Inheritance

Sisoft Technologies Pvt Ltd
SRC E7, Shipra Riviera Bazar, Gyan Khand-3, Indirapuram, Ghaziabad
Website: www.sisoft.in Email:info@sisoft.in
Phone: +91-9999-283-283

Reusability is the one of the best feature of OOP. C++ strongly support the concept of reusability. C++ provide the concept of Inheritance to reuse the class in several way.

Inheritance is one of the key feature of object-oriented programming which allows user to create a new class(derived class) from a existing class(base class). When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused also derived class have an additional features.

The class whose properties are inherited by other class is called the Parent or Base or Super class. And, the class which inherits properties of other class is called Child or Derived or Sub 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.

Purpose of Inheritance:

1) Code Reusability
2) Method Overriding (Hence, Runtime Polymorphism.)
3) Use of Virtual Keyword

Basic Syntax of Inheritance:

class Subclass_name : access_mode Superclass_name

While defining a subclass, It is compulsory that the super class must be already defined or at least declared before the subclass declaration.

Access Mode is used to specify, the mode in which the properties of superclass will be inherited into subclass, public, private or protected

Types of Inheritance

There are 5 types of inheritance in C++.

1) **Single Inheritance:** When one class derived form one base class, is called Single Inheritance.

2) **Multiple Inheritance**: When more than one base class derived a class , called Multiple Inheritance.

3) **Hierarchical Inheritance:** When more than one derived class inherit from a common base class, called Hierarchical Inheritance.

4) **Multilevel Inheritance:** The mechanism of deriving a class from another derived class is known as Multilevel Inheritance.

5) **Hybrid Inheritance**: When one or more types of inheritance are combined together and used, is called Hybrid Inheritance.

Defined Derived Classes:

A derived class can be defined by specifying its relationship with the base class in addition to its own details.

Syntax: class derived_ class : visibility_mode base_class

```
{
……
……
};
```

Here the colon indicates that the derived class is derived from the base class. The visibility mode is optional. It may be private, public or protected. By default it is private.

Visibility mode specifies whether the feature of base class are privately inherited, publicly inherited or protected inherited.

Implementation of Inheritance :

```
class Rectangle
{
 ... .. ...
 };
class Area : public Rectangle   // public derivation
 { ... .. ...
 };
class Perimeter : Rectangle     // private derivation
 {
 .... .. ...
};
```

Inheritance Visibility Mode

As we know that the visibility mode may be private, public or protected. Depending on these access modifier , the availability of class members of Super class in the sub class changes . We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied:

1) **Public Inheritance:** When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.

2) **Protected Inheritance:** When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.

3) **Private Inheritance:** When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Table showing all the Visibility Modes:

Access in Base Class	Base Class Inherited as	Access in Derived Class
Public Protected Private	Public	Public Protected No access
Public Protected Private	Protected	Protected Protected No access
Public Protected Private	Private	Private Private No access

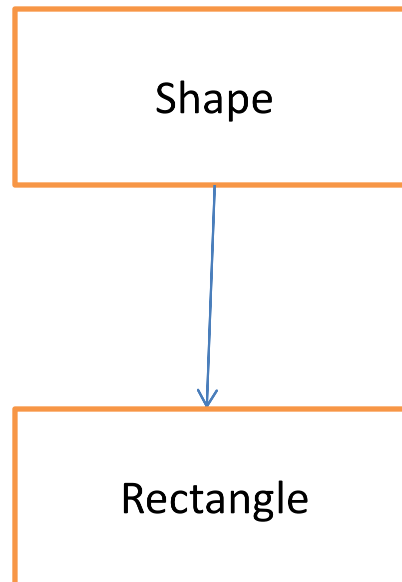
A derived class inherits all base class methods with the following exceptions:

1) Constructors, destructors and copy constructors of the base class.
2) Overloaded operators of the base class.
3) The friend functions of the base class.

Single Inheritance:

In Single Inheritance one class derived form one base class.

