

INHERITANCE: EXTENDING CLASSES

Helps in reusability -> saves time, money and increases reliability

A new class uses the properties of existing class

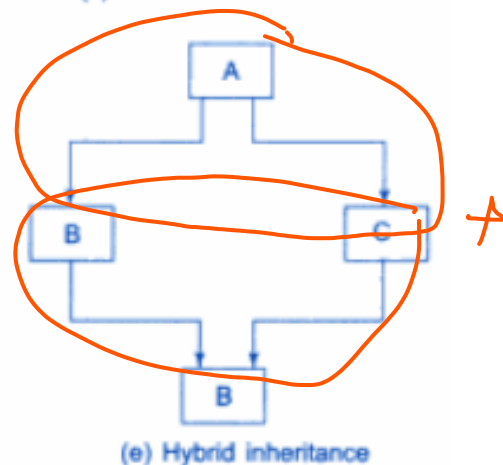
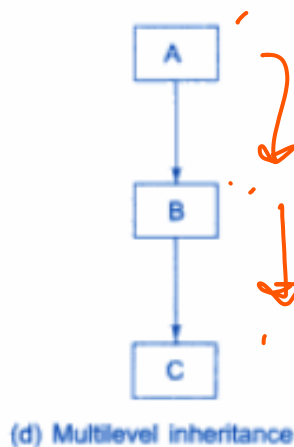
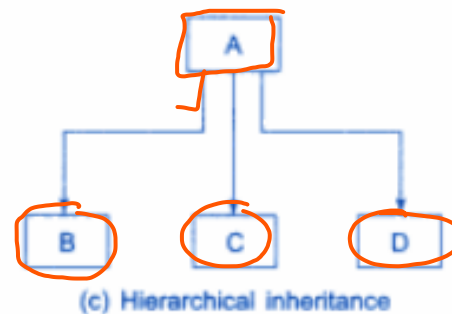
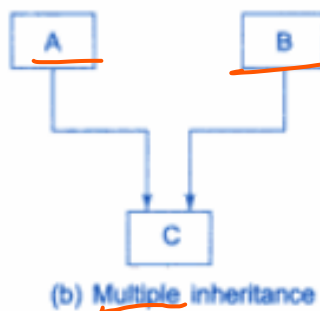
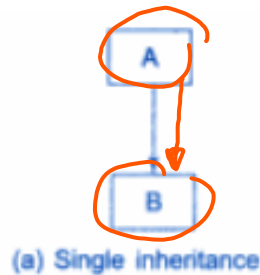
Deriving a new class from old one is **Inheritance**

Old class is known as **Base class** or **Parent class**.

New class is known as **Derived class** or **Child class**.

A derived class inherits some or all of the traits from base class

TYPES OF INHERITANCE



HOW TO DEFINE DERIVED CLASS?

```
class derived-class-name : visibility-mode base-class-name
{
    .....//
    .....//  members of derived class
    .....//
};
```

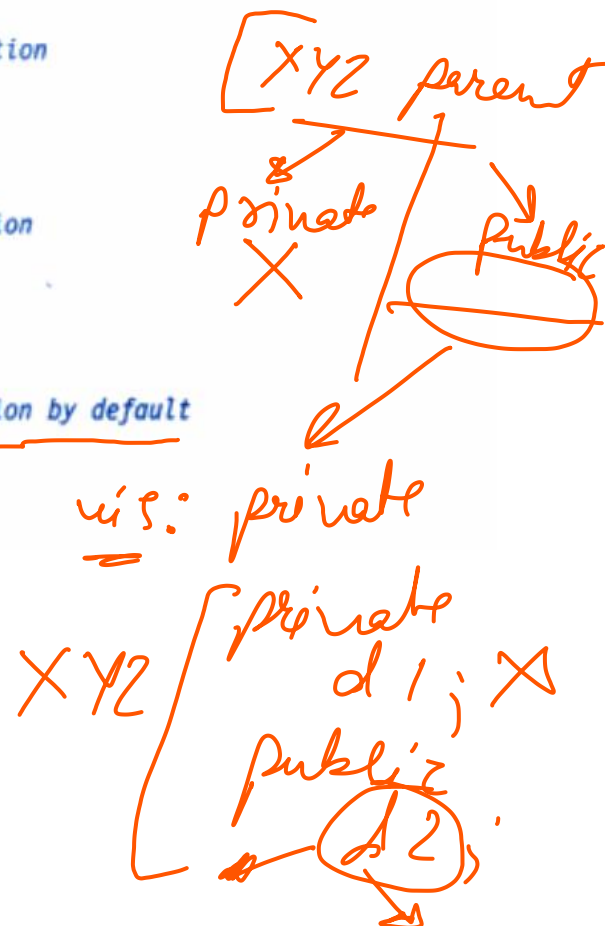
The colon indicates that the *derived-class-name* is derived from the *base-class-name*. The *visibility-mode* is optional and, if present, may be either **private** or **public**. The default visibility-mode is **private**. Visibility mode specifies whether the features of the base class are *privately derived* or *publicly derived*.

