Inheritance and Polymorphism

Contents

- What is inheritance?
- Derived and base class
- Inheritance types:
  - Single level
  - Multi level
  - Hierarchical
  - Multiple
  - Hybrid
- Virtual Base Class

Cont…

- Abstract Classes
- Pointer to object
- This Pointer
- Pointer to derived Class
- Virtual Function
- Pure Virtual function
- Early vs Late binding

Inheritance

- Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.
- This also provides an opportunity to reuse the code functionality and fast implementation time.
- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class.

Cont…

- This existing class is called the **base** class, and the new class is referred to as the **derived** class.
- **Syntax:**

  class derived-class: access-specifier base-class

- Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class.
- If the access-specifier is not used, then it is private by default.

Access Control and Inheritance

- A derived class can access all the non-private members of its base class.
- Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.
- A derived class inherits all base class methods with the following exceptions:
  - Constructors, destructors and copy constructors of the base class.
  - Overloaded operators of the base class.
  - The friend functions of the base class.

Access Control

- **Access Specifier and their scope**

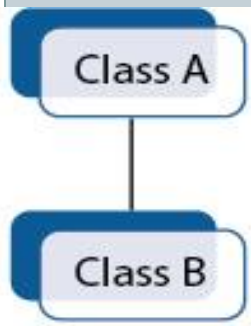
Base Class Access Mode	Derived Class Access Modes		
	Private derivation	Public derivation	Protected derivation
Public	Private	Public	Protected
Private	Not inherited	Not inherited	Not inherited
Protected	private	Protected	Protected

Function Type	Access Directly to		
	Private	Public	Protected
Class Member	Yes	Yes	Yes
Derived Class Member	No	Yes	Yes
Friend	Yes	Yes	Yes
Friend Class Member	Yes	Yes	Yes

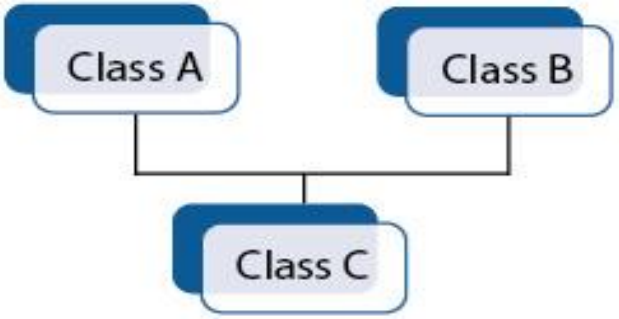
Access control to class members

Base-class member-access specifier	Type of inheritance		
	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class. Can be accessed directly by member functions, `friend` functions and nonmember functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`private`	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.

