Unit-III Extending classes using Inheritance

Inheritance is the capability of one class to acquire properties and characteristics from another class. 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.

Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.

Purpose of Inheritance

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

Defining a derived class

Syntax of Inheritance

class Subclass_name : access_mode Superclass_name

While defining a derived class like this, the base class must be already defined or atleast declared before the derived class declaration.

Visibility modes and effects

Access Mode is used to specify, the mode in which the properties of superclass will be inherited into subclass, public, privtate or protected.
Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

1) Public Inheritance

This is the most used inheritance mode. In this the protected member of super class becomes protected members of sub class and public becomes public.

class Subclass : public Superclass

2) Private Inheritance

In private mode, the protected and public members of super class become private members of derived class.

class Subclass : Superclass // By default its private inheritance

3) Protected Inheritance

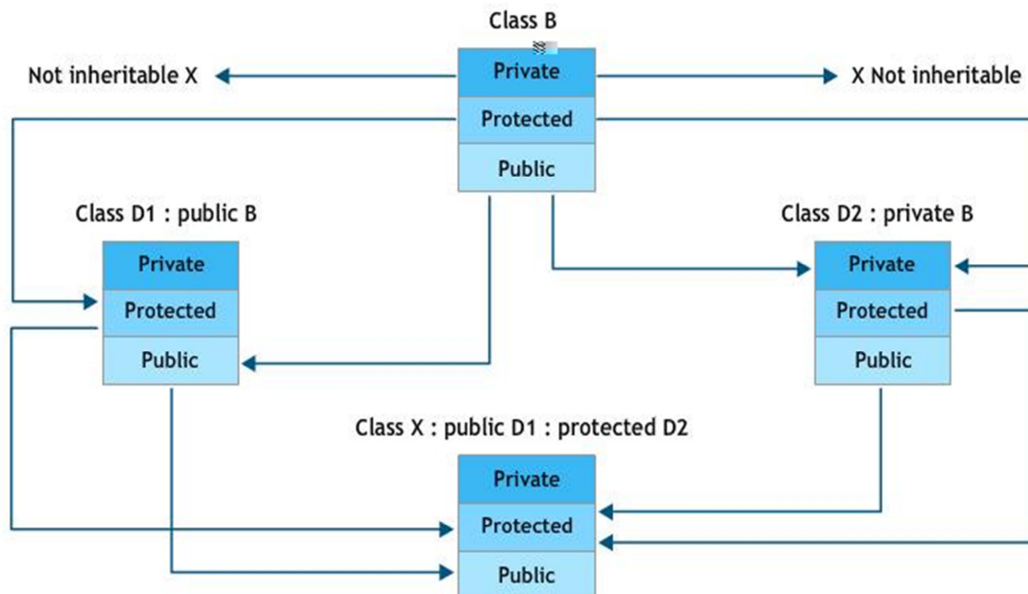
In protected mode, the public and protected members of Super class becomes protected members of Sub class.

class subclass : protected Superclass

Table showing all the Visibility Modes

	Derived Class		
Base class	**Public Mode**	**Private Mode**	**Protected Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

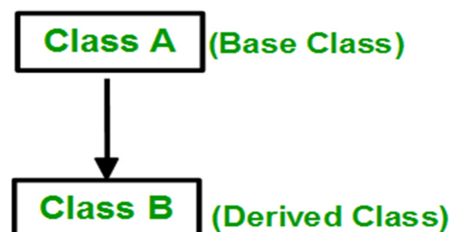
Effects of inheritance on visibility modes



Types of Inheritance :

Single Inheritance

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



Syntax:

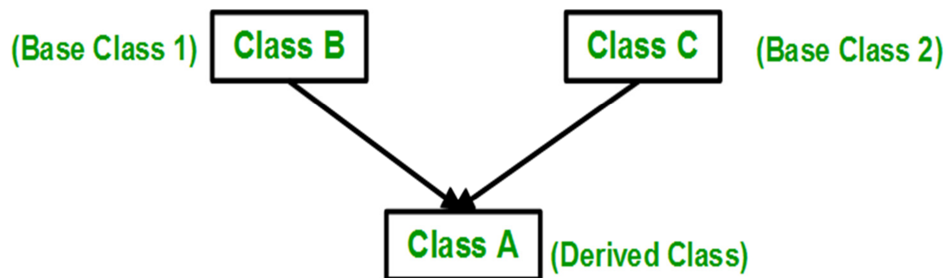
```
class subclass_name : access_mode base_class
{
  //body of subclass
};
```

```
// C++ program to explain Single inheritance
#include <iostream.h>

class Vehicle                    // base class
{
  public:
```

```
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

class Car: public Vehicle           // sub class derived from two base classes
{

};

int main()    // main function
{
    // creating object of sub class will invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Multiple Inheritance

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



Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  //body of subclass
};
```