Examples:

```
class ABC: private XYZ    // private derivation
{
    members of ABC ✓
};
```

```
class ABC: public XYZ    // public derivation
{
    members of ABC ✓
};
```

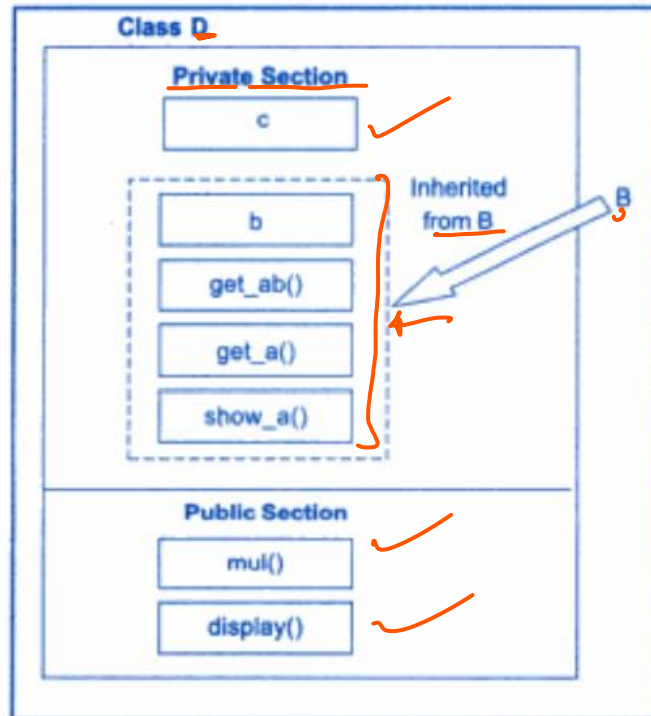
```
class ABC: XYZ    // private derivation by default
{
    members of ABC ✓
};
```



IF USING **PRIVATE** VISIBILITY MODE

Private members -> Not inherited

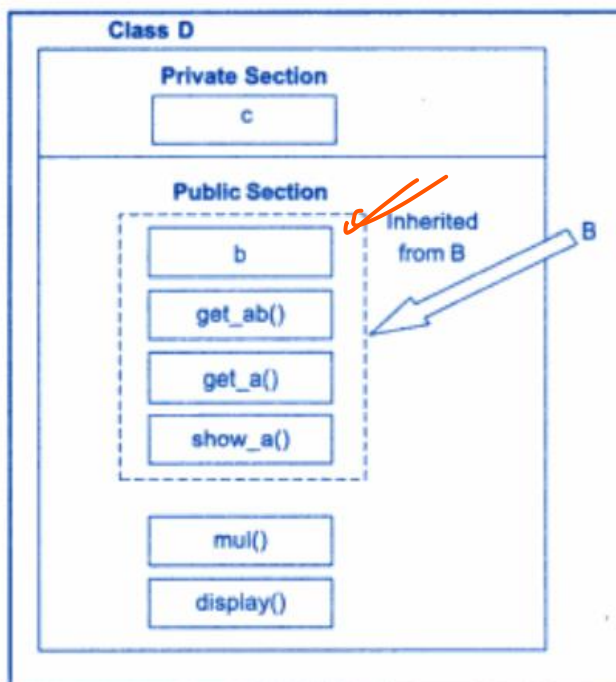
Public members -> Are inherited as Private



