Introduction

Inheritance is the process of creating new class ,known as derived class from existing class known as base class. The derived class inherits all the properties of base class and can add its own properties as well. The inherited properties may be hidden (if private in base class) or visible (if public or protected in base class) in derived class.

Sub Class (Derived Class): The class that inherits properties from another class is called Sub class or Derived Class.

Super Class (Base Class): The class whose properties are inherited by sub class is called Base Class or Super class.

Inheritance uses the concept of code reusability. Once a base class is written and debugged, we can reuse the properties of the base class in other classes by using the concept of inheritance. Reusing existing code saves time and effort and increases program reliability.

Example:

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

// Base class
class Vehicle {
  public:
        string brand = "Ford";
        void honk() {
        cout << "Noise:"<<"Tuut, tuut"<<endl ;
        }
};
```

```
// Derived class
class Car: public Vehicle {
  public:
        string model = "Mustang";
};

int main() {
  Car myCar;
  myCar.honk();
  cout << "Brand:"<< myCar.brand <<endl;
  cout<<"Model:"<< myCar.model;
  return 0;
}
```

Output:

```
Noise:Tuut, tuut
Brand:Ford
Model:Mustang
```

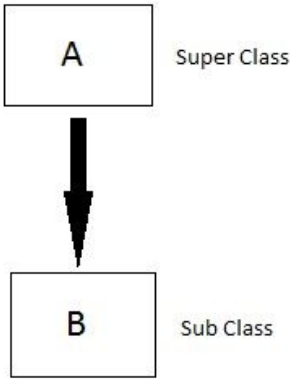
Types of Inheritance

In C++, we have 5 different types of Inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

Single Inheritance in C++

In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



Implementation Skeleton

```
class A

{

  //members of A

}

class B: public A

{

//members of B

}
```

Example:

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

```
class shape{
   public:

        int height, width;

        void getDimension(){
                cout<<"Enter height:"<<endl;
                cin>>height;
                cout<<"Enter width:"<<endl;
                cin>>width;
        }
};

class rectangle: public shape{
   public:
        int area;

        void calculate()
        {
                area=height*width;
                cout<<"Area of rectangle is:"<<area;
        }
};

int main()
{
   rectangle rect;
   rect.getDimension();
   rect.calculate();
   return 0;
}
```

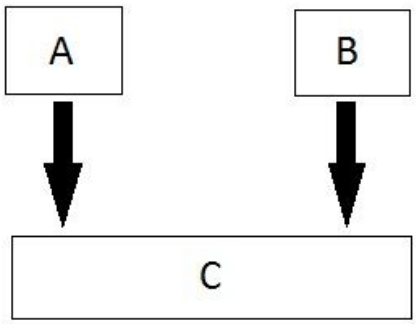
Output:

```
Enter height:
12
Enter width:
12
```

Area of rectangle is:144

Multiple Inheritance in C++

In this type of inheritance a single derived class may inherit from two or more than two base classes.



Implementation Skeleton

```
class A
{
//members of A
}

class B
{
//members of B
}
class C: public A, public B
{
//members of C
}
```

Example:

```
#include<iostream>
using namespace std;
class A
{
	public:
	int x;
	void getx()
	{
		cout << "enter value of x: "; cin >> x;
	}
};
class B
{
	public:
	int y;
	void gety()
	{
		cout << "enter value of y: "; cin >> y;
	}
};
class C : public A, public B   //C is derived from class A and class B
{
	public:

	int sum;
	void add()
	{
		 sum=x+y;
		cout << "Sum = " <<sum ;
	}
};
```

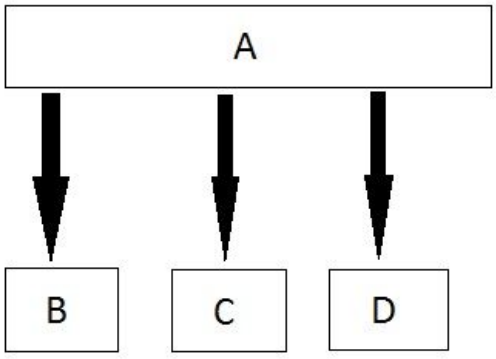
```
int main()
{
        C obj1; //object of derived class C
        obj1.getx();
        obj1.gety();
        obj1.add();
        return 0;
}
```

Output:

```
enter value of x: 12
enter value of y: 21
Sum = 33
```

Hierarchical Inheritance in C++

In this type of inheritance, multiple derived classes inherits from a single base class.



Implementation Skeleton

```
class A
{
//members of A
}

class B : public A
{
//members of B
}
class C: public A
{
//members of C
}
class D: public A
{
//members of D
}
```

Example:

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

class A //single base class
{
        public:
         int x, y;
         void getdata()
         {
                cout << "\nEnter value of x and y:\n"; cin >> x >> y;
         }
```

```
};
class B : public A //B is derived from class base
{
        public:
         void product()
         {
                cout << "\nProduct= " << x * y;
         }
};
class C : public A //C is also derived from class base
{
        public:
         void sum()
         {
        cout << "\nSum= " << x + y;
         }
};
int main()
{
        B obj1;         //object of derived class B
        C obj2;         //object of derived class C
        obj1.getdata();
        obj1.product();
        obj2.getdata();
        obj2.sum();
        return 0;
}
```

Output:

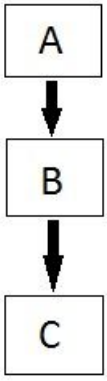
```
Enter value of x and y:
12
21
```

Product= 252
Enter value of x and y:
12
213

Sum= 225

Multilevel Inheritance in C++

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



Implementation Skeleton

```
class A
{
//members of A
}

class B : public A
{
//members of B
}
class C: public B
```

```
{
//members of C
}
```

Example:

```
#include <iostream>
using namespace std;
class base //single base class
{
        public:
        int x;
        void getdata()
        {
        cout << "Enter value of x= "; cin >> x;
        }
};
class derive1 : public base // derived class from base class
{
        public:
        int y;
        void readdata()
        {
                cout << "\nEnter value of y= "; cin >> y;
        }
};
class derive2 : public derive1   // derived from class derive1
{
        private:
        int z;
        public:
        void indata()
        {
```

```
        cout << "\nEnter value of z= "; cin >> z;
        }
        void product()
        {
                cout << "\nProduct= " << x * y * z;
        }
};
int main()
{
        derive2 a;     //object of derived class
        a.getdata();
        a.readdata();
        a.indata();
        a.product();
        return 0;
}
```

Output:

Enter value of x= 12

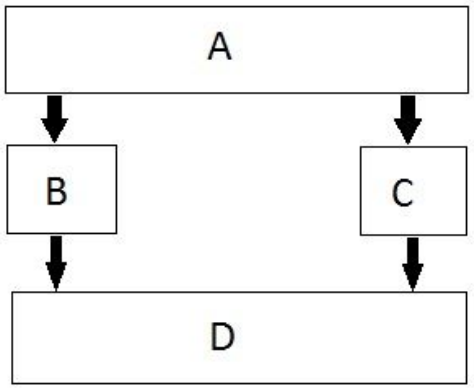
Enter value of y= 12

Enter value of z= 32

Product= 4608

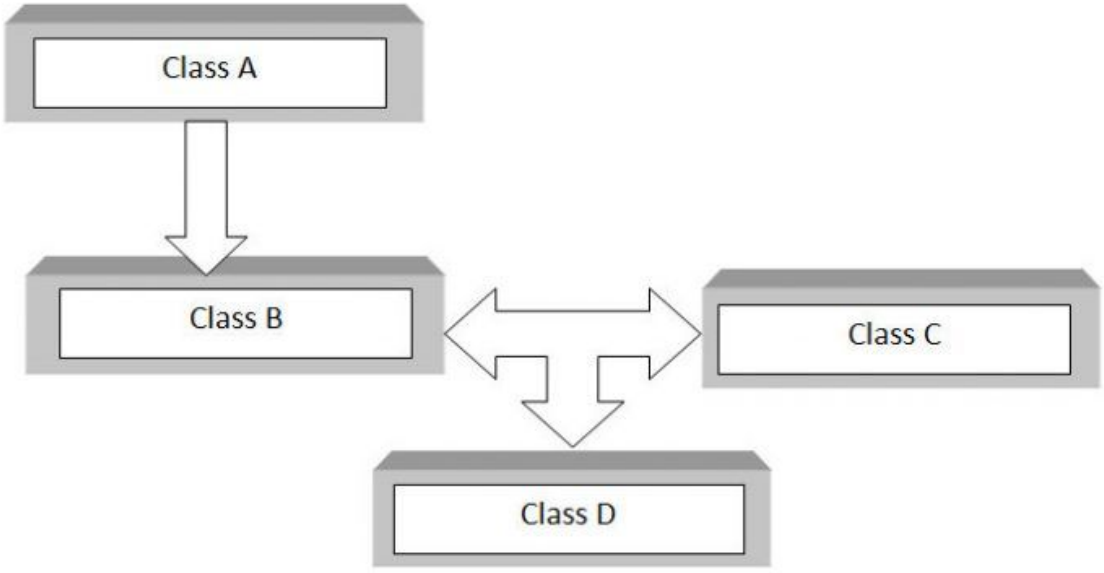
Hybrid (Virtual) Inheritance in C++

Hybrid Inheritance is the combination of more than one form of Inheritance.



Implementation Skeleton

```
Class A
{
//members of A
}

class B: public A
{
//members of B
}

class C: public A
{
//members of C
}

class D:public B ,public C
{
//members of D
}
```



Implementation Skeleton

```
class A
{
    .....
};
class B : public A
{
    .....
} ;
class C
{
    .....
};
 class D : public B, public C
{
    .....
};
```

Example:

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

class A
{
        public:
        int x;
};
class B : public A
{
        public:
        B()     //constructor to initialize x in base class A
        {
                x = 10;
        }
};
class C
 {
        public:
        int y;
        C()   //constructor to initialize y
        {
                y = 4;
       }
};
class D : public B, public C   //D is derived from class B and class C
{
        public:
        void sum()
        {
                cout << "Sum= " << x + y;
        }
```

```
};

int main()
{
        D obj1;        //object of derived class D
         obj1.sum();
         return 0;
}
```

Output

Sum=14

Ambiguity in Multiple Inheritance

Consider an example of multiple inheritances in which there are two base classes having functions with the same name and a class derived from both these base classes having no function with this name. So when we create an object of derived class and try to access these functions then ambiguity occurs.

Example:

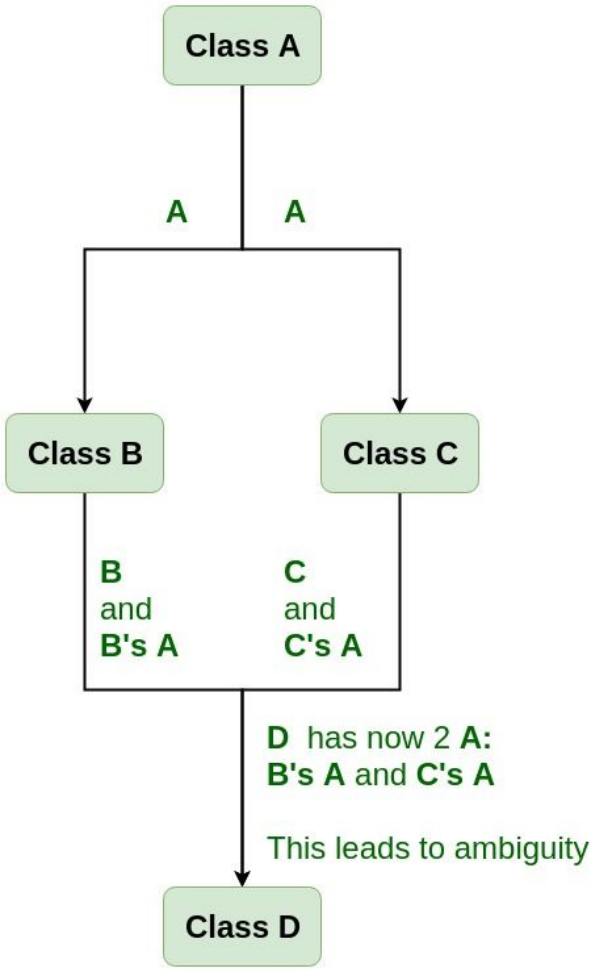
```
#include<iostream>
using namespace std;

class A
{
        public:
        void show ()
        {
                cout<<"From Class A";
        }