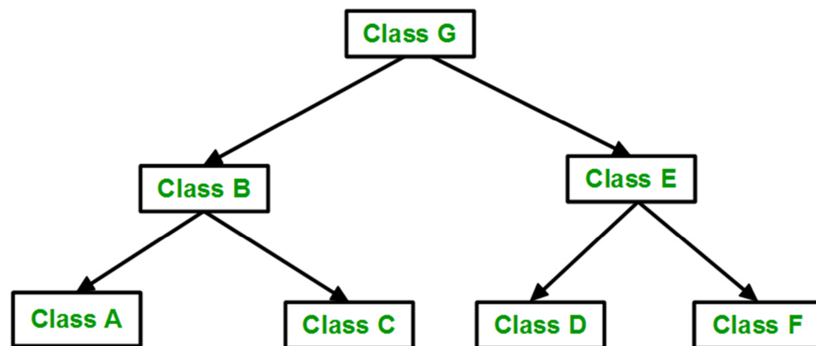
Here, the number of base classes will be separated by a comma (', ') and access mode for every base class must be specified.

```
// C++ program to explain multiple inheritance
#include <iostream.h>
class Vehicle                    // first base class
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

class FourWheeler                // second base class
```

```
{
  public:
    FourWheeler()
    {
      cout << "This is a 4 wheeler Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler
{

};

int main()
{
    // creating object of sub class will invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Hierarchical Inheritance

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

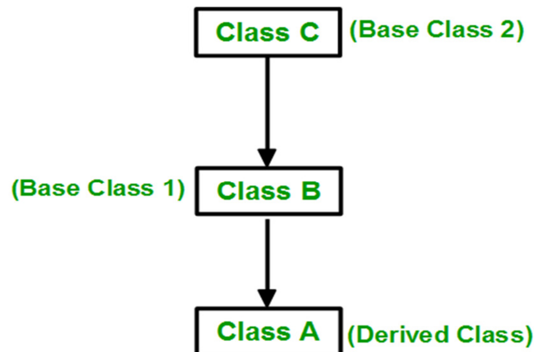



```
#include <iostream.h>
class Vehicle                    // base class
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

class Car: public Vehicle            // first sub class
{

};

class Bus: public Vehicle            // second sub class
{

};

int main()
{
```

```
    // creating object of sub class will invoke the constructor of base class
    Car obj1;
    Bus obj2;
    return 0;
}
```

Multilevel Inheritance

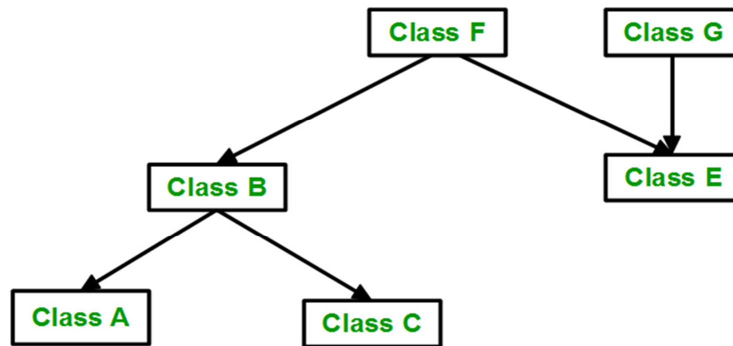
In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.




```
#include <iostream.h>
class Vehicle                    // base class
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
class fourWheeler: public Vehicle
{
  public:
    fourWheeler()
    {
      cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};

class Car: public fourWheeler             // sub class derived from two base classes
{
   public:
     car()
     {
       cout<<"Car has 4 Wheels"<<endl;
     }
};

int main()
{
    //creating object of sub class will invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Hybrid (Virtual) Inheritance

Hybrid Inheritance is combination of Hierarchical and Multilevel Inheritance.




```
#include <iostream.h>
class Vehicle                    // base class
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

class Fare              //base class
{
    public:
    Fare()
    {
        cout<<"Fare of Vehicle\n";
    }
};

class Car: public Vehicle           // first sub class
{

};

class Bus: public Vehicle, public Fare           // second sub class
{

};

int main()
{
    // creating object of sub class will invoke the constructor of base class
    Bus obj2;
    return 0;
}
```

In some situations, all the three kinds of inheritance namely, multilevel, multiple and hierarchical inheritance are involve. This is illustrated in following fig.

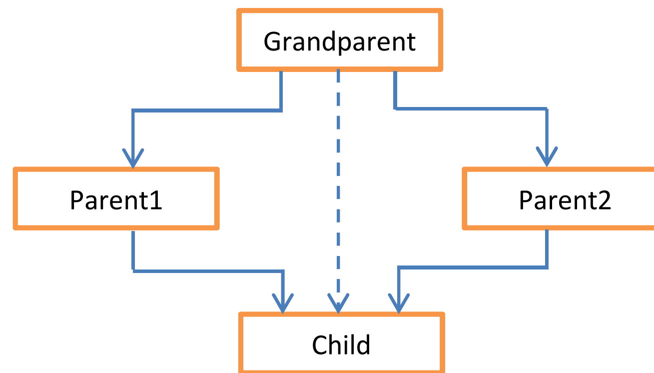
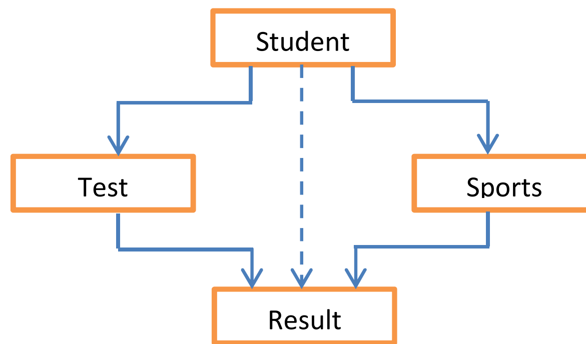


