Object Oriented Programming

5 Inheritance

-

What is Inheritance in OOP? (1/3)

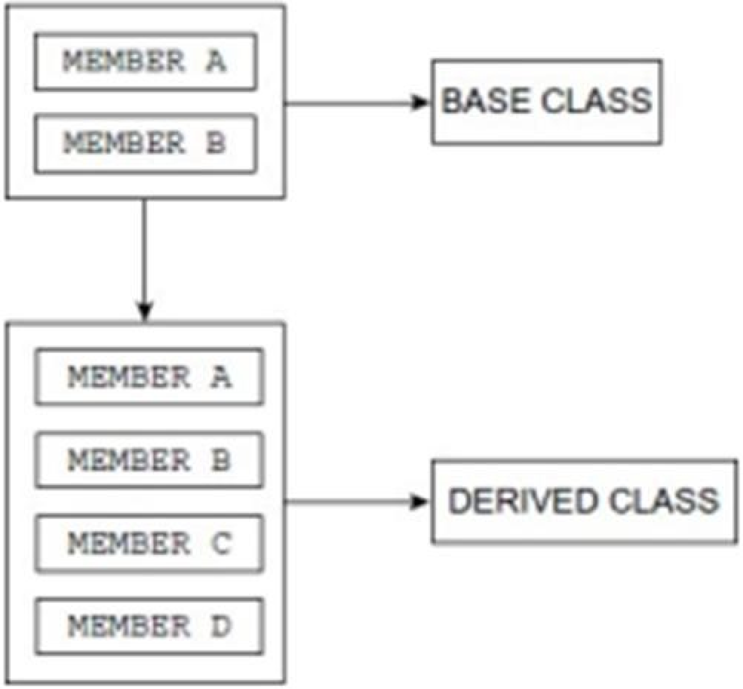
- Inheritance is one of the most useful and essential characteristics of object-oriented programming.

- The existing classes are the main components of inheritance.

- The new classes are created from **existing ones** =>The properties of the **existing classes** are simply *extended* to the new classes.

What is Inheritance in OOP? (2/3)

- The new classes created by using such a method are known as derived classes, and the existing classes are known as base classes, as shown in the figure.



What is Inheritance in OOP? (3/3)

- The base class is also called super class, parent, or ancestor.
- The derived class is called subclass, child, or descendent.
- It is also possible to derive a class from a previously derived class. A class can be derived from more than one class.

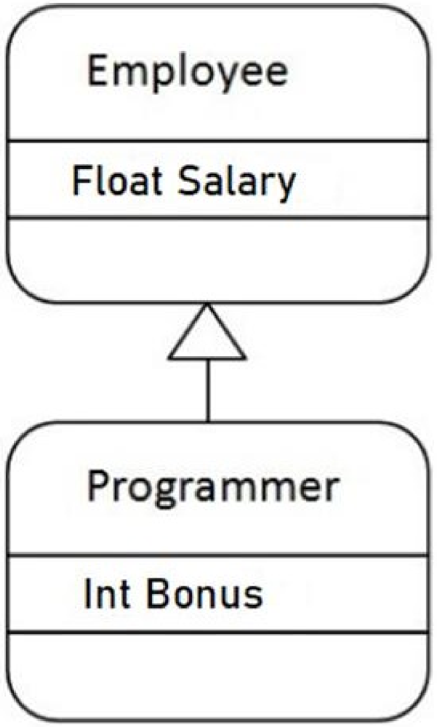
Inheritance and Reusability

- **Inheritance Definition:** The procedure of creating a new class from one or more existing classes is termed inheritance.

- **Reusability:** Reusability means the reuse of properties of the **base class** in the **derived classes**. Reusability is achieved using inheritance.

- Inheritance and reusability are not different from each other. The outcome of inheritance is reusability.

Inheritance and is-a Relationship



- As displayed in the figure, Programmer is the derived class (subclass) and Employee is the **base class** (superclass).
- The relationship between the two classes is Programmer IS-A Employee.

=> It means that Programmer is a type of Employee.

Defining Derived Classes (1/4)

- The derived class is indicated by associating with the base class.
- A new class (derived class) has, also, its own set of member variables and functions. The syntax given below creates the derived class:

Definition Syntax in C++ :

```
class name_of_the_derived_class: access specifiers name_of_the_base_class
{
 // member variables of new class (derived class)
}
```

Defining Derived Classes (2/4): Example in C++

```
class Employee{
float salary=40000;
};

class Programmer: Employee{
int bonus=10000;};

int main(){
Programmer p;
cout<<"Programmer salary is:"<<p.salary<<endl;
cout<<"Bonus of Programmer is:"<<p.bonus<<endl;
return 0;
}
```

Defining Derived Classes (3/4)

- The access specifier or the visibility mode is optional and, if present, may be public, private or protected.