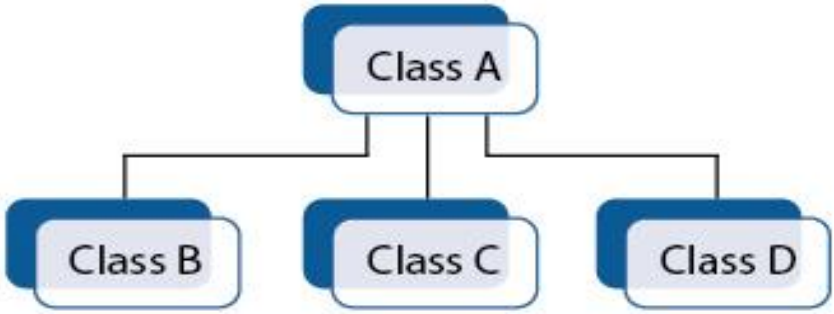
Types of Inheritance



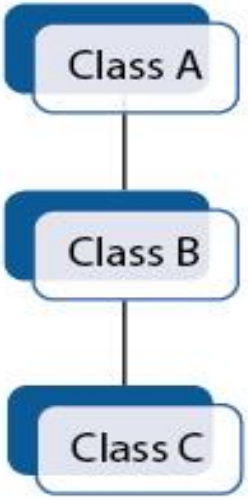
Single Inheritance



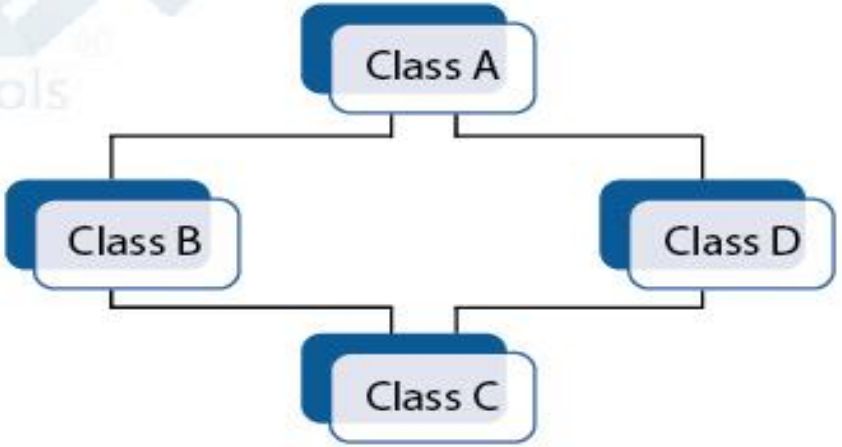
Multiple Inheritance



Hierarchical Inheritance



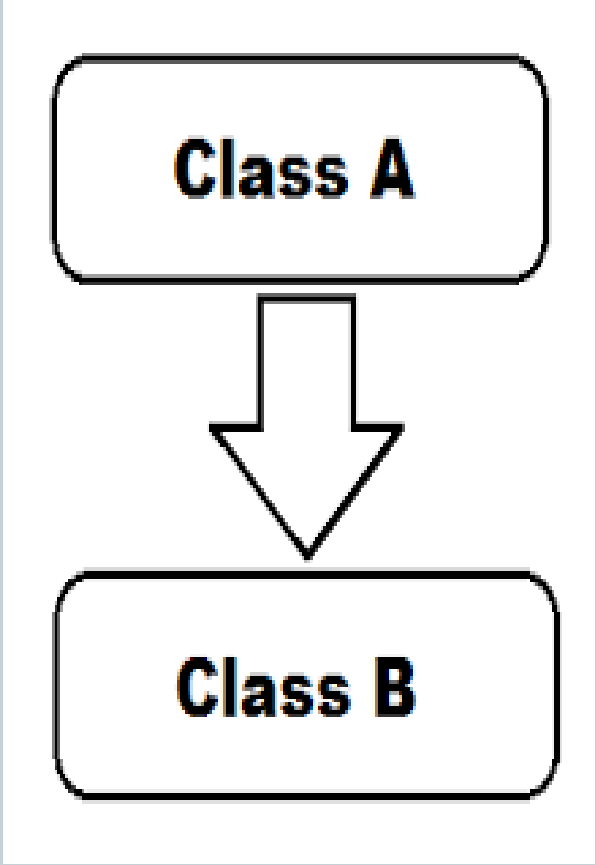
Multilevel Inheritance



Hybrid Inheritance

Single level inheritance

- In this type of inheritance one derived class inherits from only one base class.



