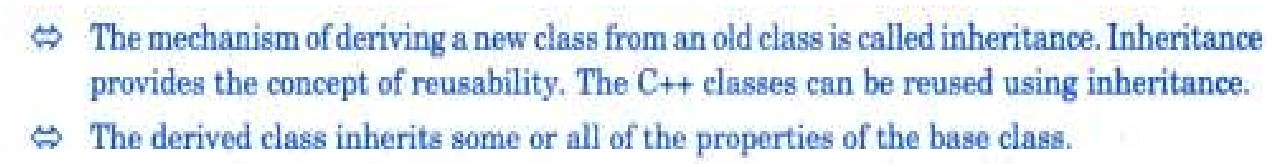
INHERITANCE-EXTENDING CLASSES

Ques: **What does inheritance mean? When do we use protected visibility specifier to a class member?**

OR

How do you inherit private member into derive class? Explain with example.

Ans:



⇔ The mechanism of deriving a new class from an old class is called inheritance. Inheritance provides the concept of reusability. The C++ classes can be reused using inheritance.

⇔ The derived class inherits some or all of the properties of the base class.

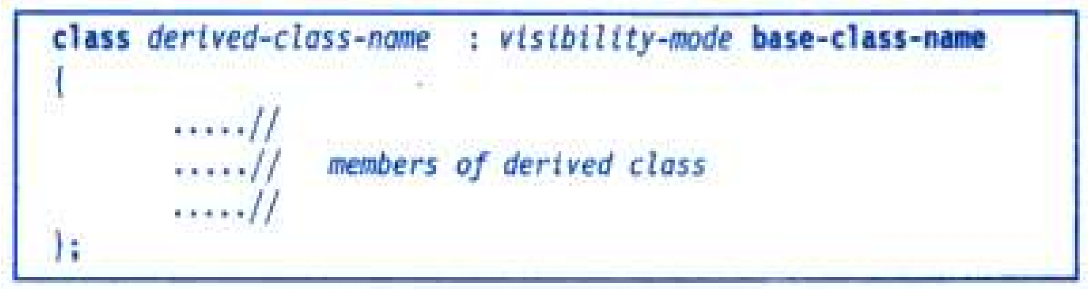
The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class. And, the class, which inherits properties of other class, is called **Child** or **Derived** or **Sub** class.

NOTE: All members of a class except Private are inherited.

Purpose of Inheritance

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

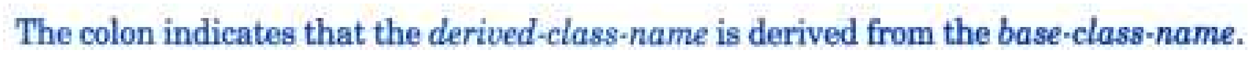
Basic Syntax of Inheritance



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

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

Visibility (Access) Mode is used to specify, the mode in which the properties of base-class will be inherited into derived-class, public, private or protected. The default visibility mode is private.


The colon indicates that the *derived-class-name* is derived from the *base-class-name*.

<u>Making a private member inheritable using protected:</u>

A private member of a class cannot be inherited, so it is not available for the derived class.

1. To make a private member inheritable, protected visibility mode is used.
2. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. it cannot be accessed by the functions outside these two classes.

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

Example:

class subclass : protected Superclass

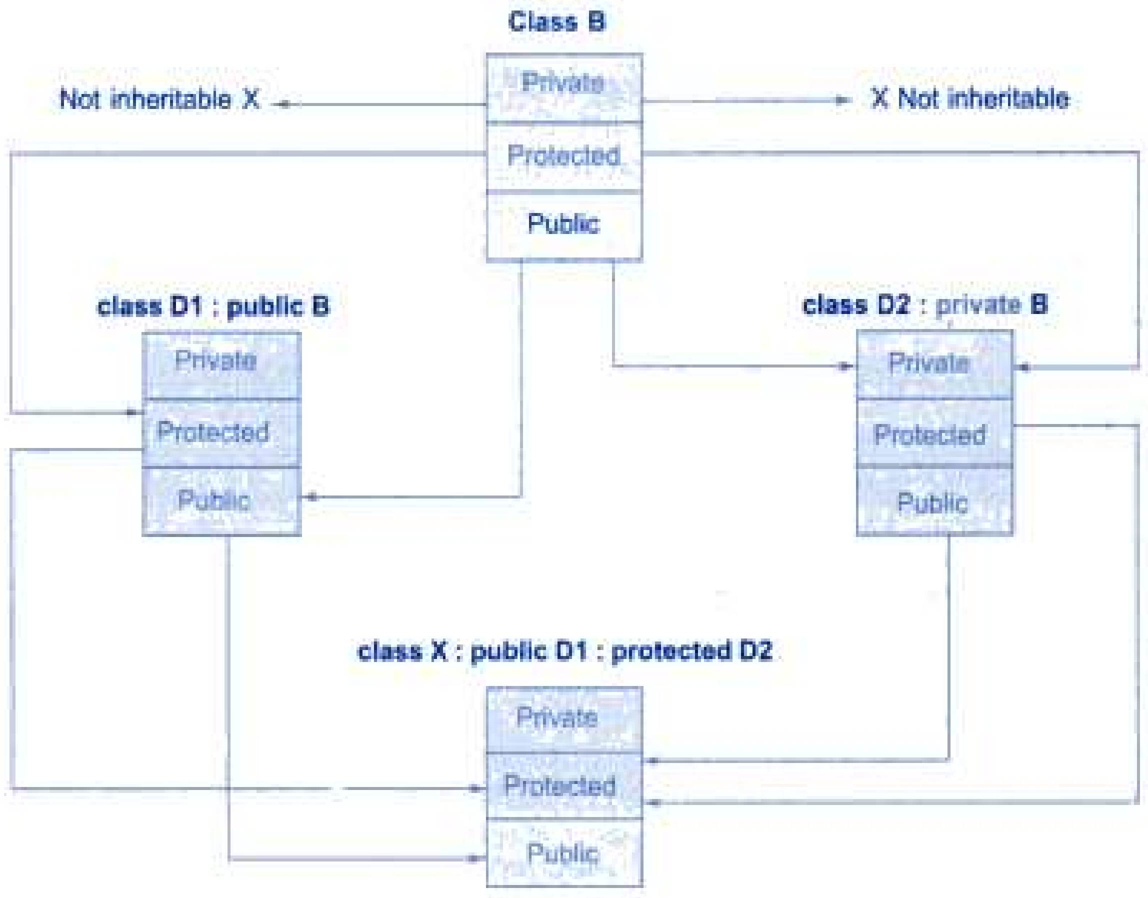


Fig. 8.4 ⇔ *Effect of inheritance on the visibility of members*

Example:

```
Class Base
{
   public:
     int a;
   protected:
     int b;
   private:
     int c;
};

class Derived:private Base
{
   void F1()
   {
      a = 10;  //Allowed
      b = 20;  //Allowed
      c = 30;  //Private member of base is not Allowed, Compiler Error
   }
};

class Derived2:public Derived
{
   void F2()
   {
      a = 10;  //Not Allowed, Compiler Error, a is private member of Derived now
      b = 20;  //Not Allowed, Compiler Error, b is private member of Derived now
      c = 30;  //Not Allowed, Compiler Error
   }
};

int main()
{
   Derived obj;
   obj.a = 10;  //Not Allowed, Compiler Error
   obj.b = 20;  //Not Allowed, Compiler Error
   obj.c = 30;  //Not Allowed, Compiler Error

}
```

SOME MORE ON INHERITANCE, CAREFULLY SEE THE QUESTION, IF FOLLOWING ASKED THEN ONLY WRITE IT.