IF USING **PUBLIC** VISIBILITY MODE

Private members -> Not inherited

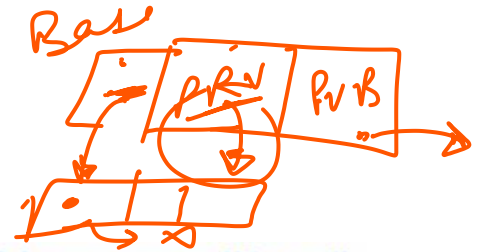
Public members -> Are inherited as Public



A PRIVATE MEMBER IS NEVER INHERITED BY A DERIVED CLASS

IMPORTANT

Suppose a base class and a derived class define a function of the same name. What will happen when a derived class object invokes the function? In such cases, the derived class function supersedes the base class definition. The base class function will be called only if the derived class does not redefine the function.

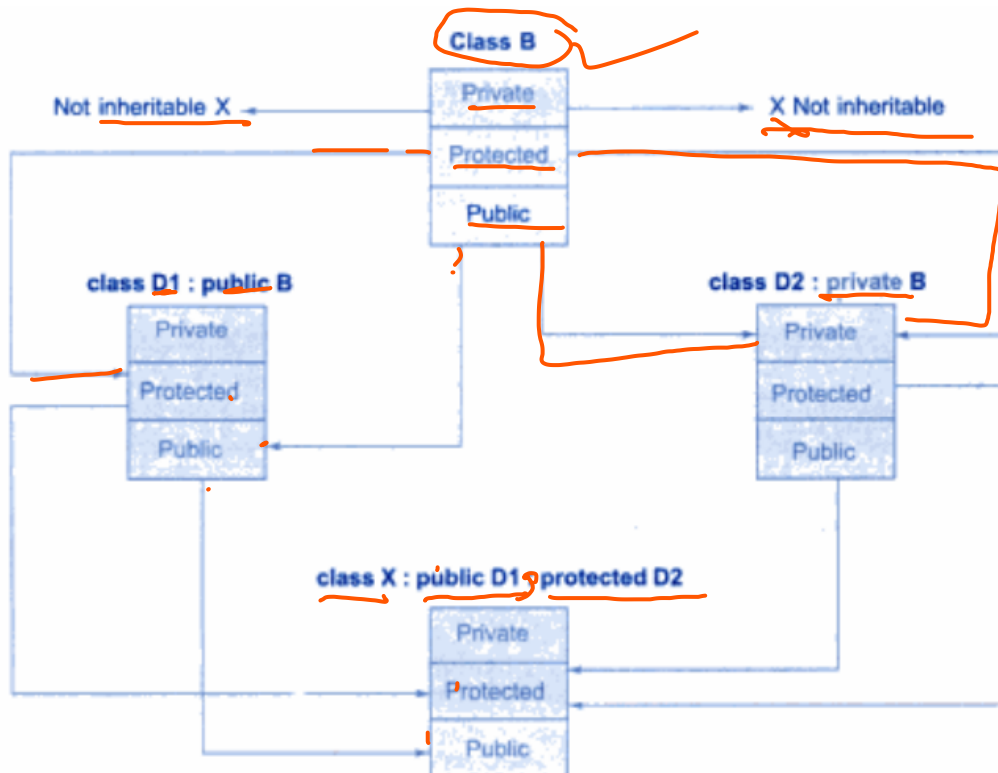
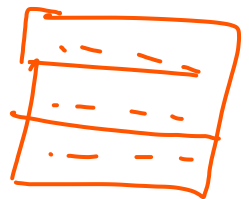
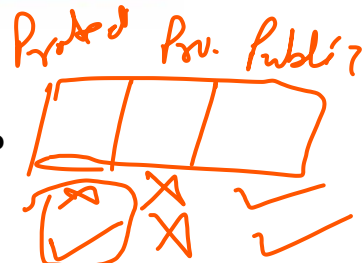


WHAT IF WE NEED TO INHERIT PRIVATE MEMBER OF BASE CLASS?