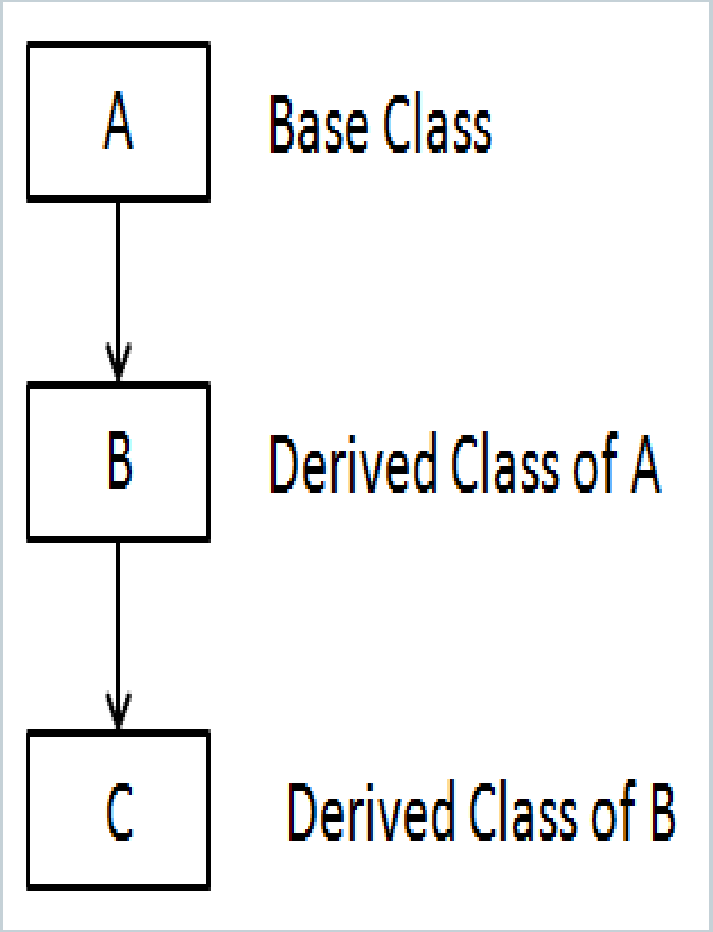
Program on single level inheritance


```
class A
 {

 public:
    void display()
     {
         cout<<"Base
   Class";
     }
 };
```

```
class B : public A
    {
   public:
     void display2()
      {
          cout<<"Derived Class";
      }

  };
            int main()
            {
              B b;
              b.display();
              b.display2();
                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.



Program on multilevel inheritance

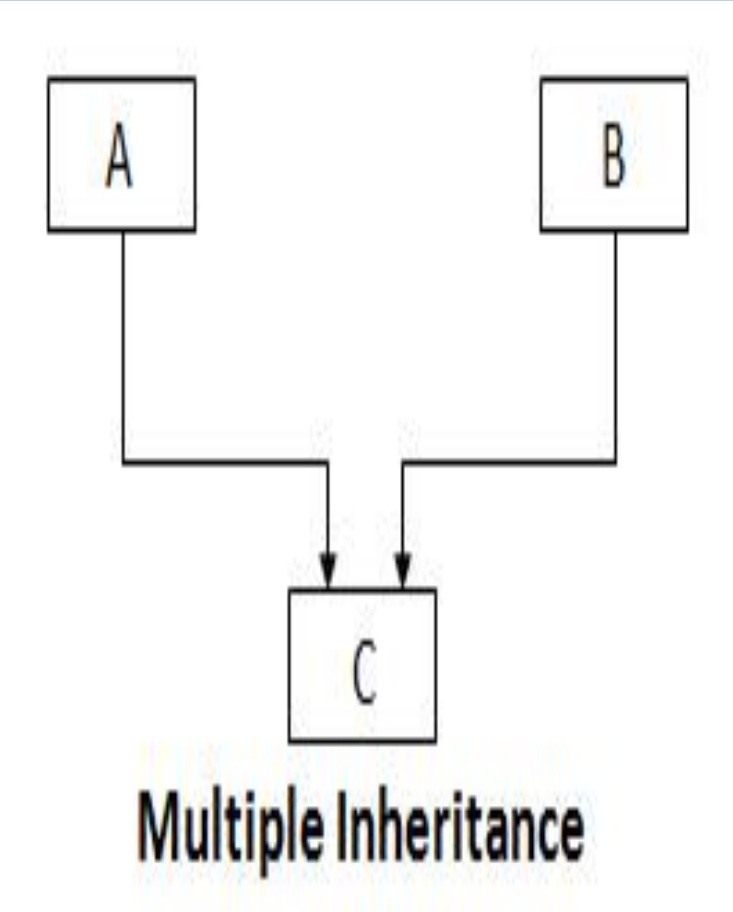

```
#include <iostream>

using namespace std;
class A
{

public:
  int a,b;

   void getdata()
   {
            cout<<"Enter a and b"<<endl;
      cin>>a>>b;
   }
};
```

```
class B : public A
{
public:
    int c;
    void sum()
     {

         c=a+b;

     }
};

class C : public B
{
public:
    void msg()
     {
        cout<<"Sum is"<<c;