Example:



```
class shape
{
protected:
   int width ,height;
public:
   void setwidth (int w)
   {
      width = w ;
      cout<<"Width is\n"<<width<<endl;
   }

   void setheigth (int h)
   {
      height = h ;
      cout<<"Height is\n"<<height<<endl;
   }
};
```

```
class Rectangle: public shape
{
public:
   int getarea ()
   {
      return width * height ;
   }
};
int main()
      {
         Rectangle r;
         r . setwidth(25);
         r . setheigth(12);
 cout<< "Total area is \n" << r . getarea ();
      }
```

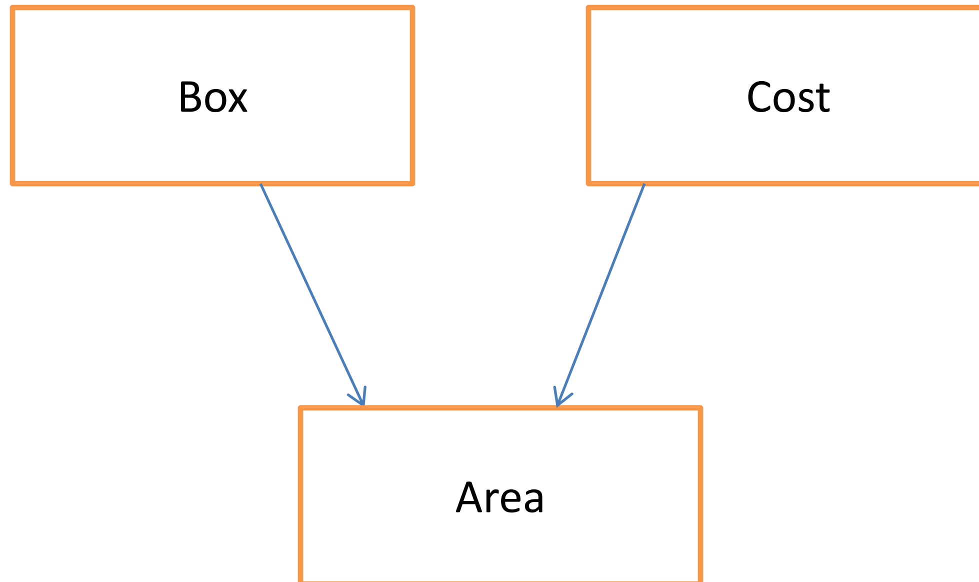
Output:

Width is 25
Height is 12
Total area is 300

Multiple Inheritance:

In Multiple Inheritance more than one base class derived a class.

Example:



```
class box
{
protected:
   int width ,length;
public:
   void setdata (int w , int l)
   {
      width = w ;
      length = l ;
      cout<<"Width is\n"<<width<<endl;
      cout<<"Length is\n"<<length<<endl;
   }
};
class Cost
{
public:
   int totalcost ( int area)
   {
      return area * 70 ;
   }
};
class boxarea : public box , public Cost
{
public:
   int totalarea()
   {
      return (width * length);
   }
 };
int main()
      {
         boxarea r;
         int area;
         r . setdata (20,15);
         area = r . totalarea();
cout<< "Total area is \n" << r . totalarea() <<endl;
cout<< "Total cost is \n" << r . totalcost (area) ;
      }
```

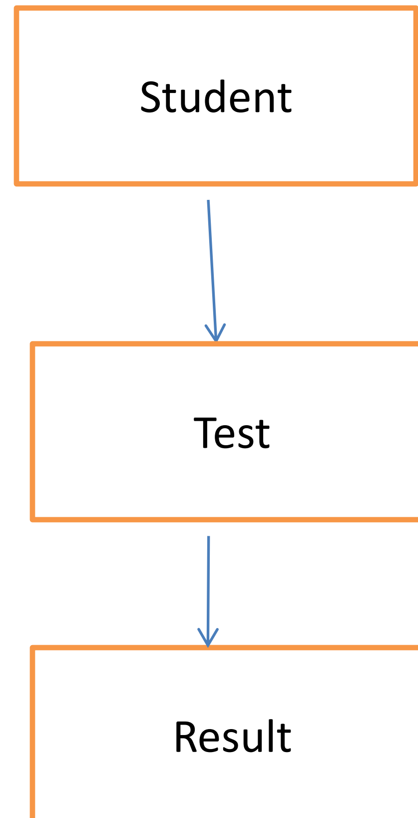
Output:

Width is 20
Length is 15
Total area is 300
Total cost is 21000

Multilevel Inheritance:

The mechanism of deriving a class from another derived class is known as Multilevel Inheritance

Example:



```
class student
{
protected:
   int rollno;
public:
   void getdata(int a)
   {
      rollno=a;
      cout<<"Roll no is\n"<<rollno<<endl;
   }
};
class test: public student
{
protected:
   float emarks , hmarks ;
public:
   void getmarks (float x, float y)
   {
      emarks =x;
      hmarks =y;
 cout<<"Marks is \n"<<"english :"<<"\t"<<
      emarks<<"\n"<< "hindi:"<<"\t“ << hmarks;
   }
};

class result: public test
   {
      float total;
   public:
      void display()
      {
         total= emarks+ hmarks;
    cout<<"Total marks is:\n"<<total<<endl;
      }
   };

   int main()
   {
      result r;
      r . getdata(111);
```

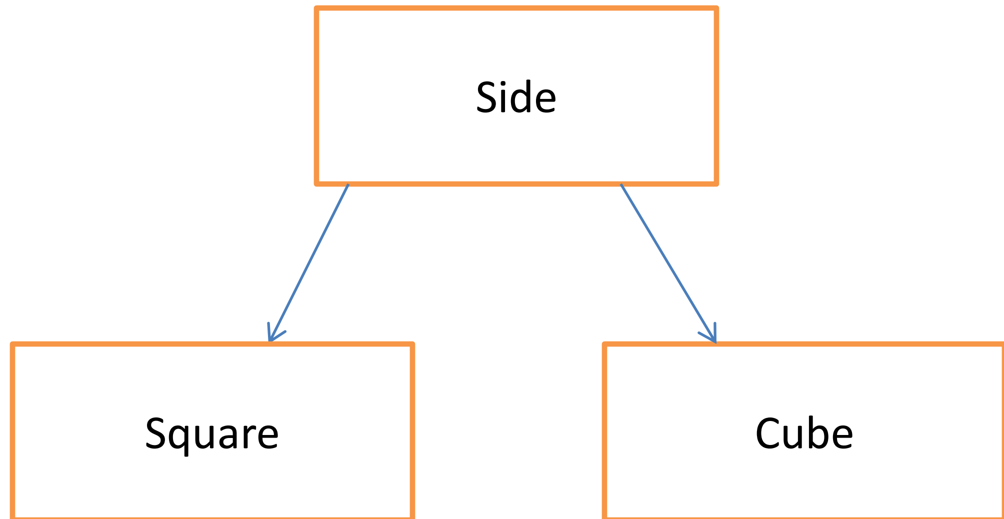
Output:

Roll no is 111
Marks is
English: 234.5
Hindi: 456.7
Total marks is : 691.2

Hierarchical Inheritance:

In Hierarchical Inheritance , more than one derived class inherit from a common base class,

Example:



```
class Side
{
protected:
int l;
public:
void set_values (int x)
{
 l=x;
}
};


class Square: public Side
{
public:
int sq()
{
return (l *l);
}
};
```

```
class Cube :  public Side
{
public:
int cub()
{
return (l *l*l);
}
};


int main ()
{
Square s;
s. set_values (10);
cout << "The square value is::" << s.sq() << endl;
Cube c;
c. set_values (20);
cout << "The cube value is::" << c.cub() << endl;
return 0;
}
```

Output:

The square value is 100
The cube value is 8000

```
class result: public test , public sports
{
   float total;
public:
   void display()
   {
      total= emarks+ hmarks + sports _score;
   cout<<"Total marks is:\n"<<total<<endl;
   }
};

int main()
{
   result r;
   r . getdata(111);
    r . getmarks (234.5, 456.7);
    r . getscore (235.89);
   r . display ();
}
```

