Inheritance

- It is the process of creating new classes, called derived classes, from existing classes called base classes.
- The derived class inherits all the properties from the base class and can add its own properties as well.
- Uses the concept of code reusability.
- Once a base class is written, we can reuse the properties of the base class in other classes by using the concept of inheritance.
- Reusing existing code saves time and money and increase program's reliability.
- A programmer can use a class created by another person, or company , and, without modifying it, derive other classes from it that are suited to the particular situations.

Advantages of Inheritance:

- ✓ Development model closer to real life world model with hierarchical relationships.
- ✓ Reusability
- ✓ Extensibility: while using concept of inheritance, we can extend new features as feature in existing features.
- ✓ Data hiding: Base class can decide to keep some data private to that it cannot be altered by the derived class.

Base Class and Derived Class

- Base Class / Super Class / Parent Class
- Derived Class/ Sub Class/ Child Class

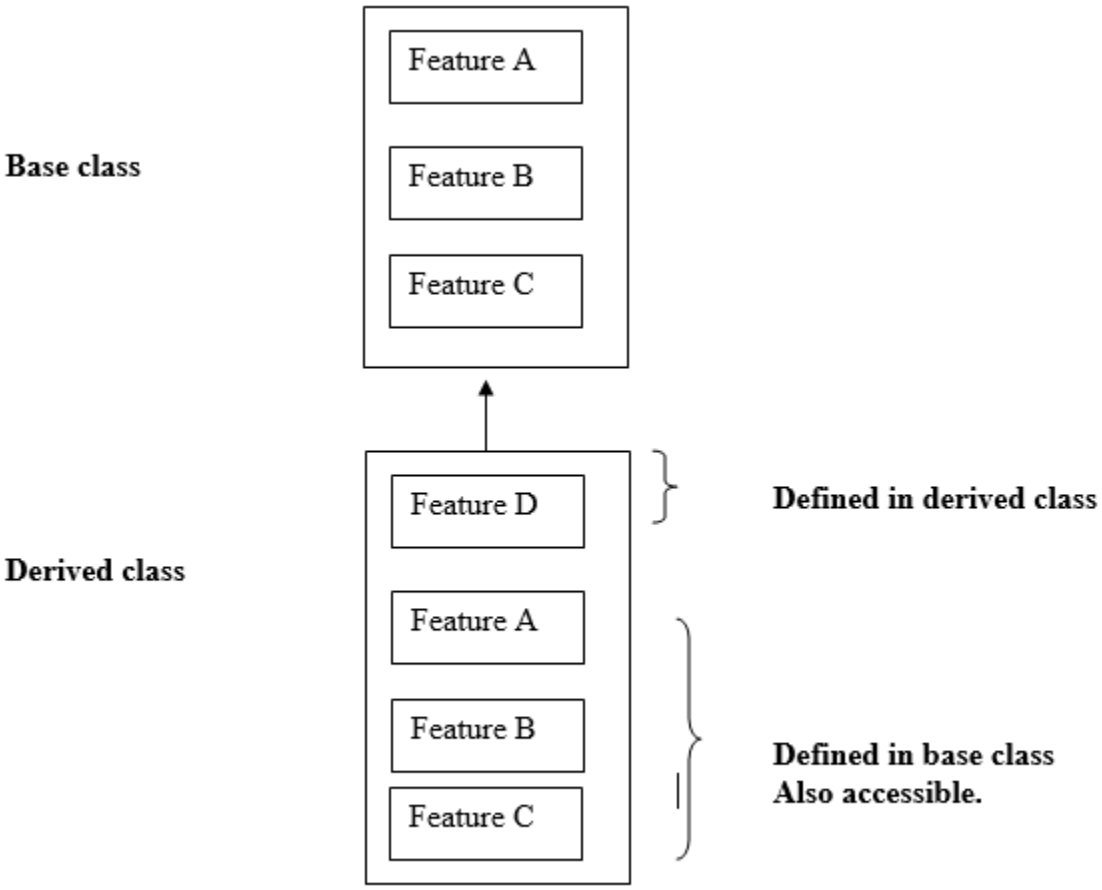
Defining Derived Class:

- Colon Symbol (:) is used.
- Syntax:

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

Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which you want to inherit this sub class (private/ public / protected) and **base_class_name** is the name of the base class from which you want to inherit the sub class. If access mode (access specifier or visibility mode) is not written default mode is private.