     }
};
```

```
int main()
{
    C c;
    c.getdata();
    cout<<endl;
    c.sum();
    cout<<endl;
    c.msg();
    cout<<endl;

    return 0;
}
```

Multiple Inheritance

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



Ambiguity in multiple inheritance

- In multiple inheritance, there may be possibility that a class may inherit member functions with same name from two or more base classes and the derived class may not have functions with same name as those of its base classes.
- If the object of the derived class need to access one of the same named member function of the base classes then it result in ambiguity as it is not clear to the compiler which base's class member function should be invoked. The ambiguity simply means the state when the compiler confused.

Solution of ambiguity in multiple inheritance

- The ambiguity can be resolved by using the scope resolution operator to specify the class in which the member function lies.

Hierarchical inheritance

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

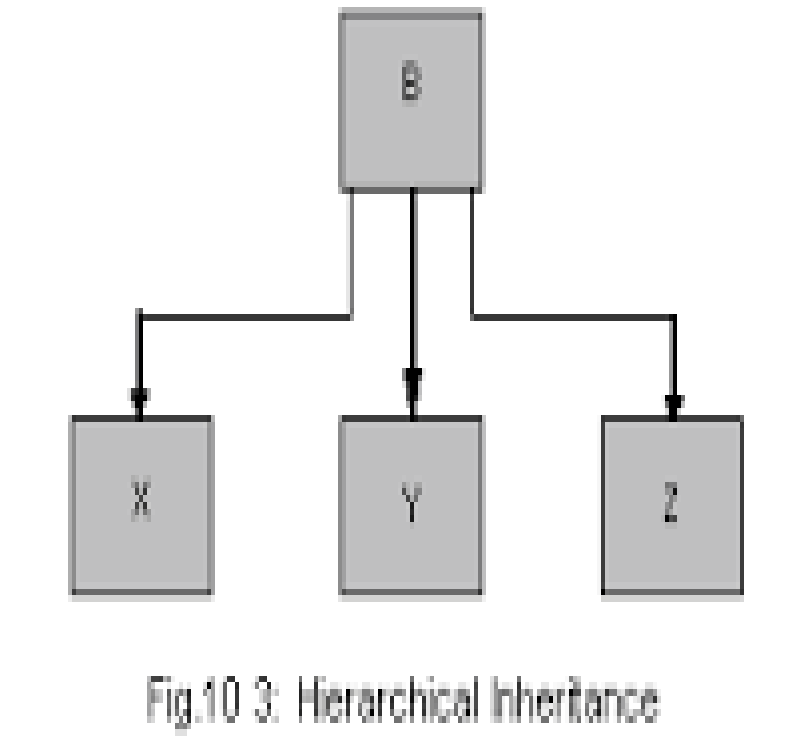
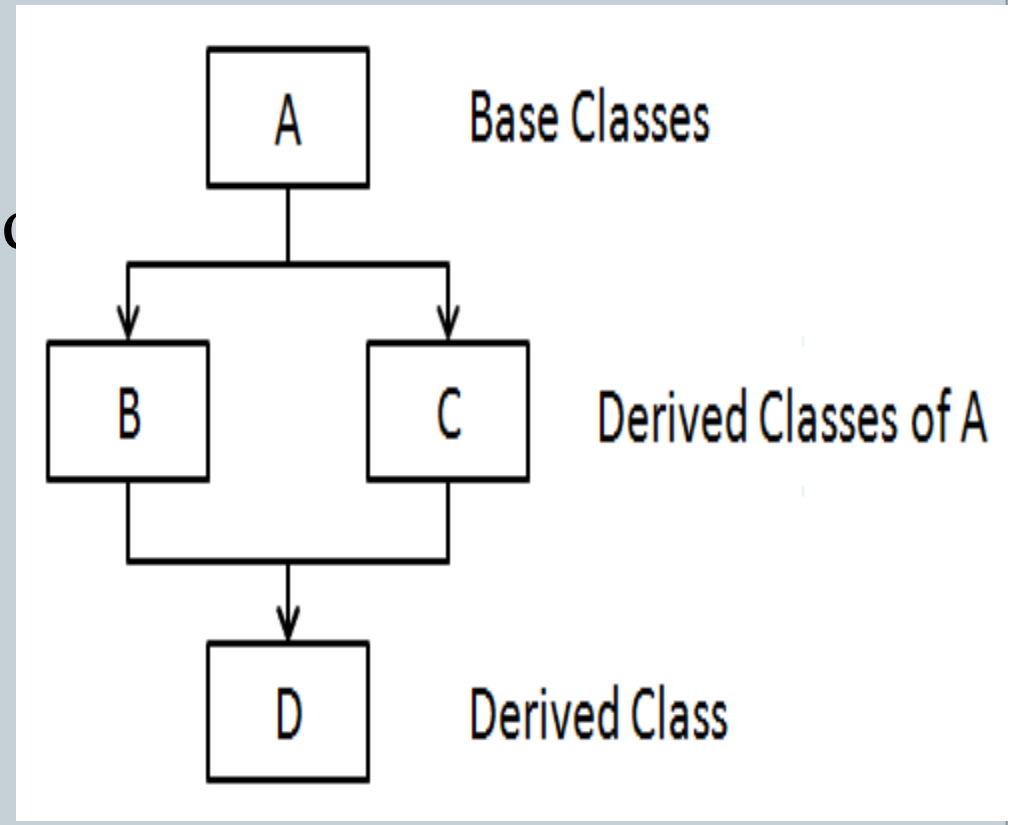


Fig.10 3: Hierarchical Inheritance

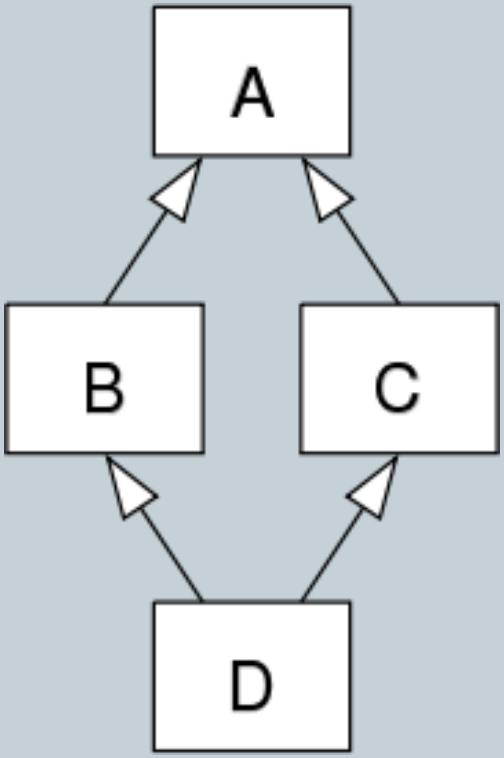
Hybrid Inheritance

- **"Hybrid Inheritance"** is a method where one or more types of inheritanc are combined together and used.



Diamond Problem

- 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.
- The "**diamond problem**" is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C.
- 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.



Virtual Base 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.

Program on virtual base class


