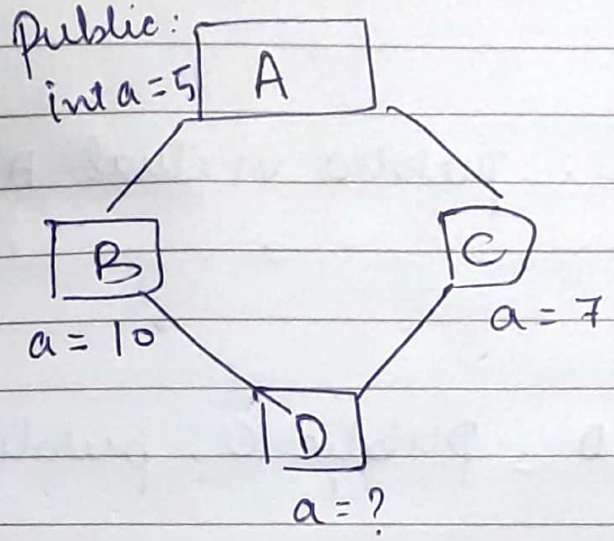
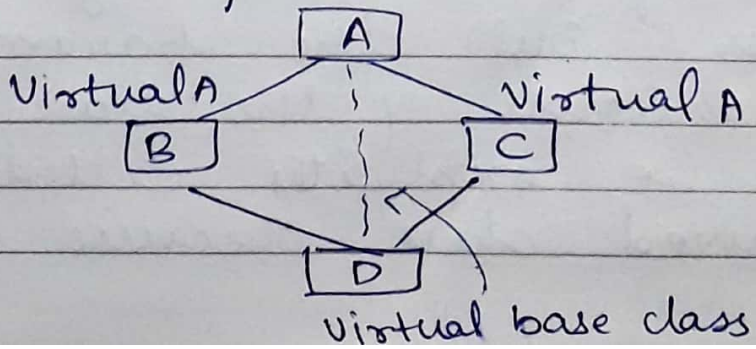
✡ Ambiguity in Hybrid inheritance



In the above diagram, D is inheriting the properties of B & C & indirectly it is inheriting the properties of A. which means that the properties of A are being inherited from two different paths leading to ambiguity To resolve this problem C++ provides the concept of the virtual base class & the syntax of the virtual base class is as follows



```
class A
{    data member;
     data fun";
};

class B: public virtual A
{
}
class C: public virtual A
{
}
class D : public C, public B
{
}
```

✡ Order of evaluation of constructor & destructors in inheritance using parameterised & non parameterised constructor

→ In inheritance all public & protected members of the base class are directly inherited by the derived class. But the parameterised constructors of the base class have to be explicitly called in the derived class because the constructor

is used for creating the object & the derived class object will call only its own constructor. Therefore explicit call is mandatory. For example

```
class Base
{
    public:
        Base ( )
        {
            cout << " Base " ;
        }
};

class derived : public Base
{
        int x;
    public:
        Derived ( )
        {   cout << " Derived " ;
        }
        Derived (int i)
        { x = i;
          cout << " para derived " ;
        }
};
```

```
main ()
{
    Base b;
    Derived d, d1(5);
}
```

O/P

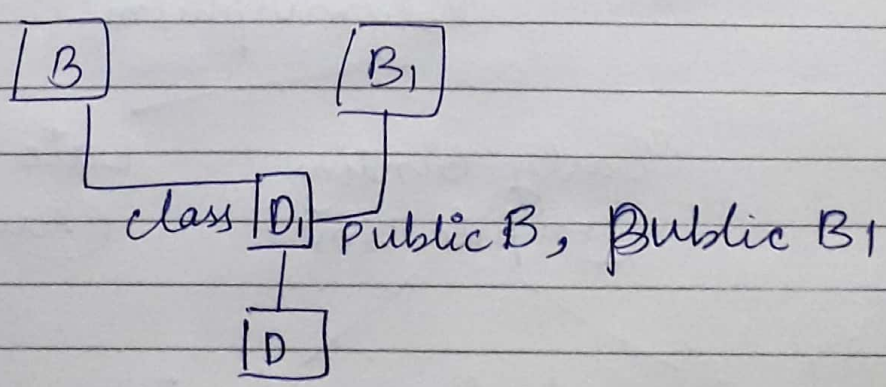
```
Base
Base derived
Base para derived
```

⇒

```
class Base
{
    protected:
        int y;

    public:
        Base(int a)
        {
            y=a;
            cout << "Base";
        }

};

class Derived : public Base
{
    int x;
```

```
public :                                  → Base (belongs)
Derived (int a, int b) : Base(a)          ← new way of assigning y=a
                                            //explicit call of base constructor
{
  //y = a
  x = b
  cout << " Derived";
}
};
main ()
{
  Base b(2);
  derived D(2, 5);
}
```

The constructor, are executed in the order of base to derive & the destructor are executed from derived to base.