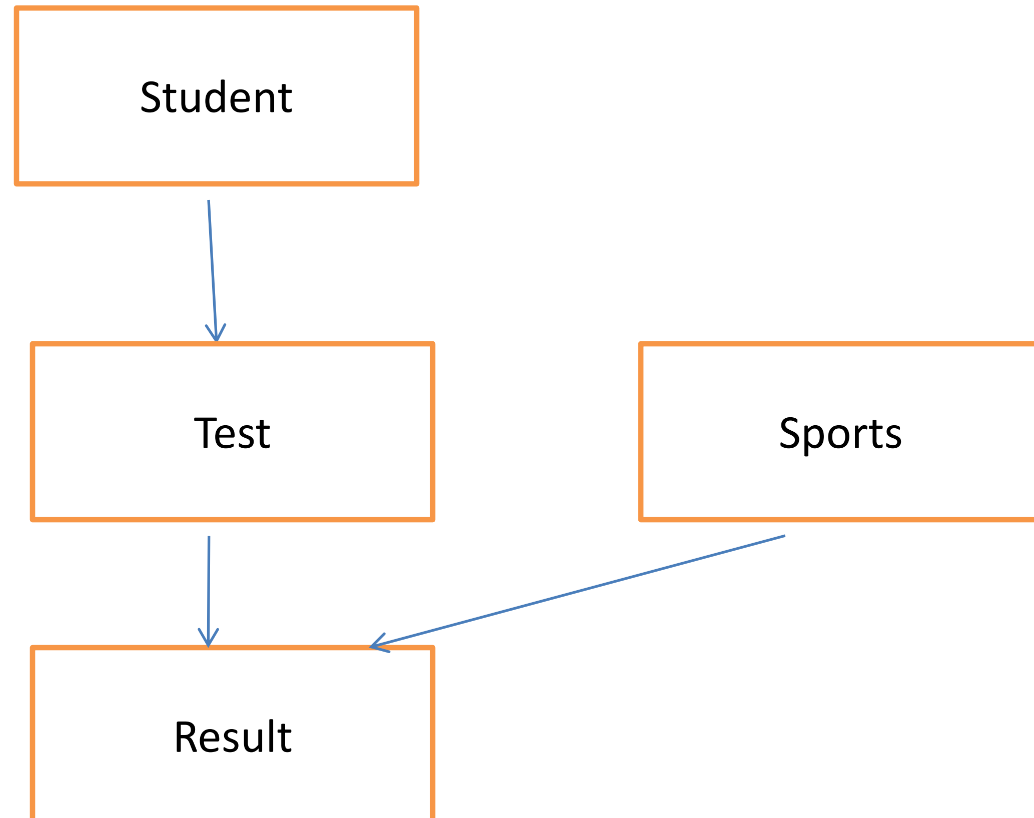
Output:

Roll no is 111
Marks is
English: 234.5
Hindi: 456.7
Sports marks is 235.89
Total marks is : 927.09

Hybrid Inheritance:

In Hierarchical Inheritance , one or more types of inheritance are combined together and used.

Example:



```
class student
{
protected:
   int rollno;
public:
   void getdata(int a)
   {
      rollno=a;
      cout<<"Roll no is\n"<<rollno<<endl;
   }
};
class test: public student
{
protected:
   float emarks , hmarks ;
public:
   void getmarks (float x, float y)
   {
      emarks =x;
      hmarks =y;
      cout<<"Marks is \n"<<"english :"<<"\t"<<
        emarks<<"\n"<< "hindi:"<<"\t“ << hmarks;
   }
};

class sports
{
protected:
   float  sports_score ;
public:
   void getscore (float s)
   {
      sports_score = s;
      cout << " sports marks is :" << sports_score <<
     endl;
   }
};
```

```
class result: public test , public sports
{
   float total;
public:
   void display()
   {
      total= emarks+ hmarks + sports _score;
  cout<<"Total marks is:\n"<<total<<endl;
   }
};

int main()
{
   result r;
   r . getdata(111);
     r . getmarks (234.5, 456.7);
     r . getscore (235.89);
   r . display ();
}
```