```
#include<iostream>
using namespace std;
class A
{
  public:
     int i;
};

class B : virtual public A
{
  public:
     int j;
};
class C: virtual public 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<<"\n";
  cout<<"Value of j is :"<< ob.j<<"\n";
   cout << "Value of k is :"<< ob.k<<"\n";
  cout << "Sum is : "<< ob.sum <<"\n";
  return 0;
}
```

Abstract Class

- In C++ an *abstract class* is one which defines an interface, but does not necessarily provide implementations for all its member functions.
- An abstract class is meant to be used as the base class from which other classes are derived.
- The derived class is expected to provide implementations for the member functions that are not implemented in the base class.
- A derived class that implements all the missing functionality is called a *concrete class* .

Cont…

- Abstract Class is a class which contains atleast one Pure Virtual function in it.
- Abstract classes are used to provide an Interface for its sub classes.
- Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Why we cant create object of abstract class

- When we create a pure virtual function in Abstract class, we reserve a slot for a function in the VTABLE, but doesn't put any address in that slot. Hence the VTABLE will be incomplete.
- As the VTABLE for Abstract class is incomplete, hence the compiler will not let the creation of object for such class and will display an errror message whenever you try to do so.

Program on abstract class


```
#include <iostream>

using namespace std;
class Base          //Abstract base class
{
 public:
 virtual void show() = 0;          //Pure Virtual Function
};

class Derived:public Base
{
 public:
 void show()
 {
cout << "Implementation of Virtual Function in
Derived class"; }
};
```