Using **PROTECTED** visibility mode. It works same as **PRIVATE** mode in a class.

But when we inherit in

- Private mode: Private members are **never inherited** ✗
 Protected members become **private** in derived class
 Public members become **private** in derived class
- Protected mode: Private members are **never inherited**
 Protected members become **protected** in derived class
 Public members become **protected** in derived class
- Public mode: Private members are **never inherited**
 Protected members become **protected** in derived class
 Public members become **public** in derived class



The keywords **PRIVATE**, **PROTECTED** and **PUBLIC** may appear in any order any number of times in declaration of a class. For example

```
class beta
{
    protected: ✓
        .....
    public: ✓
        .....
    private: ✓
        .....
    public: ✓
        .....
};
```

BUT THIS IS NOT THE BEST PRACTICE:

Must use this ->

```
class beta
{
    ..... } // private by default
    ..... }
    protected: }
    ..... }
    public: }
    ..... }
}
```

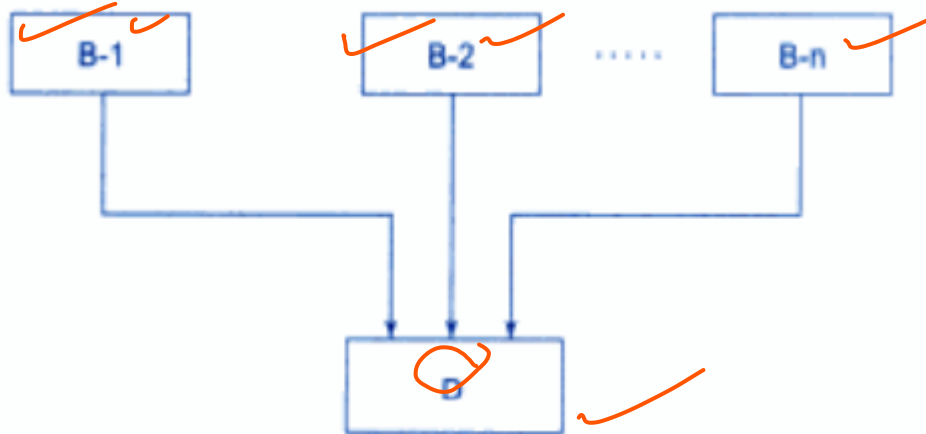
SUMMARY TILL NOW:

Table 8.1 *Visibility of inherited members*

<i>Base class visibility</i>	<i>Derived class visibility</i>		
	<i>Public derivation</i>	<i>Private derivation</i>	<i>Protected derivation</i>
<u>Private</u> →	Not inherited	Not inherited	Not inherited
Protected →	<u>Protected</u>	<u>Private</u>	<u>Protected</u>
Public →	Public	Private	Protected

MULTIPLE INHERITANCE

A class inheriting attributes of multiple classes



The syntax of a derived class with multiple base classes is as follows:

```
class D: visibility B-1, visibility B-2 ...  
{  
    .....  
    .....(Body of D)  
    .....  
};
```

```
class P : public M, public N  
{  
    public:  
        void display(void);  
};
```

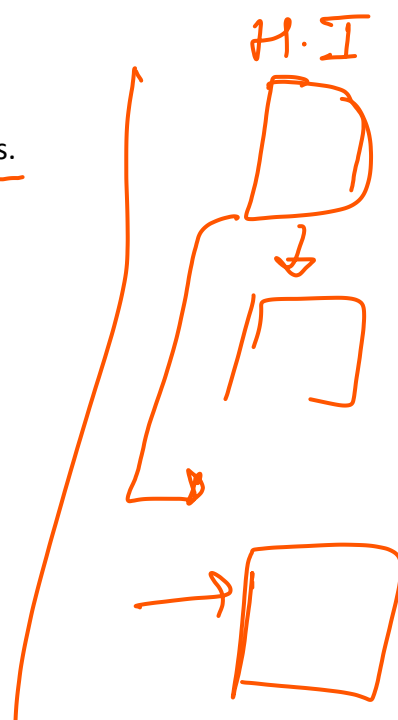
AMBIGUITY RESOLUTION

Function with same name appears in more than one base class.

```
class M ✓
{
    public:
        void display(void)
        {
            cout << "Class M\n";
        }
};

class N ✓
{
    public:
        void display(void)
        {
            cout << "Class N\n";
        }
};

class P : public M, public N
{
    public:
        void display(void) // overrides display() of M and N
        {
            M :: display();
            N :: display();
        }
};
```