- By default it is private. Visibility mode describes the status of derived features.

Defining Derived Classes (4/4)

Example 1:

```
class Programmer: public Employee
{
// Members of class Programmer
};
```

Example 2:

```
class Programmer: private Employee
{
// members of class Programmer };
```

Example 3:

```
class Programmer: Employee // by default private derivation
{
// members of class Programmer
};
```

Example 4:

```
class Programmer: protected Employee
{
// members of class Programmer
};
```

Important Notes (1/2)

- When a public access specifier is used, the public members of the base class are public members of the derived class. Similarly, the protected members of the base class are protected members of the derived class.

- When a private access specifier is used, the public and protected members of the base class are the private members of the derived class.

Important Notes (2/2)

- In the inheritance, some of the base class data elements and member functions are inherited into the derived class. We can add our own data and member functions and thus extend the functionality of the base class.

- Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

Public Inheritance (1/3)

- When a class is derived publicly, all the public members of the base class can be accessed directly in the derived class.

```
class A
{ public:
 int x;
};
class B: public A
{  public:
 int y;
};
```

```
int main()
{  B b;
 b.x=20;
 b.y=30;
 cout<<“\n member of
A:”<<b.x;
 cout<<“\n Member of
B:”<<b.y;
 return 0;
}
```

Output:

```
Member of A : 20
Member of B : 30
```

Public Inheritance (2/3)

- In case the base class has private member variables and a class derived publicly, the derived class can access the member variables of the base class using only member functions of the base class.

- The public derivation does not allow the derived class to access the private member variable of the class directly as is possible for public member variables.

Public Inheritance (3/3)

```
class A
{
private:
 int x;
public:
 A() {x=20;}
 void showx()
 {
 cout<<“\n x=”<<x;
 }
};
```

```
class B : public A
{
 public:
 int y;
 B() {y=30;}
 void showy()
 {
 showx();
 cout<<“\n y=”<<y;
 }
};
```

```
int main()
{
 B b;  b.showy();
 return 0;
}
```

Private Inheritance (1/2)

- The objects of the privately derived class cannot access the public members of the base class directly. Hence, the member functions are used to access the members.
- *Example 1:*

```
class A
{
 public:
 int x;
};
```

```
class B : private A
{  public:
 int y;
 B()
 {  x=20;  y=40;  }
 void show()
 {
 cout<<“\n x=”<<x;
 cout<<“\n y=”<<y;
 } };
```

```
int main()
{
 B b;
b.show();
 return 0;
}
```

Private Inheritance (2/2)

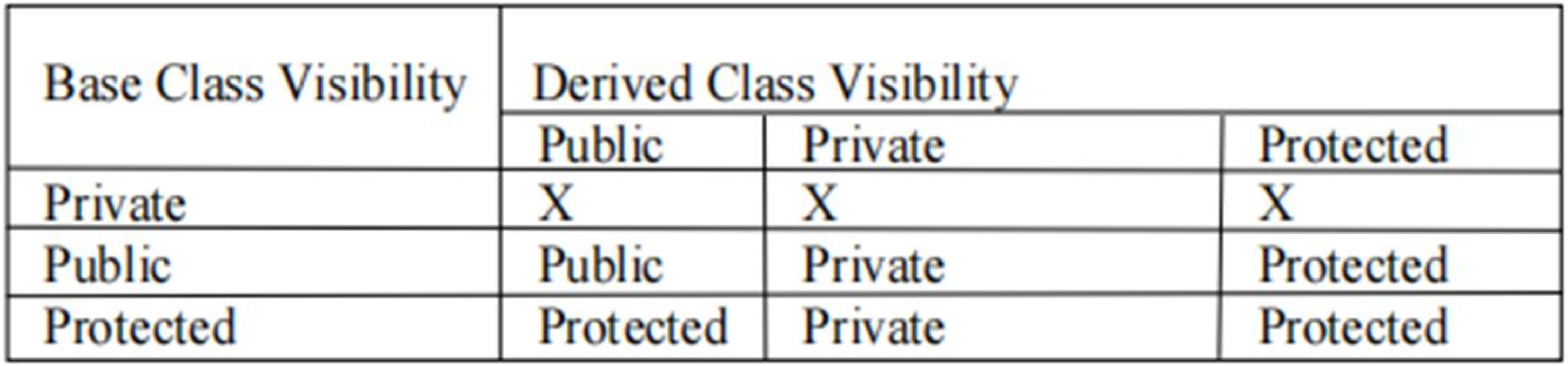
- *Example 2:*

```
class A {
 int x;
 public:
 A()
 {  x=20;  }
 void showx()
 {  cout<<“\n x=”<<x;
 }
};
```

```
class B : private A
{
 public:
 int y;
 B()  {  y=40;
void showy()
 {
 showx();
 cout<<“\n y=”<<y;
 } };
```

```
int main()
{
 B b;
 b.showy();
 return 0;
}
```