Note: A derived class doesn't inherit access to private data members. However, it does inherit a full parent object, which contains any private members which that class declares



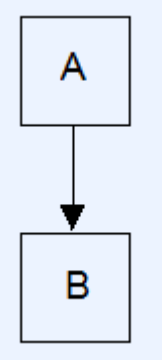

```
// C++ program to demonstrate implementation
// of Inheritance

#include <iostream>
using namespace std;

//Base class
class Parent
{
    public:
      int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
    public:
      int id_c;
};
```

```
//main function
int main()
   {

        Child obj1;

        // An object of class child has all data members
        // and member functions of class parent
        obj1.id_c = 7;
        obj1.id_p = 91;
        cout << "Child id is " <<  obj1.id_c << endl;
        cout << "Parent id is " <<  obj1.id_p << endl;

        return 0;
   }
```

Different forms of inheritance:

Single Inheritance

A derived class with only one base class is called single inheritance.



class A

{

…….

};

class B : public A

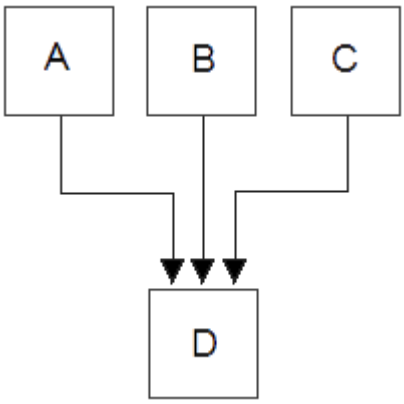
{

…….

};

Multiple Inheritance

A derived class with multiple base class is called multiple inheritance.



class A

{

…….

};

class B

{

…….

};

class C

{

…….

};

class D: public A, public B, public C

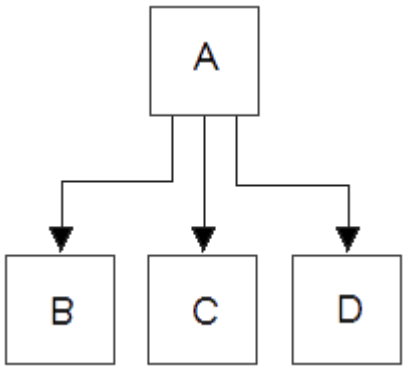
{

…….

};

Heirarchical Inheritance

In such type, two or more classes inherit the properties of one base class.



class A

{

…….

};

class B: public A

{

…….

};

class C : public A

{

…….

};

class D: public A

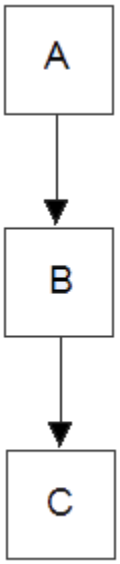
{

…….

};

Multilevel Inheritance

A derived class with one base class and that base class is a derived class of another is called multilevel inheritance. The process can be extended to an arbitrary number of levels.



class A

{

…….

};

class B: public A

{

…….

};

class C : public B

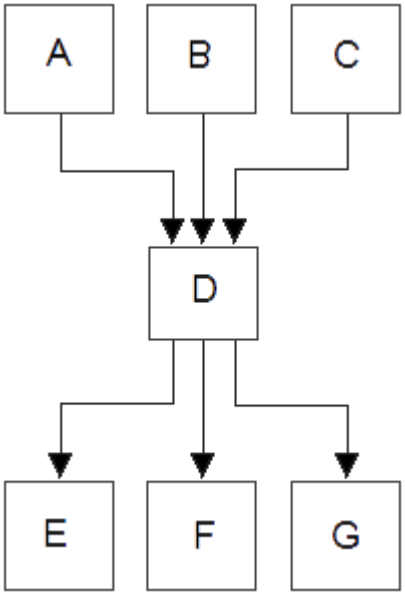
{

…….