Ques: What is visibility modifier? List out them and differentiates with proper example.

Inheritance Visibility Mode/Access Specifier

There are 3 visibility Mode for a class in C++. These access specifiers define how the members of the class can be accessed. C++ supports the following access modifiers:

1. Private Modifier
2. Protected Modifier
3. Public Modifier

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 become protected members of Sub class.

*class subclass : **protected** Superclass*

	Own class	Derived class	main()
Private	✓	✗	✗
Protected	✓	✓	✗
Public	✓	✓	✓

Example:

For this write above program

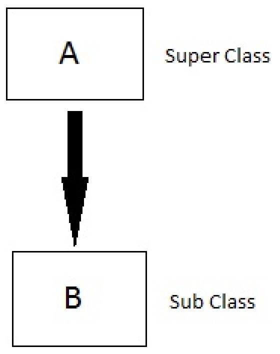
Types of Inheritance

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

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance

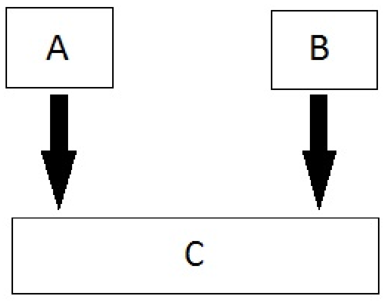
Single Inheritance

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



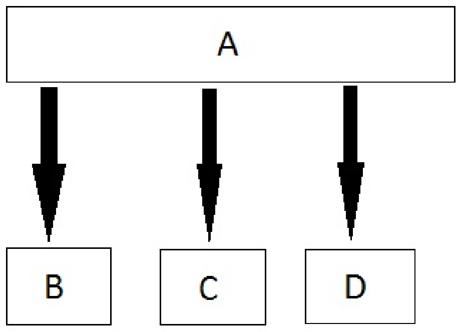
Multiple Inheritance

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



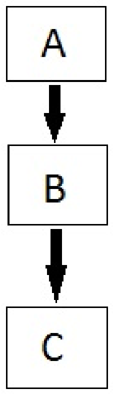
Hierarchical Inheritance

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



Multilevel Inheritance

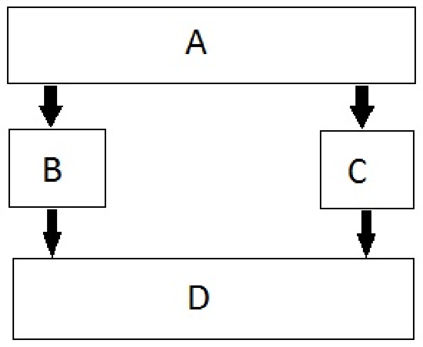
In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one is sub class for the other.



Hybrid Inheritance

Hybrid Inheritance is combination of Hierarchical and Multilevel Inheritance.

Example:



Ques: What is inheritance? Differentiates multiple and multilevel inheritance with example.

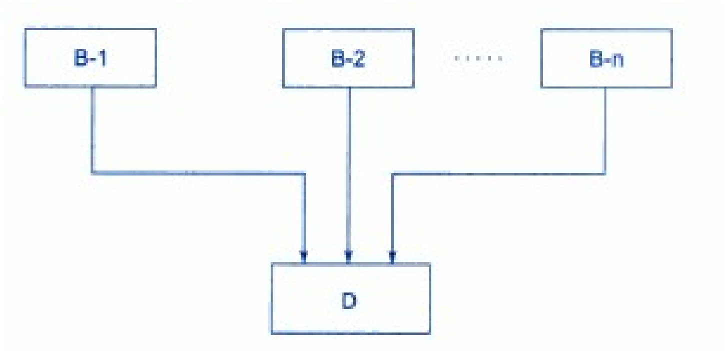
Ans: what is inheritance is given above. For difference write definition then diagram then syntax then example. Prepare this from book.

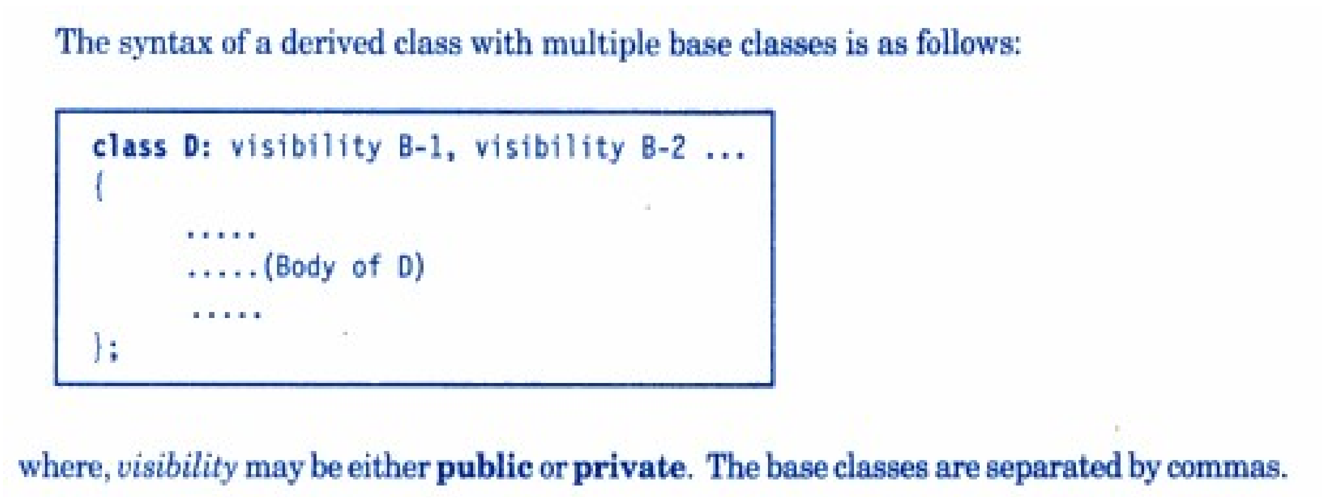
Ques: What is inheritance? Explain any one inheritance type with example.

Ans: inheritance intro and types given above, write here

Multiple Inheritance

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



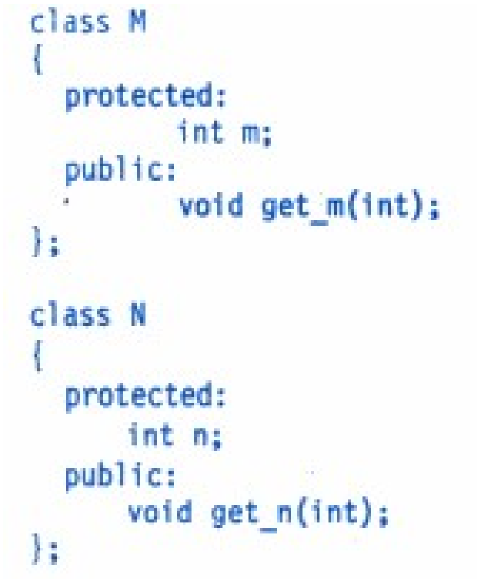


The syntax of a derived class with multiple base classes is as follows:

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

where, *visibility* may be either **public** or **private**. The base classes are separated by commas.