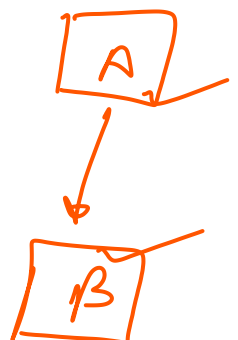


EVEN IN SINGLE INHERITANCE, FUNCTION WITH SAME NAME APPEARS BUT CHILD OVERRIDES IT

WE CAN USE THIS

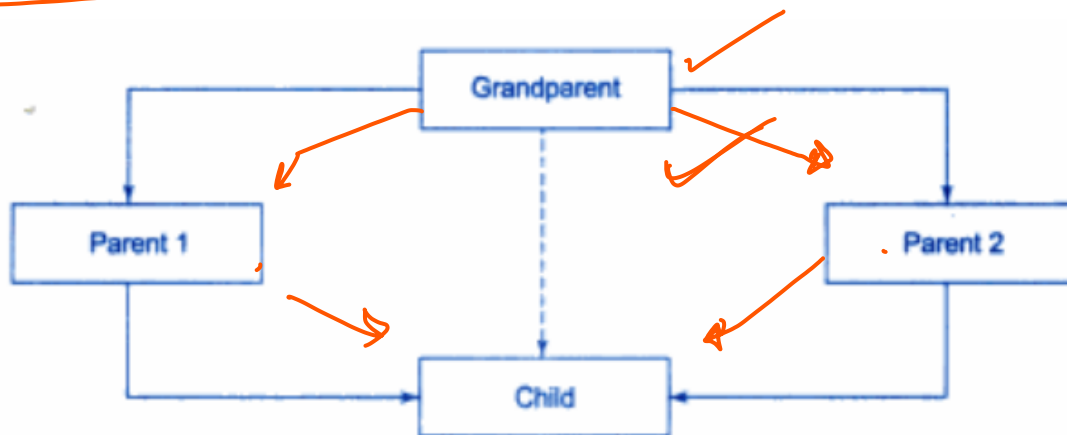
```
B b; ✓
b.display(); ✓
b.A::display();
b.B::display();
```

// derived class object
// invokes display() in B
// invokes display() in A
// invokes display() in B



VIRTUAL BASE CLASS

Just now we saw some cases of ambiguity, let's see some more if we have a hybrid inheritance like this;



Here Child class is getting duplicate set of attributes of Grandparent class. Hence, it will cause ambiguity, which we can avoid by using,

Making common base class (Grandparent) as virtual base class while declaring intermediate class (parent classes).

```
class A                                // grandparent
{
    ....
};

class B1 : virtual public A             // parent1
{
    ....
};

class B2 : public virtual A            // parent2
{
    ....
};

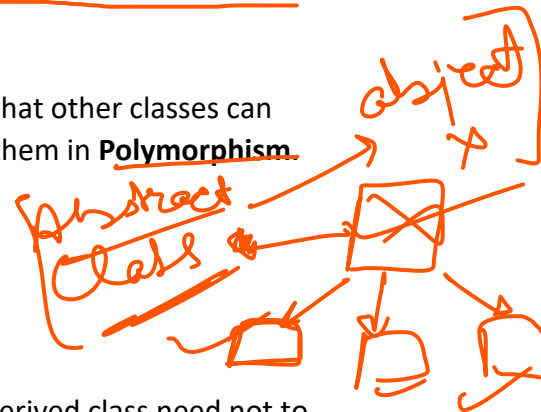
class C : public B1, public B2          // child
{
    ....
    // only one copy of A
    // will be inherited
};
```

Compiler
Smart

C++ will take care of making only one copy, regardless of inheritance path followed now.