Output:

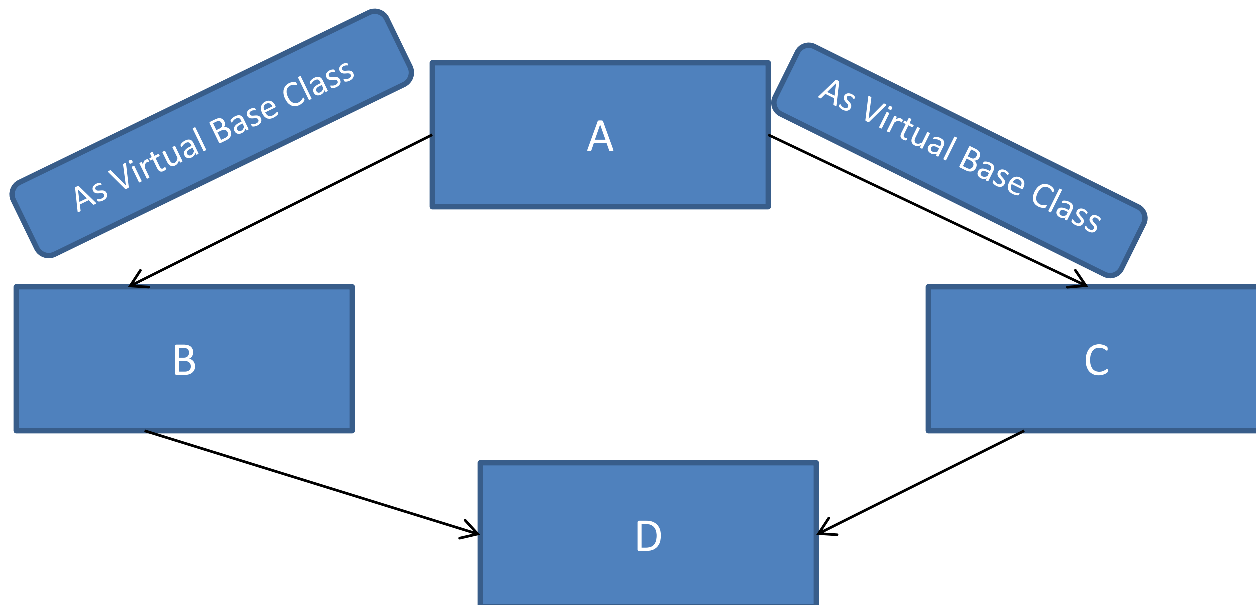
Roll no is 111
Marks is
English: 234.5
Hindi: 456.7
Sports marks is 235.89
Total marks is : 927.09

Virtual Base Class in C++

An ambiguity can arise when several paths exist to a class from the same base class. This means that a child class could have duplicate sets of members inherited from a single base class.

C++ solves this issue by introducing a virtual base class. When a base class is made virtual, C++ necessary care to see that only one copy of that class is inherited , regardless of how many inheritance paths exist between he virtual base class and a derived class.

Example:



```
class A
    {
        public:
            int i;
    };
class B : virtual public A
    {
        public:
                int j;
    };
class C: virtual public A
    {
        public:
            int k;
    };
class D: public B, public C
    {
    public:
        int sum;
    };
```

```
int main()
    {
        D ob;

ob.i = 10; //unambiguous since only one copy of i is inherited.

    ob.j = 20;
     ob.k = 30;
     ob.sum = ob.i + ob.j + ob.k;

 cout << "Value of i is : "<< ob.i<<"\n";
cout << "Value of j is : "<< ob.j<<"\n"; cout << "Value of k is :"<<
    ob.k<<"\n";

cout << "Sum is : "<< ob.sum <<"\n";
    }
```

Output:

i=10 j=20 k=30
Sum=60

Abstract Class in C++

An abstract class is that class, which is not used to create objects. It is designed only to act as a base class which is inherited by other classes or we can say that It is provide an Interface for its sub classes.