Access Specifiers and their Scope (1/4)



Base Class Visibility	Derived Class Visibility		
	Public	Private	Protected
Private	X	X	X
Public	Public	Private	Protected
Protected	Protected	Private	Protected

Access Specifiers and their Scope (2/4)

1. All private members of the class are accessible to public members of the same class. They cannot be inherited.

2. The derived class can access the private members of the base class using the member function of the base class.

3. All the protected members of the class are available to its derived classes and can be accessed without the use of the member function of the base class.

Access Specifiers and their Scope (3/4)

4. If any class is prepared for deriving classes, it is advisable to declare all members of the base class as protected, so that derived classes can access the members directly.

5. All the public members of the class are accessible to its derived class. There is no restriction for accessing elements.

Access Specifiers and their Scope (4/4)

6. The access specifier required while deriving class is either private or public. If not specified, private is default.

7. *Constructors* and *destructors* are declared in the public section of the class. If declared in the private section, the object declared will not be initialized and the compiler will flag an error.

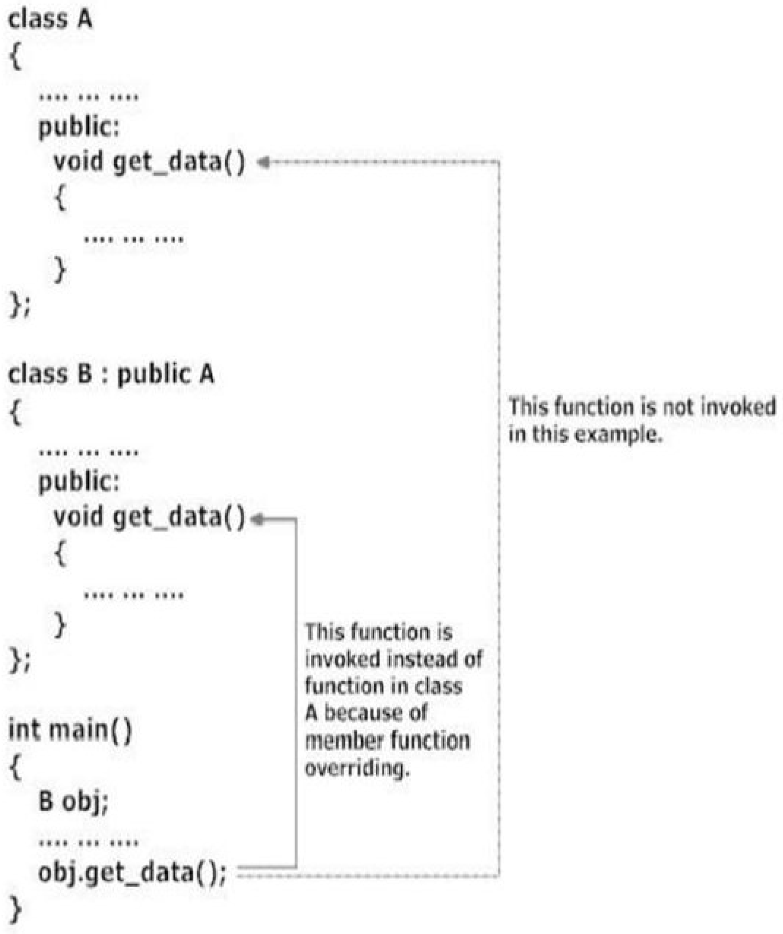
Constructor Chaining

- In C++, the base class constructor is called first to initialize base class members, followed by the derived class constructor. Upon destruction, the derived class destructor is called first to clean up resources specific to the derived class, then the base class destructor is invoked to handle base class cleanup.