};

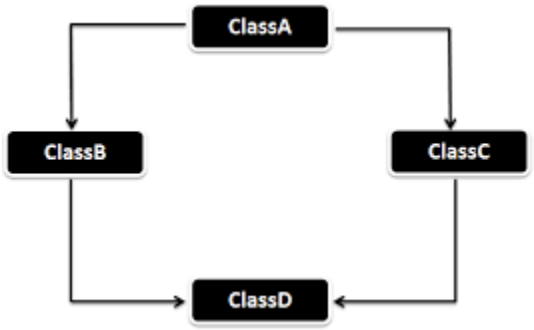
Hybrid Inheritance:

This inheritance is the combination of more than one type of inheritance mentioned above.



Multipath inheritance:

Derivation of a class from other derived classes, which are derived from same base class is called multi-path inheritance. Also a hybrid inheritance.



```
//Multiple Inheritance
#include<iostream>
using namespace std;
class Father {
      protected:
            int moneyf;
      public:
            void getMoney_Father(){
                  cout<<"Enter Pocket money received from
father:"<<endl;
                  cin>>moneyf;
            }
            void displayMoney_Father(){
                  cout<<"Pocket money given by
Father="<<moneyf<<endl;
            }


};
class Mother {
      protected:
            int moneym;
      public:
            void getMoney_Mother(){
                  cout<<"Enter Pocket money received from
mother:"<<endl;
                  cin>>moneym;
            }
            void displayMoney_Mother(){
                  cout<<"Pocket money given by
Father="<<moneym<<endl;
            }
```

```
};
class Child : public Father, public Mother{
      int moneyt;
      public:
            void total_pocket_money(){
                  moneyt = moneyf+ moneym;
                  cout<<"Total pocket money with
child="<<moneyt<<endl;
            }
};
int main(){
      Child obj;
      obj.getMoney_Father();
      obj.getMoney_Mother();
      obj.displayMoney_Father();
      obj.displayMoney_Mother();
      obj.total_pocket_money();
}
```

//hierarchical inheritance

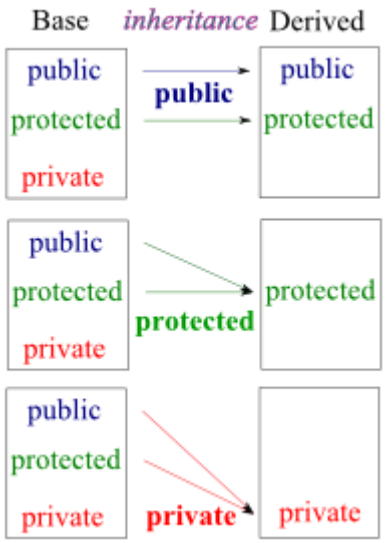
```
#include<iostream>
using namespace std;
class Common{
      protected:
            string name;
            int age;
      public:
            void getInfo(){
                  cout<<"Enter Detail:"<<endl;
                  cout<<"Name: "<<endl;
                  cin>>name;
                  cout<<"Age: "<<endl;
                  cin>>age;
            }
            void displayInfo(){
                  cout<<"Name: "<<name<<endl;
                  cout<<"Age: "<<age<<endl;
            }
};
class Teacher: public Common{
      int salary;
      public:
            getSal(){
                  cout<<"Enter Teacher's salary:"<<endl;
                  cin>>salary;
            }
            displaySal(){
                  cout<<"Salary of the teacher is : "<<salary<<endl; }
};
class Student: public Common{
```

```
      string faculty;
      int sem;
      public:
            getStudent(){
                  cout<<"Student's Faculty : "<<endl;
                  cin>>faculty;
                  cout<<"Student's Semester: "<<endl;
                  cin>>sem;
            }
            displayStudent(){
                  cout<<"Student is of "<<sem <<"th semester of
"<<faculty<<" Faculty"<<endl;
            }
};

int main(){
      Teacher t;
      Student s;
      cout<<"Teacher type object t"<<endl;
      t.getInfo();
      t.getSal();
      t.displayInfo();
      t.displaySal();
      cout<<"Student type object s"<<endl;
      s.getInfo();
      s.getStudent();
      s.displayInfo();
      s.displayStudent();
      }
```