Fig: Multipath Inheritance

- In above fig. The 'child' has two direct base classes 'parent l' and 'parent 2' which themselves have a common base class 'grandparent'. The child inherits the properties of 'grandparent' via two separate paths. It can also inherit directly as shown by the broken line.
- All the public and protected members of 'grandparent' are inherited into 'child' twice first via 'parent l' and again via 'parent 2'. This means, 'child' would have duplicate sets of the members inherited from 'grandparent'. This introduces ambiguity and should be avoided.
- This duplication can be avoided by making the common base class (ancestor class)as virtual base class while declaring the direct or intermediate base classes as shown :

```
Class A                                  // Grandparent
{

};

Class B1 : virtual public A         // Parent 1
{

 };

Class B2: public virtual A           // Parent 2
{

};

 Class C : public B1, public B2     // Child
 {
          //Only one copy of A will be inherited
 };
```

- When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.
- The keywords virtual and public may be used in either order.
- For Example:




```
Program: //Virtual Base Classes
#include<iostream.h>
#include<conio.h>
class student
{
        protected :
                int rollno;
        public:
          void get_rollno();
           void put_rollno();
};
void student :: get_rollno()
{
        cout<<"\nPlease Enter rollno:->\t";
        cin>>rollno;
}
void student :: put_rollno()
{
        cout<<"\nRoll No:->\t"<<rollno;
}
 class test :virtual public student
{
        protected:
                float marks1,marks2;
        public:
                void getmarks();
                void putmarks();
};
void test :: getmarks()
{
        cout<<"\nEnter Marks of Test1:->\t";
        cin>>marks1;
        cout<<"\nEnter Marks of Test2:->\t";
        cin>>marks2;
}
void test :: putmarks()
{
        cout<<"\nMark obtained in Test1:->\t"<<marks1;
        cout<<"\nMarks obtained in Test2:->\t"<<marks2;
}
```

```
class sports : public virtual student
{
	protected:
		float score;
	public:
		void get_score()
		{
			cout<<"\nEnter Sports Score:->";
			cin>>score;
		}
		void put_score()
		{
			cout<<"\nEnter Sports Score:->"<<score;
		}
};

class result : public test,public sports
{
		float total;
	public:
		void display();
};
void result :: display()
{
	put_rollno();
	putmarks();
	put_score();
	total=marks1+marks2+score;
	cout<<"\nTotal Marks=>\t"<<total;
}

void main()
{
	clrscr();
	result r1;
	r1.get_rollno();
	r1.getmarks();
	r1.get_score();
	r1.display();
	getch();
}
```

Abstract classes

An abstract class is a one that is not used to create objects. An abstract class is designed only to act as base class (to be inherited by other classes). It is designed concept in program development and provide base upon which other classes may be built. In the following example, **refer program for virtual base classes.**

The student class is an abstract class since it was not used to create any objects.

Constructors in Derived Classes

When any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors.

When both the base and derived class contains constructors, the base constructor is executed first and then the constructor in derived class is executed.

The general form of defining derived constructor is:

derived_constructor(arglist1,arglist2,...,arglistN):base1(arglist1),base2 (arglist2),...,baseN(arglistN)
{
//body of derived constructor
}

Example 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.

```
#include<iostream.h>
#include<conio.h>
class Base
{
  int x;
  public:
  Base(int i)
  {
    x = i;
    cout <<"\nBase Parameterized Constructor\n";
    cout<<"x= "<<x;
  }
};

class Derived : public Base
{
  int y;
  public:
  Derived(int j,int k) : Base(j)
  {
    y = k;
    cout <<"\nDerived Parameterized Constructor\n";
    cout<<"y= "<<y;
  }
};

int main()
{
 clrscr();
```

```
 Derived d(10,20) ;
 getch();
 return 0;
}
```

Output:

Base Parameterized Constructor
x=10
Derived Parameterized Constructor
y=20

For multiple inheritance order of constructor call is, the base class's constructors are called in the order of inheritance and then the derived class's constructor.

```
// C++ program to show the order of constructor calls in Multiple Inheritance

#include <iostream.h>
class Parent1
{
   public:

   Parent1()
   {
      cout << "Inside first base class" << endl;
   }
};

class Parent2
{
   public:
   Parent2()
   {
      cout << "Inside second base class" << endl;
   }
};
class Child : public Parent1, public Parent2
{
   public:
       Child()
      {
        cout << "Inside child class" << endl;
      }
};

int main()
{
   Child obj1;
   return 0;
}
```

Output:

Inside first base class
Inside second base class
Inside child class