};
```

```
class B
{
        public:
        void show ()
        {
                cout<<"From Class B";
         }
};

class C: public A, public B
{

};

int main ()
{
         C obj;
         obj.show ();  //ambiguous
         obj.A::show()   // Ok
        obj.B::show()  // Ok
         return 0;
}
```

Here, the compiler cannot distinguish whether the method show() is of class A or B and hence ambiguity occurs. And this can be solved by using scope resolution operator as above.

Ambiguity in Multipath Inheritance

Consider the situation where we have one class **A** . This class is **A** is inherited by two other classes **B** and **C**. Both these classes are inherited into another in a new class **D** as shown in figure below.



As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called. This ambiguity can be solved by using virtual base class.

Virtual Base Class

Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Syntax for Virtual Base Classes:

Syntax 1:

```
class B : virtual public A
{
};
```

Syntax 2:

```
class C : public virtual A
{
};
```

Example:

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

class A {
public:
   void show()
   {
      cout << "Hello from A \n";
   }
};

class B : public virtual A {
};

class C : public virtual A {
};
```

```
class D : public B, public C {
};

int main()
{
   D object;
   object.show();
}
```

Output:

```
Hello from A
```

Here, D inherits only one copy from B and C.

Abstract Class

A class with **pure virtual function** is known as an abstract class. For example the following function is a pure virtual function:

```
virtual void fun() = 0;
```

A pure virtual function is marked with a virtual keyword and has = 0 after its signature. You can call this function an abstract function as it has no body. The derived class must give the implementation to all the pure virtual functions of parent class else it will become abstract class by default.

Why we need a abstract class?

Let's understand this with the help of a real life example. Lets say we have a class Animal, animal sleeps, animal make sound, etc. For now I am considering only

these two behaviours and creating a class Animal with two functions sound() and sleeping().

Now, we know that animal sounds are different cat says “meow”, dog says “woof”. So what implementation do I give in Animal class for the function sound(), the only and correct way of doing this would be making this function pure abstract so that I need not give implementation in Animal class but all the classes that inherits Animal class must give implementation to this function. This way I am ensuring that all the Animals have sound but they have their unique sound.

Example of Abstract Class

```
#include<iostream>
using namespace std;
class Animal{
public:
  //Pure Virtual Function
  virtual void sound() = 0;

  //Normal member Function
  void sleeping() {
    cout<<"Sleeping";
  }
};
class Dog: public Animal{
public:
  void sound() {
    cout<<"Woof"<<endl;
  }
};
int main(){
  Dog obj;
  obj.sound();
  obj.sleeping();
```

```
  return 0;
}
```

Rules of Abstract Class

1) As we have seen that any class that has a **pure virtual function** is an abstract class.

2) We cannot create the instance of abstract class. For example: If I have written this line Animal obj; in the above program, it would have caused compilation error.

3) We can create pointer and reference of base abstract class points to the instance of child class. For example, this is valid:

```
Animal *obj = new Dog();
obj->sound();
```

4) Abstract class can have constructors.

5) If the derived class does not implement the pure virtual function of parent class then the derived class becomes abstract.

Using constructor and destructor in inheritance

Invocation of constructors and destructors depends on the type of inheritance being implemented. We have presented you the sequence in which constructors and destructors get called in single and multiple inheritance.

Constructor and destructor in single inheritance

- Base class constructors are called first and the derived class constructors are called next in single inheritance.

- Destructor is called in reverse sequence of constructor invocation i.e. The destructor of the derived class is called first and the destructor of the base is called next.

Example:

```
#include<iostream>
using namespace std;
class base
{
public:
   base()
   {
      cout<<"base class constructor"<<endl;
   } ~base()
   {
      cout<<"base class destructor"<<endl;
   }
};
class derived:public base
{
public:
   derived()
   {
      cout<<"derived class constructor"<<endl;
   } ~derived()
   {
      cout<<"derived class destructor"<<endl;
   }
};
int main()
{
   derived d;
```

```
    return 0;
}
```

Output:

```
base class constructor
derived class constructor
derived class destructor
base class destructor
```

Constructor and destructor in single inheritance

- Constructors from all base class are invoked first and the derived class constructor is called.
- Order of constructor invocation depends on the order of how the base is inherited.