Recalling Access Specifiers:

Class Member Access Specifiers	Accessible from own class	Accessible from derived class	Accessible from object
Private Member	Yes	No	No
Protected Member	Yes	Yes	No
Public Member	Yes	Yes	Yes

Public Protected and Private Inheritance



Syntax for deriving a class is :

class derived_class_name : visibility_mode base_class_name

{

//member of derived class

};

Here visibility mode (also known as access specifier) can be either private, protected or public. Default mode is private.

Private inheritance:

- When a base class is derived in the subclass with private access specifier, it cannot access the private members of the base class.
- A public and protected member of the base class becomes private member of the derived class by using private visibility mode.
- The member function of the child class can access the public and protected members of the base class.

Protected inheritance:

When the member of the base class is derived with protected access mode, then the protected and public members of the base class becomes protected members of the base class. These members can be inherited further by other subclasses.

Public inheritance:

When the member of the base class is derived with public access mode, then the public members of the base class becomes public member of the derived class and protected members become protected members of the derived class, but a private member is not accessible.

Function overriding

- Function overriding is a feature that allows us to have a same function in child class which is already present in the parent class. A child class inherits the data members and member functions of parent class, but when you want to override a functionality in the child class then you can use function overriding. It is like creating a new version of an old function, in the child class.
- To override a function you must have the same signature in child class. It means the data type and sequence of parameters and number of parameter.
- In function overriding, the function in parent class is called the overridden function and function in child class is called overriding function.
- If we want to call the overridden function by using the object of child class, we have to create the child class object in such a way that the reference of parent class points to it.
  Obj.class_name::function_name();

```
//function overriding
#include<iostream>
using namespace std;
class Father{
      public:
            void property(){
                  cout<<"All property will be given to my
child!"<<endl;
            }
            void marriage(){//overridden function
                  cout<<"Child should do arrange marriage !"<<endl;
            }
};
class Child: public Father{
      public:
            void marriage(){//overriding function
                  cout<<" Child should do love marriage !"<<endl;
            }
};
int main(){
      Child c;
      c.property();
      c.marriage();//modified function is called
      c.Father::marriage();//overridden function is called
}
```

Ambiguity in multiple inheritance:

Dictionary meaning of ambiguity:

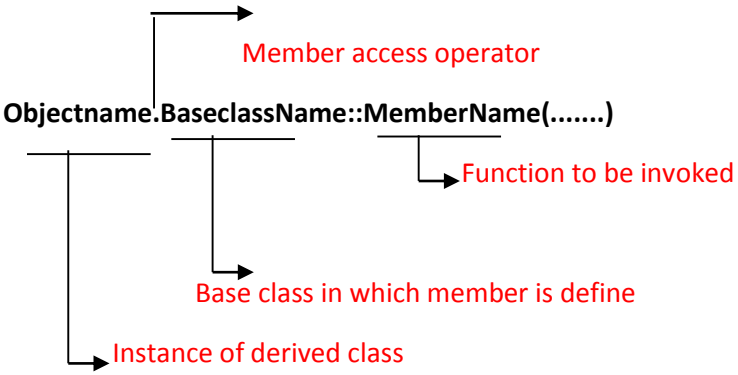
The fact of something having more than one possible meaning and therefore possibly causing confusion.

Ambiguity is a problem that surfaces in certain situation involving multiple inheritances.