Example of multiple inheritance:



```
class M
{
  protected:
         int m;
  public:
         void get_m(int);
};

class N
{
  protected:
      int n;
  public:
      void get_n(int);
};
```

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

void M :: get_m(int x)
{
      m = x;
}

void N :: get_n(int y)
{
      n = y;
}

void P :: display(void)
{
      cout << "m = " << m << "\n";
      cout << "n = " << n << "\n";
      cout << "m*n = " << m*n << "\n";
}

int main()
{
```

```
      P p;

      p.get_m(10);
      p.get_n(20);
      p.display();

      return 0;
}
```

Output:

```
m = 10
n = 20
m*n = 200
```

Note: Example (program) and syntax of all inheritance types separately given in textbook, so you can select any one program from it.

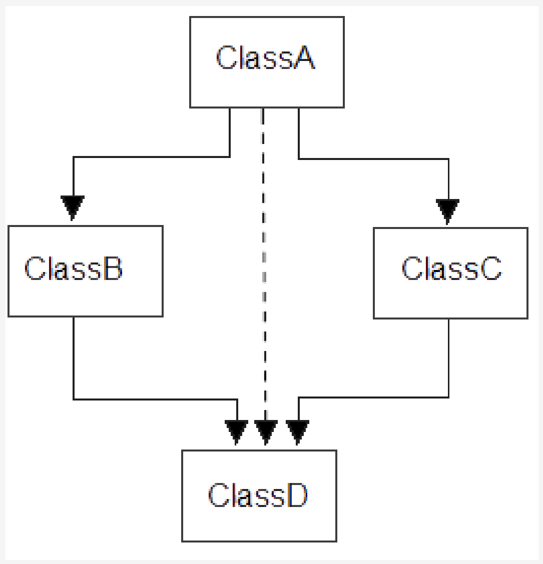
Ques: What is ambiguity in hybrid inheritance? How ambiguities remove from compile time? Explain with example.

OR

What is inheritance ? When ambiguity occurs in hybrid inheritance. What are solutions to avoid ambiguity.

Ans: Hybrid Inheritance is combination of Hierarchical and Multilevel Inheritance.

Ambiguity in C++ occurs when a derived class has two base classes and these two base classes have one common base class. Consider the following figure:



Example:

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

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

Which **display**() function is used by the derived class when we inherit these two classes? We can solve this problem by defining a named instance within the derived class, using the class resolution operator with the function as shown below:

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

We can now use the derived class as follows:

```
int main()
{
      P p;
      p.display();
}
```

Ambiguity in single inheritance (write it if asked in question):

Ambiguity may also arise in single inheritance applications. For instance, consider the following situation:

```
class A
{
  public:
      void display()
      {
             cout << "A\n";
      }
};
class B : public A
{
  public:
     void display()
     {
             cout << "B\n";
     }
};
```

In this case, the function in the derived class overrides the inherited function and, therefore, a simple call to **display()** by **B** type object will invoke function defined in **B** only. However, we may invoke the function defined in **A** by using the scope resolution operator to specify the class.

Example:

```
int main()
{
     B b;                    // derived class object
     b.display();            // invokes display() in B
     b.A::display();         // invokes display() in A
     b.B::display();         // invokes display() in B

      return 0;
}
```

This will produce the following output:

```
B
A
B
```

Ques: Explain virtual base class with example.

Multipath inheritance may lead to duplication of inherited members from a 'grandparent' base class. This may be avoided by making the common base class a virtual base class.

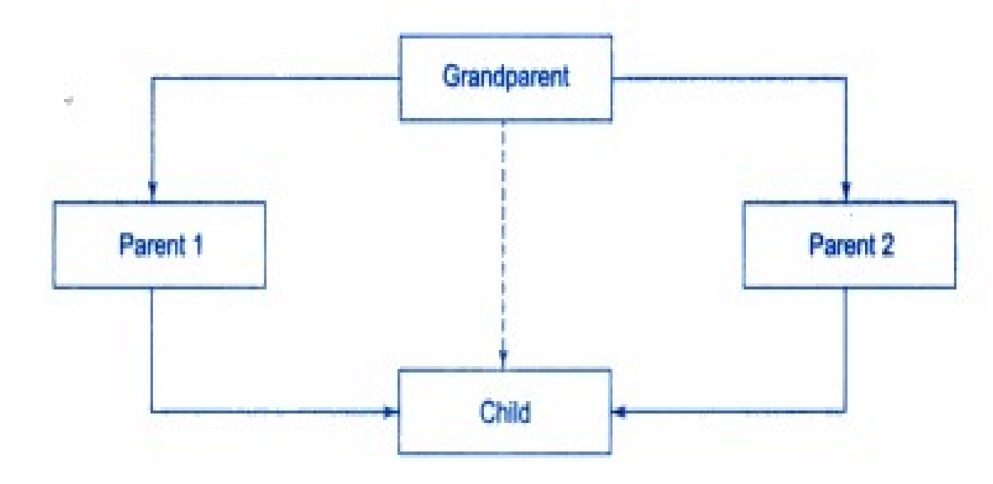


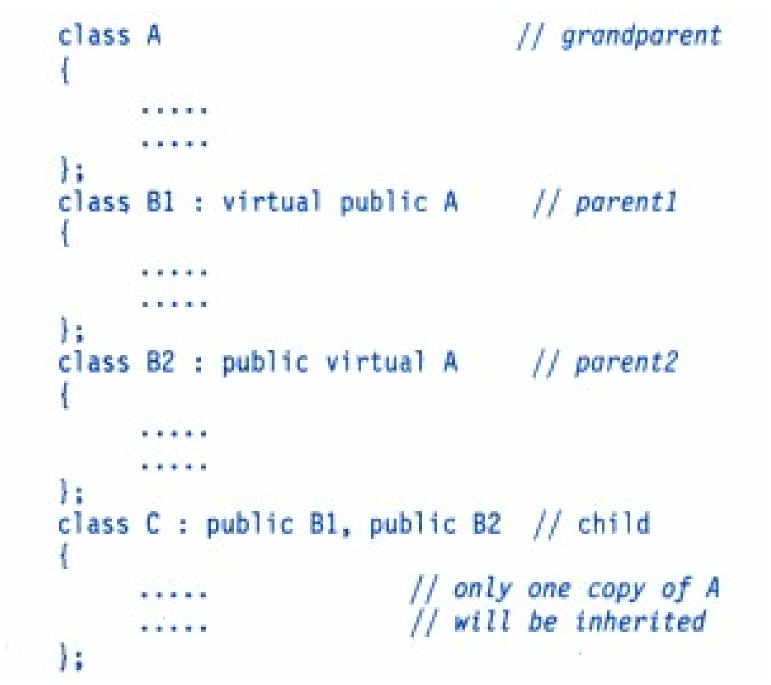
Diagram of multipath inheritance

The 'child' has two *direct base classes* 'parent1' and 'parent2' which themselves

have a common base class 'grandparent'. The 'child' inherits the traits of 'grandparent' via two separate paths. It can also inherit directly as shown by the broken line. The 'grandparent' is sometimes referred to as *indirect base class*.

Inheritance by the 'child' as shown in Fig. 8.12 might pose some problems. All the public and protected members of 'grandparent' are inherited into 'child' twice, first via 'parent1' and again via 'parent2'. This means, 'child' would have *duplicate* sets of the members inherited from 'grandparent'. This introduces ambiguity and should be avoided.

The duplication of inherited members due to these multiple paths 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 below:



```
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
};
```

If two copies will be there of members in a class then it will generate error, bco'z compiler can't differentiate between two copies of members.

To remove multiple copies we must use virtual **base** class.

Example using virtual base class

```
#include<iostream.h>
#include<conio.h>
class ClassA
{
    public:
    int a;
};
class ClassB : virtual public ClassA
{
```

```
          public:
          int b;
     };
     class ClassC : virtual public ClassA
     {
          public:
          int c;
     };
     class ClassD : public ClassB, public ClassC
     {
          public:
          int d;
     };
     void main()
     {
                    ClassD obj;
                    obj.a = 10;        //Statement 1
                    obj.a = 100;       //Statement 2
                    obj.b = 20;
                    obj.c = 30;
                    obj.d = 40;
                    cout<< "\n A : "<< obj.a;
                    cout<< "\n B : "<< obj.b;
                    cout<< "\n C : "<< obj.c;
                    cout<< "\n D : "<< obj.d;
 }
```

Output :

```
A : 100
```

```
B : 20
C : 30
D : 40
```

According to the above example, **ClassD** have only one copy of **ClassA** and statement 4 will overwrite the value of **a**, given in statement 3.

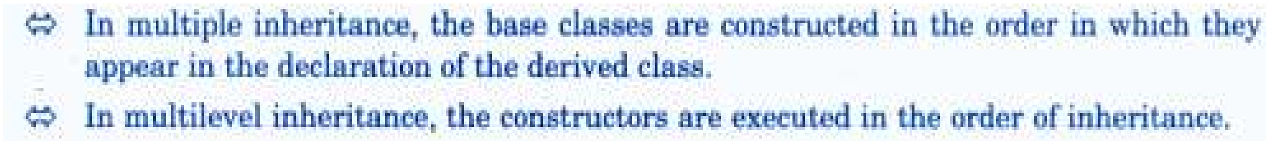
CONSTRUCTOR IN DERIVED CLASSES:

Ques: What do you mean by constructor in derived classes? If a constructor is present in derived and base class then which constructor function gets executed? Explain with example.

OR

In what order are the class constructor and destructor called when derived class object is created?

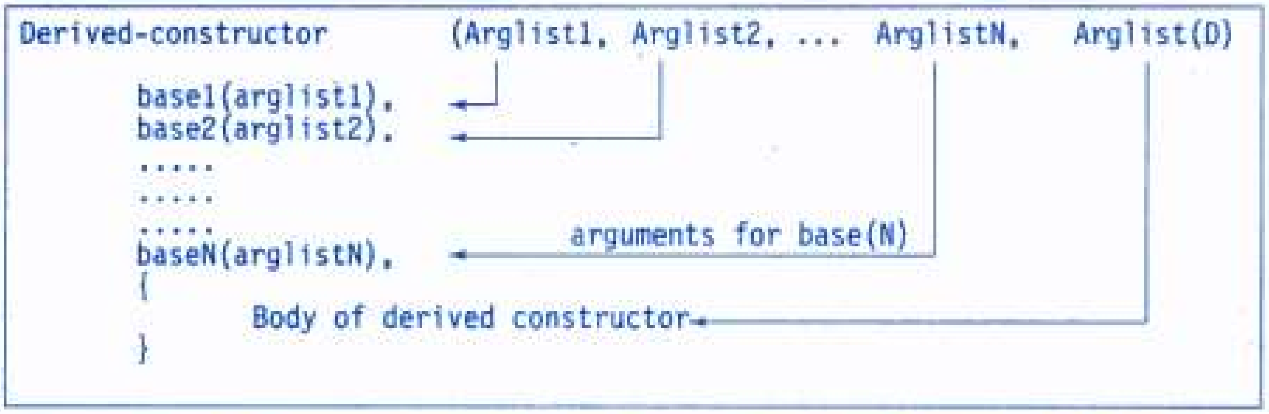
Ans: Base class constructors are always called in the derived class constructors. Whenever you create derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution.



⇔ In multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class.

⇔ In multilevel inheritance, the constructors are executed in the order of inheritance.

The general form of defining a derived constructor is:

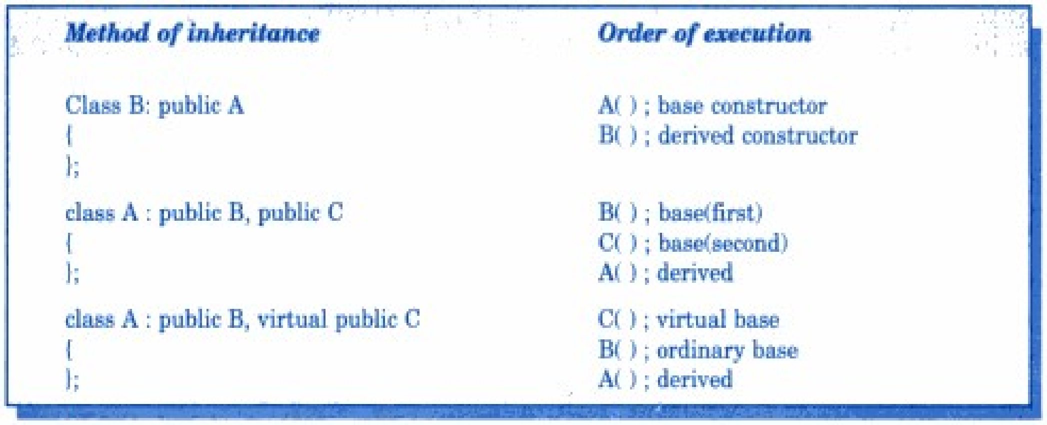


```
Derived-constructor      (Arglist1, Arglist2, ... ArglistN,   Arglist(D)

        base1(arglist1),
        base2(arglist2),
        .....
        .....
        .....                       arguments for base(N)
        baseN(arglistN),
        {
              Body of derived constructor
        }
```

The header line of *derived-constructor* function contains two parts separated by a colon(:). The first part provides the declaration of the arguments that are passed to the *derived-constructor* and the second part lists the function calls to the base constructors.

base1(arglist1), base2(arglist2) ... are function calls to base constructors **base1(), base2()**, ... and therefore *arglist1, arglist2, ...* etc. represent the actual parameters that are passed to the base constructors. *Arglist1* through *ArglistN* are the argument declarations for base constructors *base1* through *baseN*. *ArglistD* provides the parameters that are necessary to initialize the members of the derived class.

Execution of base class constructor is:



Method of inheritance	***Order of execution***

Class B: public A
{
};
	A( ) ; base constructor
	B( ) ; derived constructor

class A : public B, public C
{
};
	B( ) ; base(first)
	C( ) ; base(second)
	A( ) ; derived

class A : public B, virtual public C
{
};
	C( ) ; virtual base
	B( ) ; ordinary base
	A( ) ; derived

Example:

```
class alpha
{
      int x;
   public:
      alpha(int i)
      {
            x = i;
            cout << "alpha initialized \n";
      }
      void show_x(void)
      { cout << "x = " << x << "\n"; }
};

class beta
{
      float y;
   public:
      beta(float j)
      {
            y = j;
            cout << "beta initialized \n";
      }
      void show_y(void)
      { cout << "y = " << y << "\n"; }
};

class gamma: public beta, public alpha
{
      int m, n;
   public:
      gamma(int a, float b, int c, int d):
            alpha(a), beta(b)
      {
            m = c;
            n = d;
            cout << "gamma initialized \n";
      }
```

```
      void show_mn(void)
      {
            cout << "m = " << m << "\n"
                 << "n = " << n << "\n";
      }
};
```

```
int main()
{
    gamma g(5, 10.75, 20, 30);
    cout << "\n";
    g.show_x();
    g.show_y();
    g.show_mn();

    return 0;
}
```

Output:

```
beta initialized
alpha initialized
gamma initialized

x = 5
y = 10.75
m = 20
n = 30
```

Note:

beta is initialized first, although it appears second in the derived constructor. This is because it has been declared first in the derived class header line. Also, note that **alpha(a)** and **beta(b)** are function calls. Therefore, the parameters should not include types.

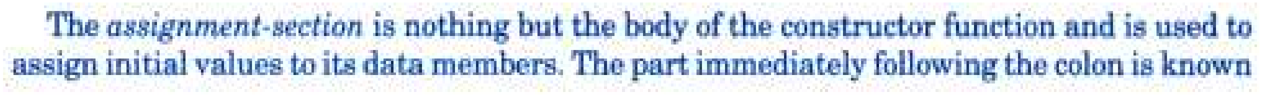
Here our answer is ended.

If initialization method is asked in exam then only write below answer:

C++ supports another method of initializing the class objects. This method uses what is known as initialization list in the constructor function. This takes the following form:

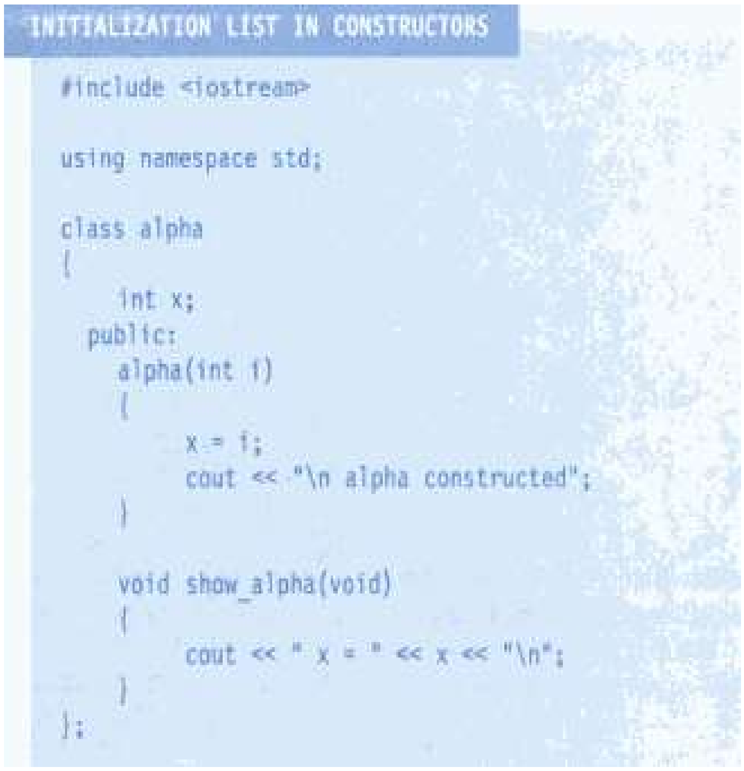
c++ uses the following method for initializing the class objects. In this we have a "initialization list" in the constructor function. Its form is:

```
constructor (arglist) : intialization-section
{
      assignment-section
}
```



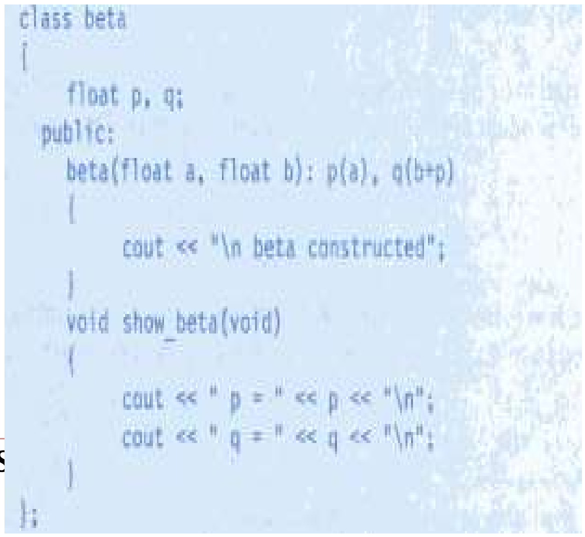
The *assignment-section* is nothing but the body of the constructor function and is used to assign initial values to its data members. The part immediately following the colon is known as the initialization section. We can provide initial values to the base constructors and also to initialize its own class members.

Example:

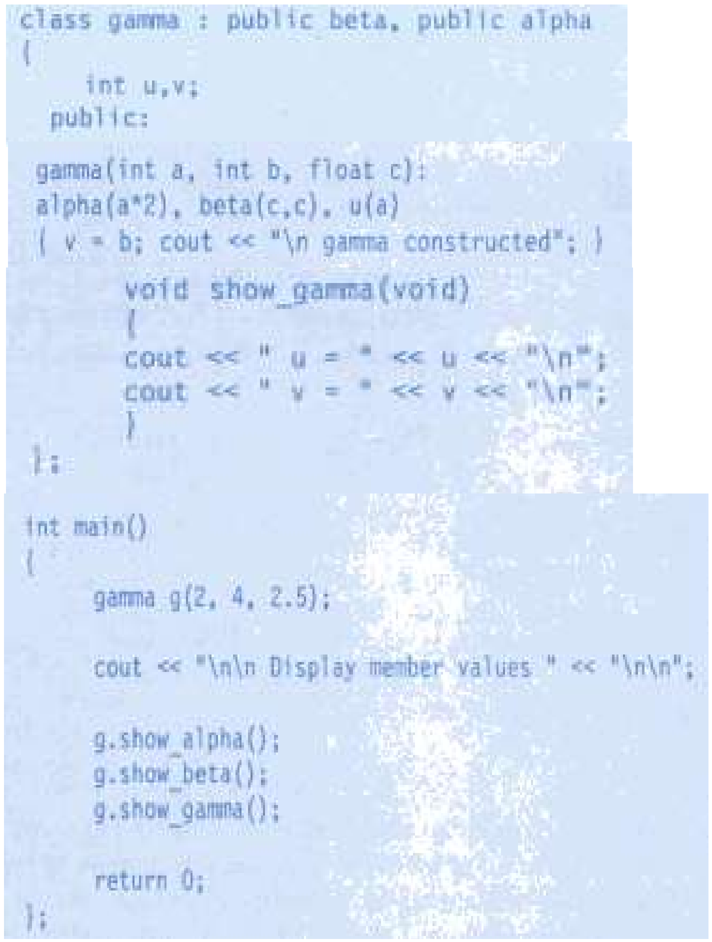


INITIALIZATION LIST IN CONSTRUCTORS