ABSTRACT CLASS

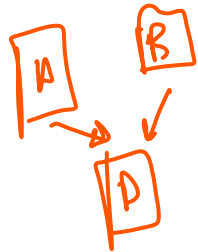
A class, not used to create object. It provides attribute so that other classes can inherit and further modify them for use. We will see more about them in Polymorphism.



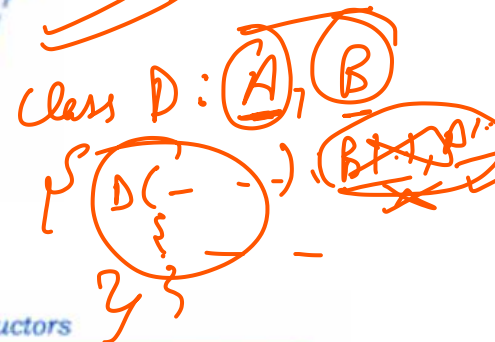
CONSTRUCTOR

Constructor is really important in initialization of objects.

- ➔ Until no base class constructor takes any arguments, the derived class need not to have constructor function
- ➔ Else it is mandatory
- ➔ Derived class constructor will pass arguments to base class first, hence base class constructor is first called then derived class constructor is called.
- ➔ In case of multiple inheritance, base class are constructed in the order in which they appear in declaration of derived class.



```
D(int a1, int a2, float b1, float b2, int d1):
A(a1, a2), /* call to constructor A */
B(b1, b2) /* call to constructor B */
{
    d = d1; // executes its own body
}
```



ALSO IMPORTANT:

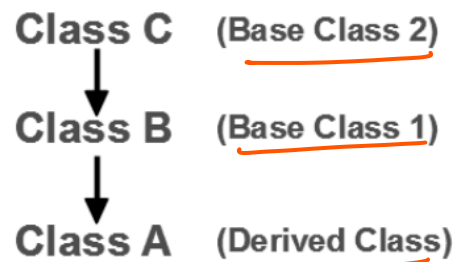
Table 8.2 Execution of base class constructors

Method of inheritance	Order of execution
1 <u>Class B: public A</u>	<u>A() ; base constructor</u>
{	<u>B() ; derived constructor</u>
};	
2 <u>class A: public B, public C</u>	<u>B() ; base(first)</u> ✓
{	<u>C() ; base(second)</u> ✓
};	<u>A() ; derived</u> ✓
3 <u>class A: public B, virtual public C</u>	<u>C() ; virtual base</u> ✓
{	<u>B() ; ordinary base</u> ✓
};	<u>A() ; derived</u> ✓



IMPORTANT

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)

DIFFERENT METHOD OF INITIALISING THE DATA MEMBER OF CLASS

```
class XYZ
{
    int a;
    int b;
public:
    XYZ(int i, int j) : a(i), b(2 * j) {}
};

main()
{
    XYZ x(2, 3);
}
```

arg list

initialization list

a → 2
b → 6

a ✓
b ✓
XYZ(int i, int j) : b(i), a(i + j) { }

In this case, **a** will be initialized to 5 and **b** to 2. Remember, the data members are initialized in the order of declaration, independent of the order in the initialization list. This enables us to have statements such as

XYZ(int i, int j) : a(i), b(a * j) { }

Here **a** is initialized to 2 and **b** to 6. Remember, **a** which has been declared first is initialized first and then its value is used to initialize **b**. However, the following will not work:

XYZ(int i, int j) : b(i), a(b * j) { }

because the value of **b** is not available to **a** which is to be initialized first.

The following statements are also valid:

{
XYZ(int i, int j) : a(i) { b = j; } ✓
XYZ(int i, int j) { a = i; b = j; } ✓
}

CONTAINERSHIP OR NESTING

Declaring objects of another class as data-member in a class.

```
class alpha {....};  
class beta {....};  
class gamma  
{  
    alpha a;  
    beta b;  
    ....  
};
```

// a is an object of alpha class
// b is an object of beta class

First, we need to create member object by using constructor hence, we use

```
class gamma
{
    .....
    alpha a;          // a is object of alpha
    beta b;           // b is object of beta
public:
    gamma(arglist): a(arglist1), b(arglist2)
    {
        // constructor body
    }
};
```