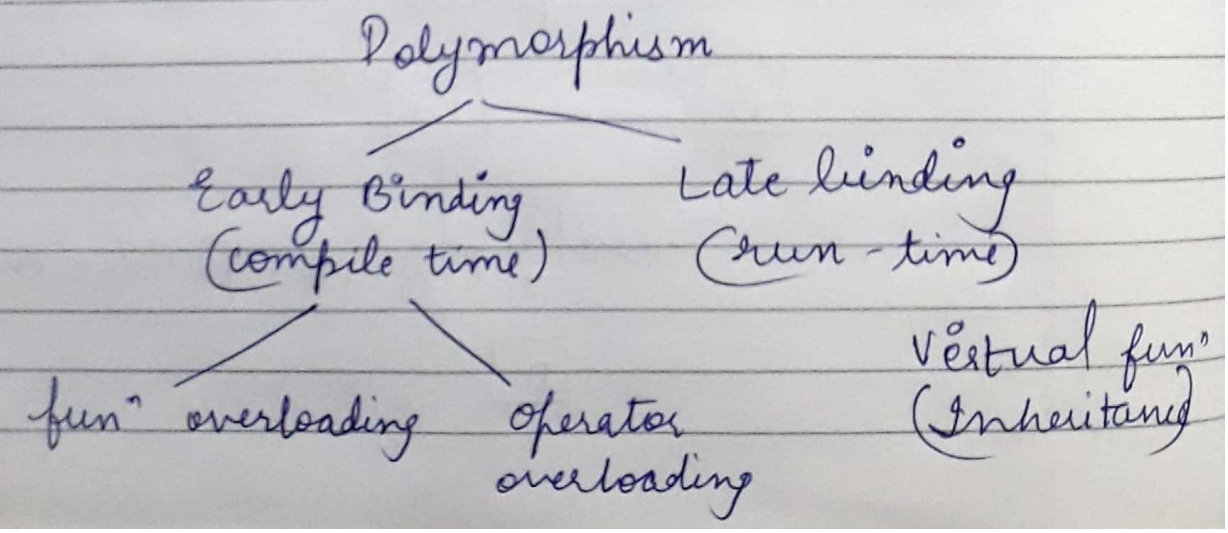


Run-time polymorphism in C++

can be achieved using the concept of virtual fun" in the base class

If the base class & the derived class both have similar fun" having same prototype than to achieve this kind of polymorphism, we create the base class fun" as a virtual fun" in the main fun". Base class pointer is created in the main fun" & it is pointing to either base class object or derived class object.

When the pointer points to base class fun" is called. If it points to derived class obj. then the derived class fun" is called. This is termed as run time polymorphism or late binding.



```
class Base
{
    public:
virtual void show()
        {
            cout <<"Base";
        }

        void disp ()
        {
            cout <<" display";
        }

};
class derived : public Base
{
    public:

        void show()
        {
            cout <<"derived";
        }

};
main ()
{
    Base * b, b1;
    derived d;
```

```
    b1. show ();        // O/P -> Base
    b1. disp ();        // O/P -> display
    d. show ();         // O/P -> Base
    d. disp ();         // O/P -> display
    b = & d;
    b -> show ();       // O/P -> Base
                           when virtual
                           is not written
                        // O/P -> derived
}
```

☆ Virtual Destructors

Like virtual funⁿ, we can have virtual distructors. But we can not have virtual constructor Because a constructor is used to create an object and the virtual table is created only after the base class pointer points to the derived class object.

```
class Base
{
virtual ~ Base ()
    {
        cout << " Base ";
    }
};
{
```

```
class derived : public Base
{
    ~derived()
    {
        cout << "derived";
    }
};

main()
{   Base *b;
    derived d;
    b = &d;
    delete b;
}
```

In this particular prog delete b statement is executed, the base class destructor is called releasing the memory of base class and leaving behind the memory of derived obj which result in memory leak.

Pure Virtual Fun"

A virtual fun" that has no body i.e it is equated to zero, this is called as a pure virtual fun". A pure virtual fun" can only exit in base class & it has to be redefine in the corresponding derived class. A base class with pure virtual

fun^n is turned as abstract class & an obj. of abstract class cannot be instantiated.

```
class Base
{
    virtual fun1()
    {
    }
    virtual func2()=0; // pure virtual
};

class der : public Base
{   void func2 () { }
};
```