```
#include <iostream>

using namespace std;

class alpha
{
    int x;
  public:
    alpha(int i)
    {
         x = i;
         cout << "\n alpha constructed";
    }

    void show_alpha(void)
    {
         cout << " x = " << x << "\n";
    }
};
```

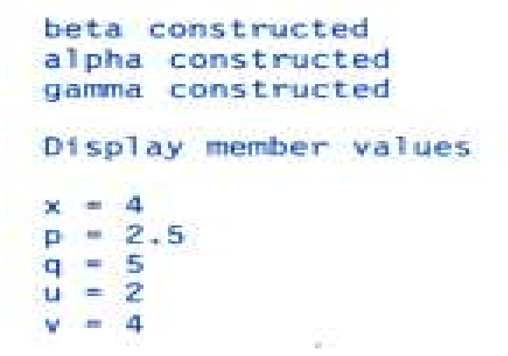


```
class beta
{
    float p, q;
  public:
    beta(float a, float b): p(a), q(b+p)
    {
         cout << "\n beta constructed";
    }
    void show_beta(void)
    {
         cout << " p = " << p << "\n";
         cout << " q = " << q << "\n";
    }
};
```



```
class gamma : public beta, public alpha
{
    int u,v;
  public:
 gamma(int a, int b, float c):
 alpha(a*2), beta(c,c), u(a)
 { v = b; cout << "\n gamma constructed"; }
        void show_gamma(void)
        {
        cout << " u = " << u << "\n";
        cout << " v = " << v << "\n";
        }
 };
int main()
{
     gamma g(2, 4, 2.5);

     cout << "\n\n Display member values " << "\n\n";

     g.show_alpha();
     g.show_beta();
     g.show_gamma();

     return 0;
};
```

OUTPUT:



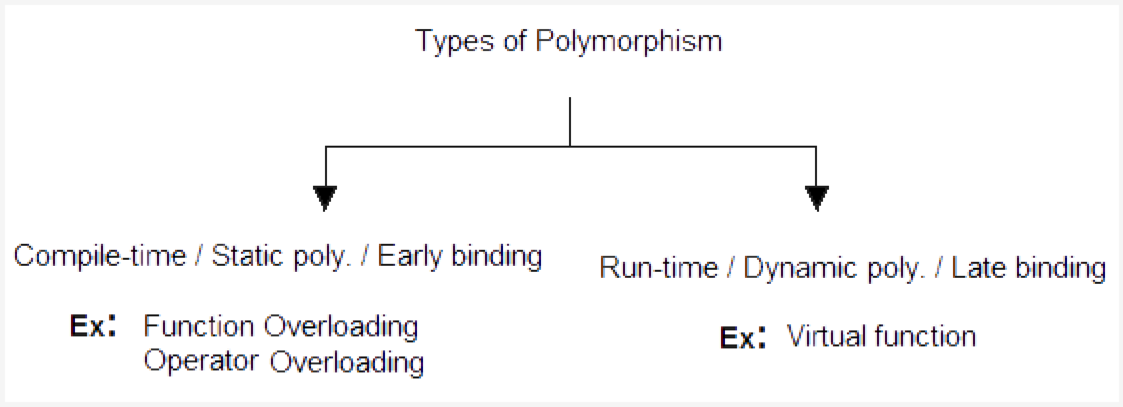
```
beta constructed
alpha constructed
gamma constructed

Display member values

x = 4
p = 2.5
q = 5
u = 2
v = 4
```

POLYMORPHISM
POINTERS
&
VIRTUAL FUNCTIONS

Ques: How is polymorphism achieved at run time and compile time?

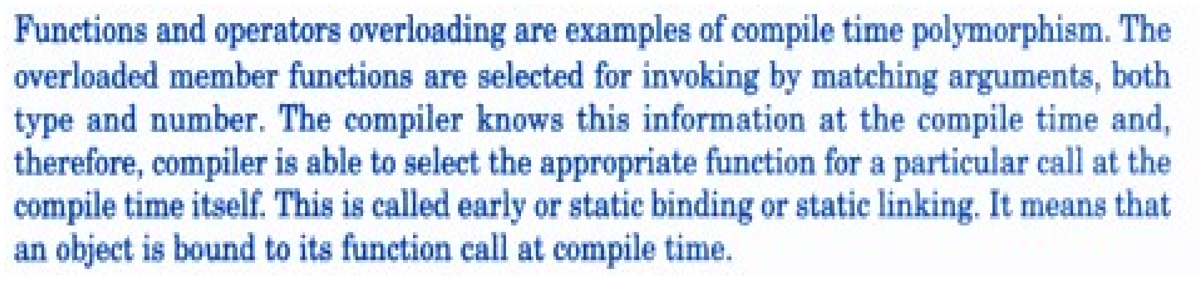
Ans: Polymorphism means more than one function with same name, with different working. Polymorphism can be static or dynamic.

- In static polymorphism memory will be allocated at compile-time.
- In dynamic polymorphism memory will be allocated at run-time.
- Both function overloading and operator overloading are an examples of static polymorphism.
- Virtual function is an example of dynamic polymorphism.
- **Static polymorphism** is also known as early binding and **compile-time polymorphism**.
- **Dynamic polymorphism** is also known as late binding and **run-time polymorphism**.



- **Function Overloading**: More than one function with same name, with different signature in a class or in a same scope is called function overloading.

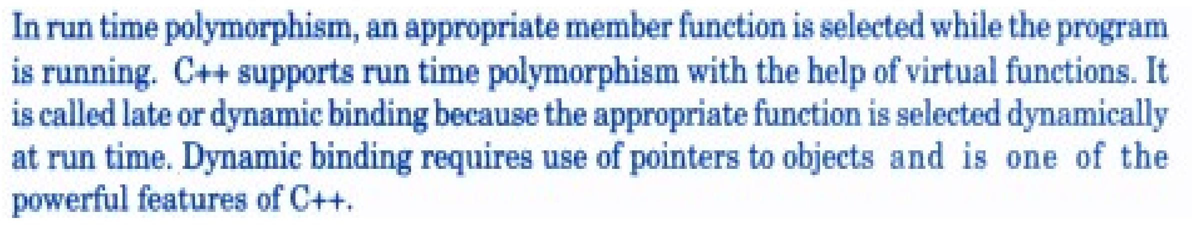
- **Operator Overloading:** Operator overloading is a way of providing new implementation of existing operators to work with user-defined data types.



Functions and operators overloading are examples of compile time polymorphism. The overloaded member functions are selected for invoking by matching arguments, both type and number. The compiler knows this information at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early or static binding or static linking. It means that an object is bound to its function call at compile time.

- **Virtual function:** Virtual function is used in situation, when we need to invoke derived class function using base class pointer.

  Giving new implementation of derived class method into base class and the calling of this new implemented function with base class's object is done by making base class function as virtual function.



In run time polymorphism, an appropriate member function is selected while the program is running. C++ supports run time polymorphism with the help of virtual functions. It is called late or dynamic binding because the appropriate function is selected dynamically at run time. Dynamic binding requires use of pointers to objects and is one of the powerful features of C++.

Function Overriding

Polymorphism means having multiple forms of one thing. In inheritance, polymorphism is done, by method overriding, when both super and sub class have member function with same declaration but different definition.

If we inherit a class into the derived class and provide a definition for one of the base class's function again inside the derived class, then that function is said to be **overridden**, and this mechanism is called **Function Overriding**

Requirements for Overriding

1. Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
2. Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

Example of Function Overriding :