For example:

```
class D:public B, public C {
   //...
 }
```

Here, B is inherited first, so the constructor of class B is called first and then constructor of class C is called next.

However, the destructor of derived class is called first and then destructor of the base class which is mentioned in the derived class declaration is called from last towards first in sequentially.

Example:

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

```
{
public:
   base_one()
   {
      cout<<"base_one class constructor"<<endl;
   }
   ~base_one()
   {
      cout<<"base_one class destructor"<<endl;
   }
};
class base_two
{
public:
   base_two()
   {
      cout<<"base_two class constructor"<<endl;
   }
   ~base_two()
   {
      cout<<"base_two class destructor"<<endl;
   }
};
class derived:public base_one, public base_two
{
public:
   derived()
   {
      cout<<"derived class constructor"<<endl;
   }
   ~derived()
   {
      cout<<"derived class destructor"<<endl;
```

```
    }
};
int main()
{
    derived d;
    return 0;
}
```

Output:

```
base_one class constructor
base_two class constructor
derived class constructor
derived class destructor
base_two class destructor
base_one class destructor
```

Derived Class Constructor

If the base class constructor do not have any arguments then the derived class doesn't need to have a constructor function. But if the base class has a constructor with one or more arguments then it is a must for a derived class to have a constructor function and should pass the arguments to base class constructors.

The derived class constructor receives the entire list of values as an argument and passes them on to the base construction in the order that they are declared in derived class.

Example:

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

```
class A
{
   public:
        int a,b,c;

        A(int x, int y, int z)
        {
                a=x;
                b=y;
                c=z;
        }
};

class B: public A
{
   public:
        int d,e,f;

        B(int p, int q,int r,int s, int t, int u):A(p,q,r)
        {
                d=s;
                e=t;
                f=u;
        }

        void display()
        {
                cout<<"a:"<<a<<endl;
                cout<<"b:"<<b<<endl;
                cout<<"c:"<<c<<endl;
                cout<<"d:"<<d<<endl;
                cout<<"e:"<<e<<endl;
                cout<<"f:"<<f<<endl;
```

```
        }
};

int main()
{
   B obj(2,3,4,5,6,7);
   obj.display();
   return 0;
}
```

Output:

```
a:2
b:3
c:4
d:5
e:6
f:7
```

Containership in C++

We can create an object of one class into another and that object will be a member of the class. This type of relationship between classes is known as **containership** or **has_a** relationship as one class contain the object of another class. And the class which contains the object and members of another class in this kind of relationship is called a **container class**.

Syntax for Containership:

```
// Class that is to be contained
class first
{
.....
};

// Container class
class second {

   // creating object of first
   first f;
};
```

Example:

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

class first {
public:
   void showf()
   {
      cout << "Hello from first class\n";
   }
};

class second {
   first f;

public:
   second()
   {
      f.showf();
   }
};

int main()
```

```
{
   second s;
}
```

Output:

Hello from first class

Public, Protected and Private Inheritance

You can declare a derived class from a base class with different access control, i.e., public inheritance, protected inheritance or private inheritance.

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

class base
{
.... ... ....
};

class derived : access_specifier base
{
.... ... ....
};
```

Note: Either public, protected or private keyword is used in place of access_specifier term used in the above code.

Example:

```
class base
{
        public:
                int x;
        protected:
                int y;
        private:
                int z;
};

class publicDerived: public base
{
        // x is public
        // y is protected
        // z is not accessible from publicDerived
};

class protectedDerived: protected base
{
        // x is protected
        // y is protected
        // z is not accessible from protectedDerived
};

class privateDerived: private base
{
        // x is private
        // y is private
        // z is not accessible from privateDerived
}
```

In the above example, we observe the following things:

- Base has three member variables: x, y and z which are public, protected and private member respectively.
- **publicDerived** inherits variables x and y as public and protected. z is not inherited as it is a private member variable of base.
- **protectedDerived** inherits variables x and y. Both variables become protected. z is not inherited

If we derive a class **derivedFromProtectedDerived** from protectedDerived, variables x and y are also inherited by the derived class.

- **privateDerived** inherits variables x and y. Both variables become private. z is not inherited

  If we derive a class **derivedFromPrivateDerived** from **privateDerived**, variables x and y are not inherited because they are private variables of privateDerived.

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)