An Abstract Class contains must contain one Pure Virtual function in it. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class:

1) Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.

2) It can have normal functions and variables along with a pure virtual function.

3) They are mainly used for Upcasting, so that its derived classes can use its interface.

4) Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Purpose of an Abstract Class:

The purpose of an abstract class is to define a common protocol for a set of concrete subclasses.

Abstract class has no instances. An abstract class must atleast one deferred method. To accomplished this in C++, a pure virtual member function is declared but not defined in the abstract calss.

Syntax:

```
Class A
{
Virtual void add ()=0;                    //   Pure Virtual Function
}
```

This function must be overridden by any concrete class, otherwise they will also become abstract class.

Example:

```
Class  A
{
Virtual void add =0;
Void fun();
}
Class B: public A
{
void add()
{
……..
……..
}
```

```
class Base    //          Abstract base class
{
public:
 virtual void show() = 0;
};

class Derived : public Base
{
 public: void show()
 {
 cout << “HELLO";
}
};
int main()
{
      Base obj;  //Compile Time Error
     Base *b;
     Derived d;
      b = &d;
     b->show();
}
```

In this example Base class is abstract, with pure virtual show() function, hence we cannot create object of base class.

Output: HELLO

Pure Virtual Functions:

Pure virtual Functions are virtual functions with no definition. They start with virtual keyword and ends with 0.

Syntax: virtual void f() = 0;

Pure Virtual definitions:

1) Pure Virtual functions can be given a small definition in the Abstract class, which you want all the derived classes to have. Still you cannot create object of Abstract class .

2) Also, the Pure Virtual function must be defined outside the class definition. If you will define it inside the class definition, complier will give an error. Inline pure virtual definition is Illegal.

```
class Base  //Abstract base class
{
 public:
virtual void show() = 0; //Pure Virtual Function
};

void Base :: show() //Pure Virtual definition
{
cout << "Hello";
}

class Derived : public Base
{
 public:
 void show()
{
 cout << "Bye";
}
};
int main()
{
 Base *b;
Derived d;
 b = &d;
b->show();
}
```

Output : Bye

Function Overriding in C++

In inheritance, polymorphism is done, by method overriding, when both super and sub class have member function with same declaration but different definition.

Function Overriding:

If we inherit a class into the derived class and provide a definition for one of the base class's function again inside the derived class, then that function is said to be overridden and this mechanism is called Function Overriding.

Requirements for Overriding:

1) Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.

2) Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

program:

```
class base
{
 public: void show()
 {
 cout << "base class";
}
};
class derived: public base
{
public:
 void show()
 {
 cout << "derived class";
}}
```

In this example, function **show()** is overridden in the derived class.

Order of Constructor call in C++

Base class constructors are always called in the derived class constructors. Whenever you create derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution.

Points to Remember:

1) Whether derived class's default constructor is called or parameterized is called, base class's default constructor is always called inside them.

2) To call base class's parameterized constructor inside derived class's parameterized constructor, we must mention it explicitly while declaring derived class's parameterized constructor.

Ex 1: Base class default constructor in derived class constructor

```
class Base
{
 int x;
 public: Base()
 {
 cout << "Base default constructor";
}
};
class Derived : public Base
{
 int y;
 public:
Derived()
{
 cout << "Derived default constructor";
 }
 Derived(int i)
 {
 cout << "Derived parameterized constructor";
}};
```

```
int main()
{
 Base b;
Derived d1;
Derived d2(10);
}
```

Output:

Base default constructor
Base default constructor
Derived default constructor
Base default constructor
Derived parameterized constructor

In this example we see that with both the object creation of the Derived class, Base class's default constructor is called.