```
int main()
{
 Base *b;
 Derived d;
 b = &d;
 b->show();
}
```

Characteristics of abstract classes

- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for Up casting, so that its derived classes can use its interface.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Virtual Functions

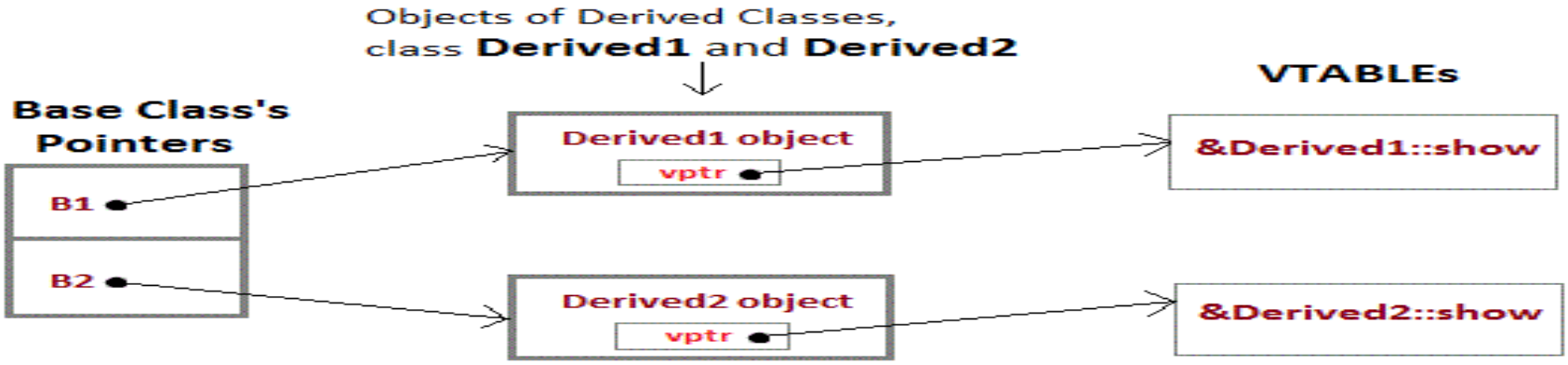
- Virtual Function is a function in base class, which is override in the derived class, and which tells the compiler to perform **Late Binding** on this function.
- Virtual Keyword is used to make a member function of the base class Virtual.
- Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object
- A virtual member function for which no implementation is given is called a *pure virtual function* . If a C++ class contains a pure virtual function, it is an *abstract class*.

Points to be remember

- Only the Base class Method's declaration needs the **Virtual** Keyword, not the definition.
- If a function is declared as **virtual** in the base class, it will be virtual in all its derived classes.
- The address of the virtual Function is placed in the **VTABLE** and the compiler uses **VPTR**(vpointer) to point to the Virtual Function.

Cont…

- To accomplish late binding, Compiler creates **VTABLEs**, for each class with virtual function. The address of virtual functions is inserted into these tables. Whenever an object of such class is created the compiler secretly inserts a pointer called **vpointer**, pointing to VTABLE for that object. Hence when function is called, compiler is able to resolve the call by binding the correct function using the vpointer**.**



vptr, is the vpointer, which points to the Virtual Function for that object.

VTABLE, is the table containing address of Virtual Functions of each class.

Without Virtual functions


```
#include<iostream>
using namespace std;
Class  A
{
 public:
 void show()
 {
  cout << "Base class";
 }
};
```

```
class B:public A
{
 public:
 void show()
 {
  cout << "Derived Class";
 }
};

int main()
{
  A * a;       //Base class pointer
 B b;     //Derived class object
 a= &b;
 a->show();     //Early Binding Ocuurs
}
```

With virtual function


```
#include<iostream>
using namespace std;

class A
{
   public:
   virtual void show()
   {
      cout << "Base class\n";
   }
};

class B: public A
{
private:
   virtual void show()
   {
      cout << "Derived class\n";
   }
};

int main()
{
   A *a;
   B b;
   a = &b;
   a -> show();
}
```

Pure Virtual Functions

- Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with= 0.
- Pure Virtual functions can be given a small definition in the Abstract class, which you want all the derived classes to have. Still you cannot create object of Abstract class.
- Also, the Pure Virtual function must be defined outside the class definition. If you will define it inside the class definition, complier will give an error. Inline pure virtual definition is Illegal.

Program on pure virtual function


