor Called";
      }

      Demo(int X, int Y=20)  //y is default argument
      {
            this->X = X;
            this->Y = Y;
            cout << endl << "Constructor Called";
      }


      ~Demo()
      {
            cout << endl << "Destructor Called" << endl;
      }

      void putValues()
      {
            cout << endl << "Value of X : " << X;
            cout << endl << "Value of Y : " << Y << endl;
      }
};

int main()
{
   Demo d1= Demo(10);

   cout << endl <<"D1 Value Are : ";
   d1.putValues();

   Demo d2= Demo(30,40);

   cout << endl <<"D2 Value Are : ";
   d2.putValues();
   return 0;
}
```