Ex 2: Base class parameterized constructor in derived class constructor

We can explicitly mention to call the Base class's parameterized constructor when Derived class's parameterized constructor is called.</p><pr

```
class A
{
public:
Int x;
A()
{
Cout<<"You are in A";
}
A(int i)
{
X=i;
Cout<<"You are in A Parameterized";
}
};


Class B: public A
{
public:
Int y;
B(int j):A(j)
{
y=j;
Cout<<"You are in B Parameterized";
}
B()
{
Cout<<"You are in B";
}
};
```

```
void main()
{
A a ;
B b  (10);
B b1;
cout<<b . x <<endl;
cout<<b . y<<endl;
}
```

Output:

You are in A
You are in A Parameterized
You are in B Parameterized
You are in A
You are in B
10
10

Execution of base class constructors:

Method of Inheritance	Order of Execution

Class B: public A
{
};
	A() ; Base Constructor
	B(); Derived Constructor

Class A: public B, public C
{
};
	B() ; Base (First)
	C(); Base (Second)
	A(); Derived

Class A: public B, virtual public C
{
};
	C() ; Virtual Base
	B(); Ordinary Base
	A(); Derived

Why is Base class Constructor called inside Derived class ?

Constructors have a special job of initializing the object properly.

A Derived class constructor has access only to its own class members, but a Derived class object also have inherited property of Base class, and only base class constructor can properly initialize base class members.

Hence all the constructors are called, else object wouldn't be constructed properly.

Constructor called in Multiple Inheritance in C++

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.

The constructors of inherited classes are called in the same order in which they are inherited.

```
class A
{
public:
  A()
 {
cout << "A's constructor called" << endl;
}
};
~A();

class B
{
public:
  B()
{
 cout << "B's constructor called" << endl;
}
};
~B();

class C: public B, public A  // Note the order
{
public:
  C()
{
 cout << "C's constructor called" << endl;
}
};
~C();
```

```
int main()
{
    C c;
    return 0;
}
```

Output:
B's constructor called
A's constructor called
C's constructor called

The destructors are called in reverse order of constructors.

Containership in C++

When a class contains objects of another class or its members, this kind of relationship is called containership or nesting and the class which contains objects of another class as its members is called as container class.

Syntax:

```
Class  A{
——
——
};
Class B
{
——
——
};
Class C
{
A obj1;                 // object of class A
B obj2;               //   object of class B
———-
};
```

Difference between Inheritance & Containership in C++

- In Inheritance, if a class B is derived from class A, B has all characteristics of A in addition to its own . Hence we can say that "B is a kind of A". So, inheritance is sometimes called as a "kind of" relationship.

- Inheritance follow is-a relationship.

- In Containership, A class contains objects of another class or its members. It means, Class B contains the object of class A. Hence we can say that “ B has a object of A.

- Containership follow has-a relationship.

Program:

```
class Department
{
	protected:
	int id;
	char name[50];
public:
	void setDepartment ()
	{
	cout<>id;
	cout<>name;
	}
	void displayDepartment  ()
	{
	cout<<"\n Department ID is:"<<id<<"\n";
	cout<<"\n Department Name
	is:"<<name<<"\n";
	}
	};
class Employee
	{
	protected:
	int eid;
	char ename[50];
	Department dobj;
```

```
public:
void setEmployee ()
	{
	cout<< eid ;
	cout<< ename;
	dobj  . setDepartment();
	}
	void displayEmployee()
	{
	cout<<"\n Employee ID is:"<<eid<<"\n";
	cout<<"\n Employee Name is:"<<ename<<"\n";
	dobj.displayDepartment();
	}
	};
int main()
	{
	Employee obj;
obj.setEmployee();
	obj.displayEmployee();
return 0;
	}
```

Nesting of Classes in C++

Nested class is a class defined inside a class, that can be used within the scope of the class in which it is defined. In C++ nested classes are not given importance because of the strong and flexible usage of inheritance.

A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.

Program:

```
class Nest
      {
      public:
      class Display
      {
      private:
      int s;
      public:
      void sum( int a, int b)
      {
s =a+b;
 }
      void show( )
      {
 cout << "\nSum of a and b is:: " << s;
}
      };
      };
```

```
void main()
      {
      Nest::Display x;
      x.sum(12, 10);
      x.show();
      }
```

Result:

Sum of a and b is::22