- Base classes having function with the same name.
- The class derived from these base classes is not having a function with the name as those of its base classes.
- Member of a derived class or its objects referring to a member, whose name is the same as those in base classes

Suppose, two base classes have a same function which is not overridden in derived class. If we try to call the function using the object of the derived class, compiler shows error. It's because compiler doesn't know which function to call. This situation is the situation of ambiguity.

- **The problem of ambiguity is resolved using the scope resolution operator as shown in following syntax.**



Example: of Ambiguity

```
//amboguty in member access
#include<iostream>
Using namespace std;

class  A
{
      public:
            void show(){
            cout<<" class  A ";
            }
};
class  B
{
      public:
            void show() {
            cout<<" class  B ";
            }
};
class C : public A , public B
{

};
int main()

{
      C objC;     // object of class C
      //objC.show() ;   //  ambiguous....error
      objC.A::show() ; // ok invokes show() in class A.
      objC.B::show() ;  // ok
}
```

The statement objC.show(); is ambiguous as compiler has to choose A::show( ) or B:: show( ). It can be resolved using the scope resolution operator as follows

objc.A::show( );

which refers to the version of show ( ) in the class A.

Aggregation (Class within Class)

- ✓ Also known as containership
- ✓ When a class contains object of another class as its data member, it is termed as containership.
- ✓ In C++, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents **HAS-A** relationship.
- ✓ 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.

- ✓ Containership also referred to as composition allows a class to contain an object of a different class as a member data. Ex: Class A could contain an object of class B as a member. Here, all the public methods (or functions) defined in B can be executed within the class A. Class A becomes the container, while class B becomes the contained class. This can also be explained as class A is composed of class B.

Syntax:

```
class A{
public:
        A ()
        {
            // this is a constructor of class A.
        }
};
class B
{
    /* An object of class A is created inside class B.
     This is called Containership. */
    A object_A;
    public:
        B ()
        {
            // this is a constructor of class B.
        }
};
```

Inheritance vs containership

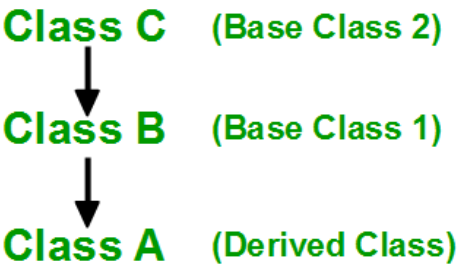
Inheritance is the ability for a class to inherit properties and behavior from a parent class by extending it, while Containership is the ability of a class to contain objects of different classes as member data. If a class is extended, it inherits all the public and protected properties/behavior and those behaviors may be overridden by the subclass. But if a class is contained in another, the container does not get the ability to change or add behavior to the contained. Inheritance represents an “is-a” relationship in OOP, but Containership represents a “has-a” relationship.

Order of Constructor/ Destructor Call in Inheritance in C++

- Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.
- If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoke, i.e the order of invocation is that **the base class's default constructor will be invoked first and then the derived class's default constructor will be invoked**.
- To call the parameterized constructor of base class when derived class's parameterized constructor is called, you have to explicitly specify the base class's parameterized constructor in derived class.

- **Whenever the derived class's default constructor is called, the base class's default constructor is called automatically.**
- **To call the parameterized constructor of base class inside the parameterized constructor of sub class, we have to mention it explicitly.**
- **The parameterized constructor of base class cannot be called in default constructor of sub class, it should be called in the parameterized constructor of sub class.**
- **Destructors in C++ are called in the opposite order of that of Constructors.**

Order of Inheritance



Order of Constructor Call

1. C() (Class C's Constructor)
2. B() (Class B's Constructor)
3. A() (Class A's Constructor)

Order of Destructor Call

1. ~A() (Class A's Destructor)
2. ~B() (Class B's Destructor)
3. ~C() (Class C's Destructor)