```
Class base
{
 public:
 void show()
{
 cout<<”base class” ;
}
};
class derived : public base
{
 public:
 void show()
{
 cout<<”derived class”;
}
};
```

In this example, function **show()** is overridden in the derived class. Now let us study how these overridden functions are called in **main()** function.

Function Call Binding with class Objects

Connecting the function call to the function body is called **Binding**. When it is done before the program is run i.e. at compile time, its called **Early** Binding or **Static** Binding or **Compile-time** Binding.

If main() of above class specification is written as:

```
void main()
{
 base b;      //base class object
 derived d;   // derived class object
 b.show();   //early binding occurs
}
```

Output : base class

In the above example, we are calling the overrided function using Base class object. Base class object will call base version of the function and derived class's object will call the derived version of the function.

Function Call Binding using Base class Pointer

when we use a Base class's pointer or reference to hold Derived class's object, then Function call Binding gives following results.

Example: if we write main() section of above base and derived class specification then we have:

```
void main()
{
 base *b;     //base class pointer
 derived d;   //derived class object
  b=&d;
 b->show();   //early binding occurs
}
```

Output : Base class

In the above example, although, the object is of Derived class, still Base class's method is called. This happens due to Early Binding.

Compiler on seeing **Base class's pointer**, set call to Base class's **show()** function, without knowing the actual object type.

Ques: When do we need virtual functions? Write down the rules for virtual functions.

OR

What is pure virtual function? Why we need pure virtual function? Write down rules for pure virtual function.

OR

When do we make a virtual function "pure"? Write down syntax for pure virtual function.

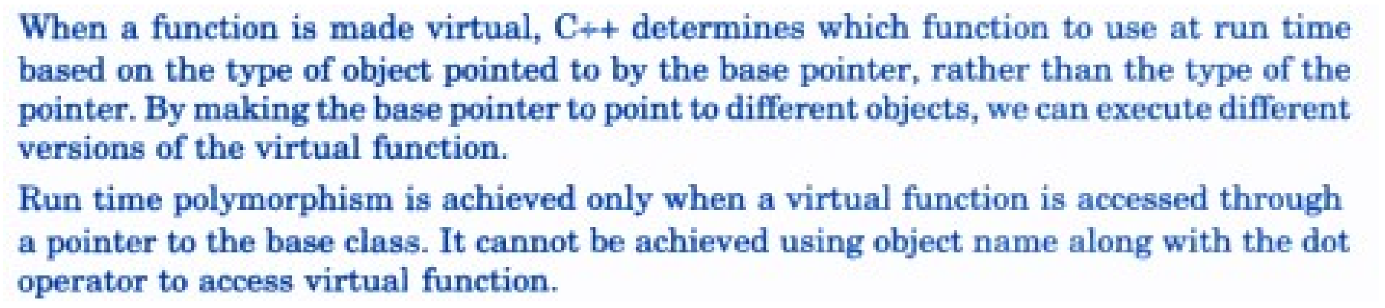
OR

State the rules to be observed when creating virtual function.

Ans: Virtual Functions

Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform **Late Binding** on this function.

Virtual Keyword is used to make a member function of the base class Virtual.



When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. By making the base pointer to point to different objects, we can execute different versions of the virtual function.

Run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class. It cannot be achieved using object name along with the dot operator to access virtual function.

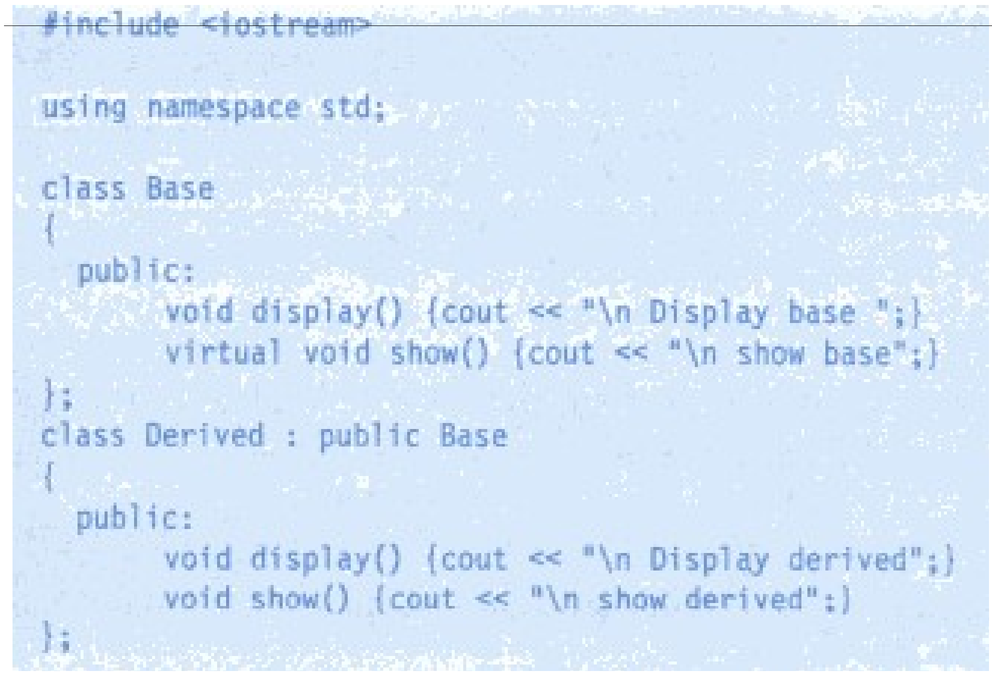
When we use Base class's pointer to hold Derived class's object, base class pointer or reference will always call the base version of the function

Using Virtual Keyword

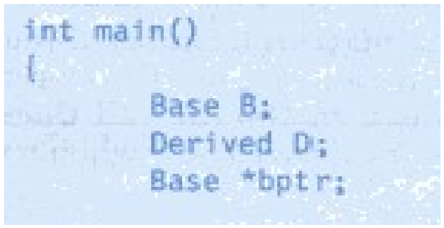
We can make base class's methods virtual by using **virtual** keyword while declaring them. Virtual keyword will lead to Late Binding of that method.

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer pointes to Derived class object.

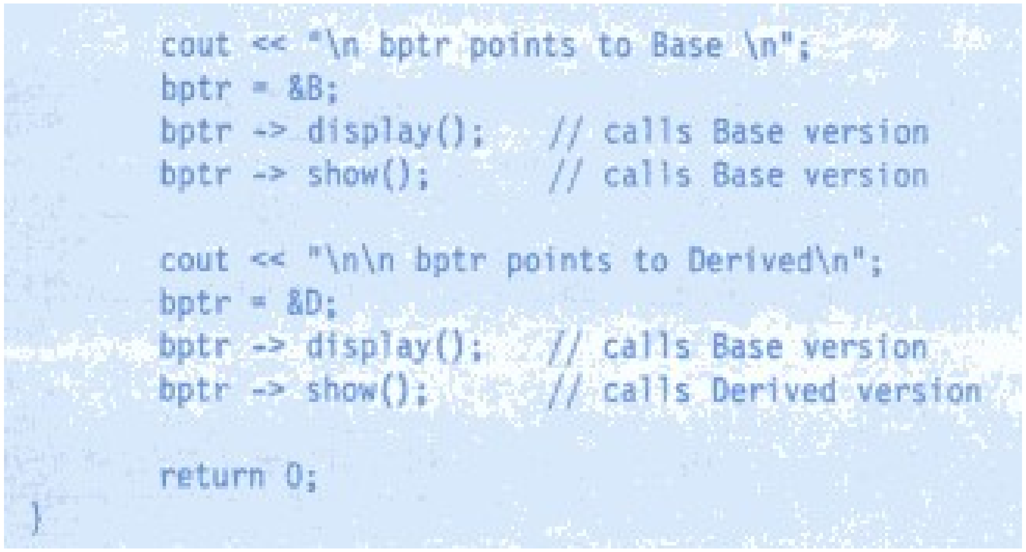
Example:



```
#include <iostream>

using namespace std;

class Base
{
  public:
        void display() {cout << "\n Display base ";}
        virtual void show() {cout << "\n show base";}
};
class Derived : public Base
{
  public:
        void display() {cout << "\n Display derived";}
        void show() {cout << "\n show derived";}
};
```

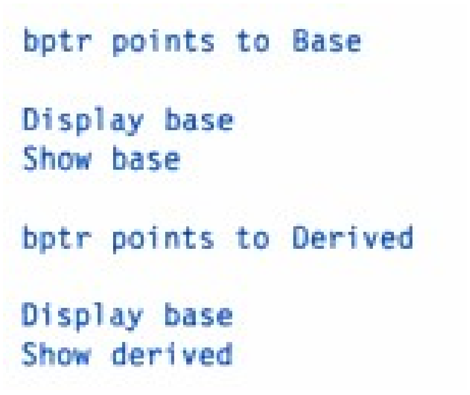


```
int main()
{
        Base B;
        Derived D;
        Base *bptr;
```



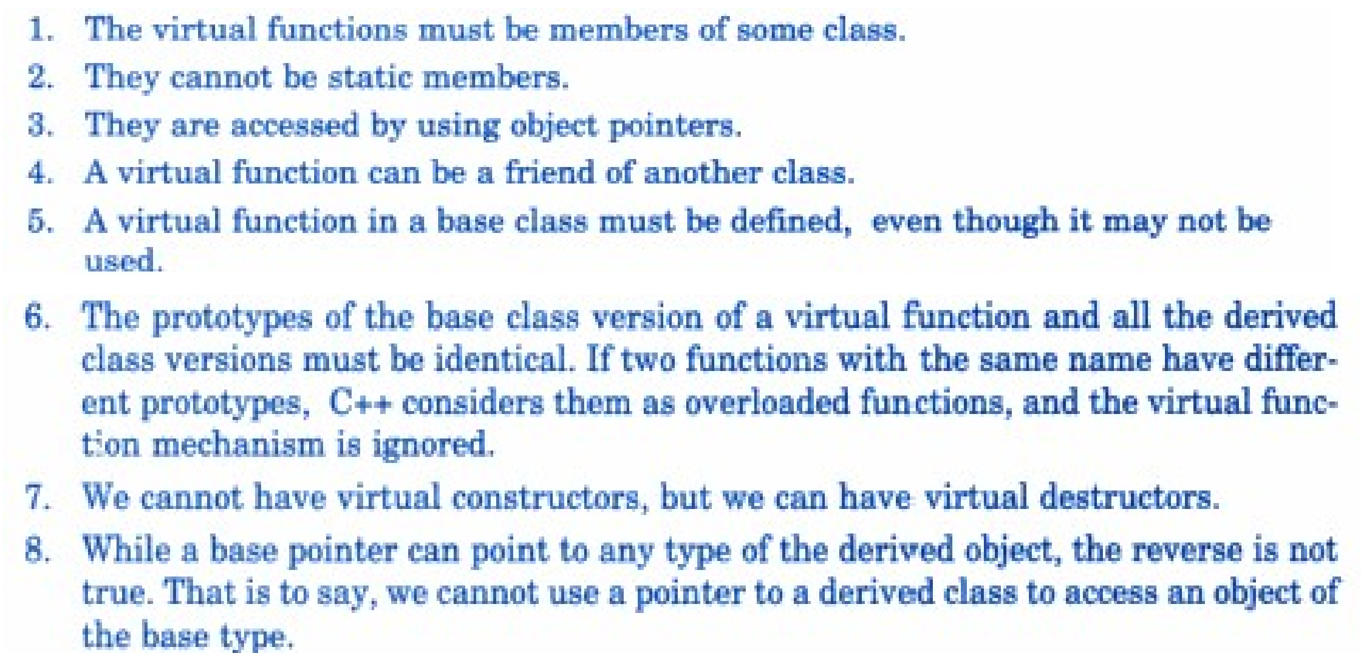
```
        cout << "\n bptr points to Base \n";
        bptr = &B;
        bptr -> display();    // calls Base version
        bptr -> show();       // calls Base version

        cout << "\n\n bptr points to Derived\n";
        bptr = &D;
        bptr -> display();    // calls Base version
        bptr -> show();       // calls Derived version

        return 0;
}
```

Output:

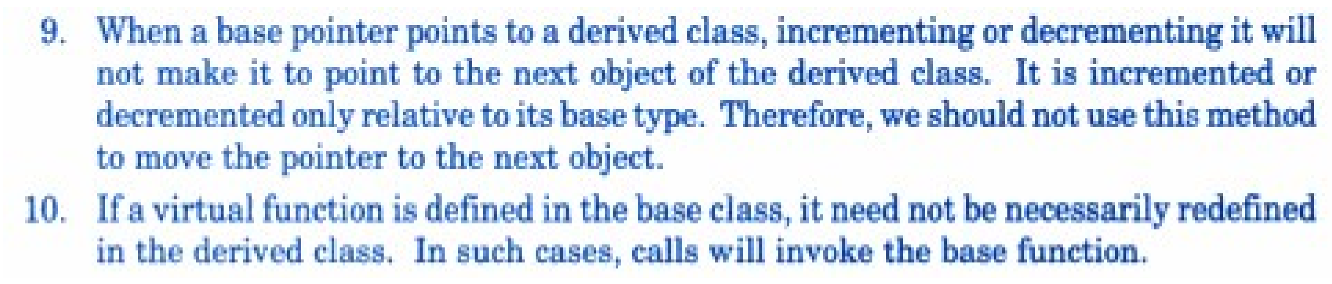


```
bptr points to Base

Display base
Show base

bptr points to Derived

Display base
Show derived
```

Rules for virtual function are:



1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.



9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

Important Points to Remember

1. Only the Base class Method's declaration needs the **Virtual** Keyword, not the definition.
2. If a function is declared as **virtual** in the base class, it will be virtual in all its derived classes.

Ques: What is Pure Virtual Functions

Ans: Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with=0. Here is the syntax for a pure virtual function,

```
Virtual void f() = 0;
```