```
#include <iostream>

using namespace std;

class Base          //Abstract base class
{
 public:
 virtual void show() = 0;          //Pure Virtual Function
};

void Base :: show()        //Pure Virtual definition
{
 cout << "Pure Virtual definition\n";
}
```

```
class Derived:public Base
{
 public:
 void show()
 { cout << "Implementation of Vi
Function in Derived class"; }
};

int main()
{
 Base *b;
 Derived d;
 b = &d;
 b->show();
}
```

Comparison between virtual and pure virtual functions

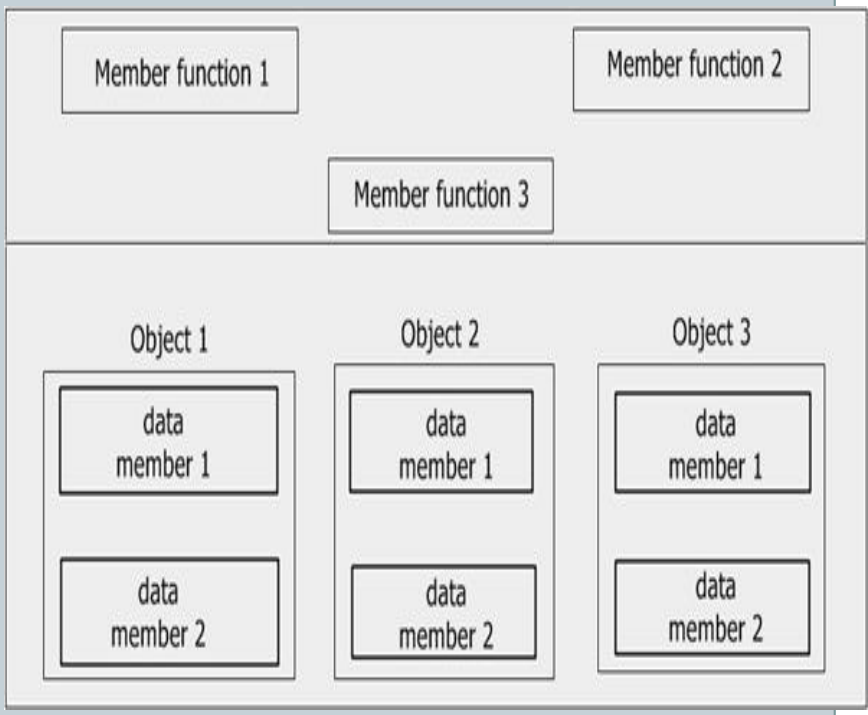
Virtual functions	Pure Virtual functions
• Virtual function have a function body. • Overloaded can be done by the virtual funciton. (Optional) • It is define as : virtual int myfunction();	• Pure virtual function have no function body. • Overloading is must in pure virtual funciton. (Must) • It is define as : virtual int myfunction() = 0;

Pointer to object

- The pointers pointing to objects are referred to as Object Pointers.
- Just like other pointers, the object pointers are declared by placing in front of a object pointer's name. It takes the following general form :
- class-name * object-pointer
- When accessing members of a class using an object pointer, the arrow operator (->) is used instead of dot operator.

This pointer

- As soon as you define a class, the member functions are created and placed in the memory space only once.
- That is, only one copy of member functions is maintained that is shared by all the objects of the class.
- Only space for data members is allocated separately for each object



This pointer

- When a member function is called, it is automatically passed an implicit (in-built) argument that is a pointer to the object that invoked the function. This pointer is called ***this*.**

Consider a situation

- The this pointer can be thought of analogous to the ATM card.
- For instance, in a bank there are many accounts.
- The account holders can withdraw amount or view their bank-statements through Automatic-Teller-Machines.
- Now, these ATMs can withdraw from any account in the bank, but which account are they supposed to work upon ?
- This is resolved by the ATM card, which gives the identification of user and his accounts, from where the amount is withdrawn.
- Similarly, the this pointer is the ATM cards for objects, which identifies the currently-calling object.
- The this pointer stores the address of currently-calling object.

Early binding vs late binding

Early binding	Late binding
• Compiler knows at compile time which function to invoke. • Direct function calls can be resolved using a process known as early binding.	• Compiler doesn't know until runtime which function to invoke. • one way to get late binding is to use function pointers or the other way is the use of virtual functions in inheritance