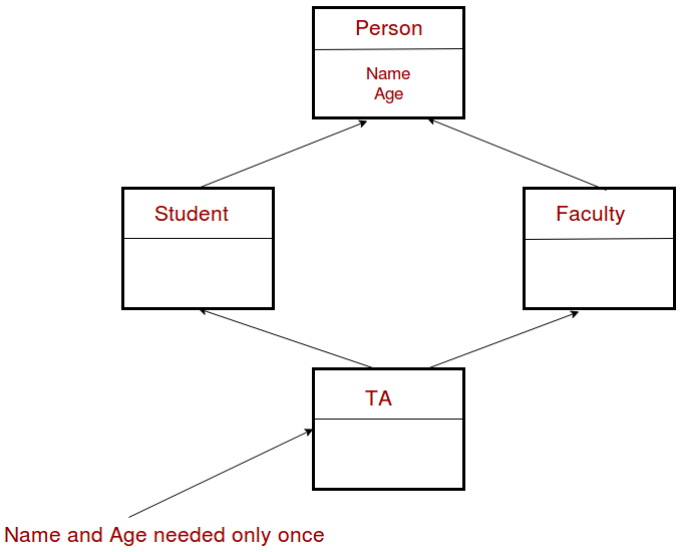
```
// C++ program to show how to call parameterized Constructor
// of base class when derived class's Constructor is called
```

```cpp
#include<iostream>
using namespace std;
class Parent{
    int a; int b;
    public:
        Parent(int x, int y){
        a= x;
        b=y;
        cout<<"From Parent Class Constructor:"<<endl;
        cout<<"a="<<a<<endl;
        cout<<"b="<<b<<endl;
        }
};
class Child: public Parent{
    int d;
    public:
        Child(int p, int q, int r): Parent(p,q){
            d=r;
            cout<<"From Base Class Constructor:"<<endl;
            cout<<"d="<<d;
        }
};
int main(){
    Child obj(5,6,7);
}
```

```
//program to show order of constructors and destructors call
in inheritance
#include<iostream>
using namespace std;
class A{
     public:
          A(){
               cout<<" Hi! from class A Constructor.."<<endl;
          }
          ~A(){
               cout<<"Bye! from Class A Destructor.."<<endl;
          }
};
class B: public A{
     public:
          B(){
               cout<<" Hi! from class B Constructor.."<<endl;
          }
          ~B(){
               cout<<"Bye! from Class B Destructor.."<<endl;
          }
};
class C: public B{
     public:
          C(){
               cout<<" Hi! from class C Constructor.."<<endl;
          }
          ~C(){
               cout<<"Bye! from Class C Destructor.."<<endl;
          }
};
int main(){
     C obj;
}
```

Diamond Problem in inheritance:

- The diamond problem occurs when two super classes of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.



- **Solution to Diamond problem → Virtual base class**

Virtual Base Class

The principle behind the virtual base class is very simple. When the same class is inherited more than once via multiple paths, multiple copies of the base class members are created in memory. By declaring the base class inheritance as virtual, only one copy of the base class is inherited. A base class inheritance can be specified as a virtual using the virtual qualifier.

```
class A
{
…..
…..
};
class B1 : virtual public A
{
….
….
};
class B2 : public virtual A
{
……
…...
};
class C : public B1, public B2
{
……
……
};
```

- Keywords 'virtual' and 'public' can be used in either way.

- When a class is made virtual base class, only one copy of that is inherited, regardless of how many number of inheritance path exists between the virtual base class and derived class. Since there is only one copy, there is no ambiguity.

- Mechanism is also termed as **virtual inheritance**.

❖ 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 class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.

❖ When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class' name with the word virtual.

Example

```
#include<iostream>
using namespace std;
class A
{
   public:
        int i;
};
class B : virtual public A
{
   public:
        int j;
};
class C: public virtual  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<<endl;
   cout <<"Value of j is :"<< ob.j<<endl;
   cout <<"Value of k is :"<< ob.k<<endl;
   cout << "Sum is :"<< ob.sum <<endl;
   return 0;
}
```