- This behavior is **standard** in **object-oriented programming languages** like C++, ensuring proper initialization and cleanup in inheritance hierarchies.

Function Overriding

- Base class and derived class have member functions with same name and arguments.
- if we create an object of derived class and write code to access that member function => the member function in derived class is only invoked => the member function of derived class overrides the member function of base class.
- This feature in C++ programming is known as *function overriding.*



```
class A
{
  .... ... ....
  public:
    void get_data()
    {
      .... ... ....
    }
};

class B : public A
{
  .... ... ....
  public:
    void get_data()
    {
      .... ... ....
    }
};

int main()
{
  B obj;
  .... ... ....
  obj.get_data();
}
```

This function is not invoked in this example.

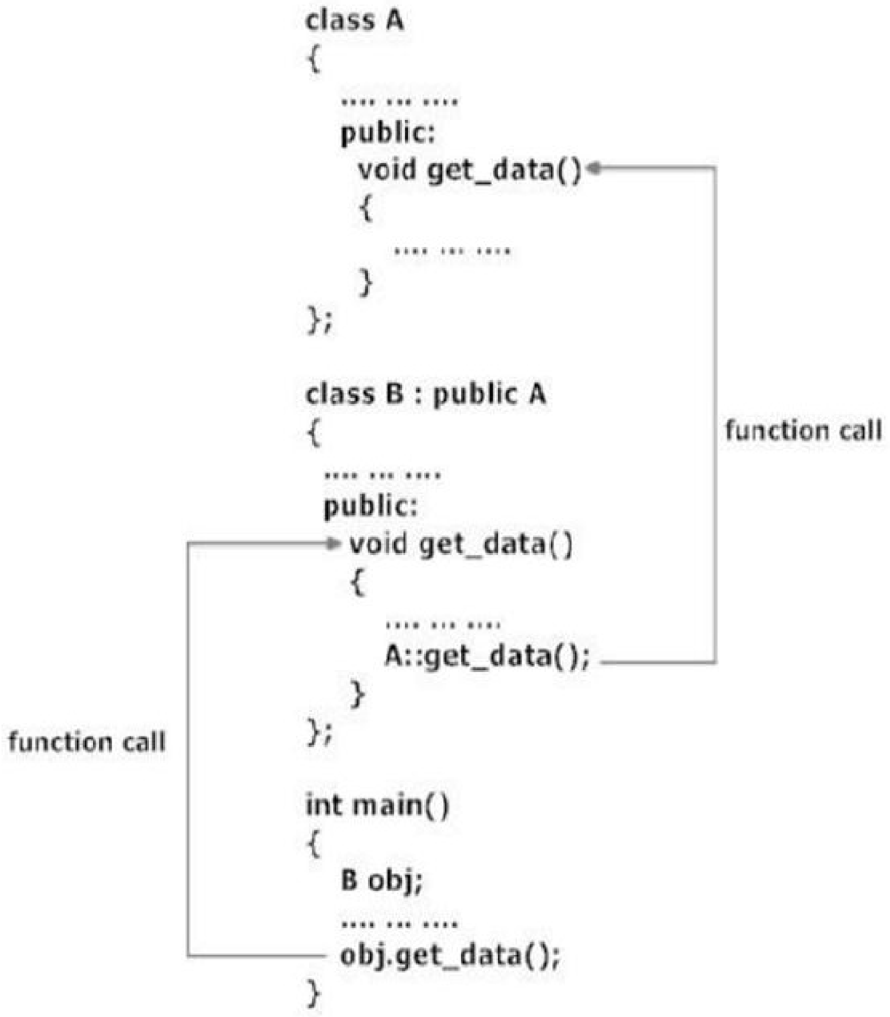
This function is invoked instead of function in class A because of member function overriding.

Figure: Member Function Overriding in C++

Accessing the Overridden Function in Base Class From Derived Class

- To access the overridden function of base class from derived class, scope resolution operator :: is used.

- For example, if the name of class is not specified, the compiler thinks get_data() function is calling itself.



```
class A
{
   .... ... ....
   public:
    void get_data()
    {
         .... ... ....
    }
};

class B : public A
{
  .... ... ....
  public:
   void get_data()
   {
       .... ... ....
       A::get_data();
   }
};

int main()
{
   B obj;
   .... ... ....
   obj.get_data();
}
```

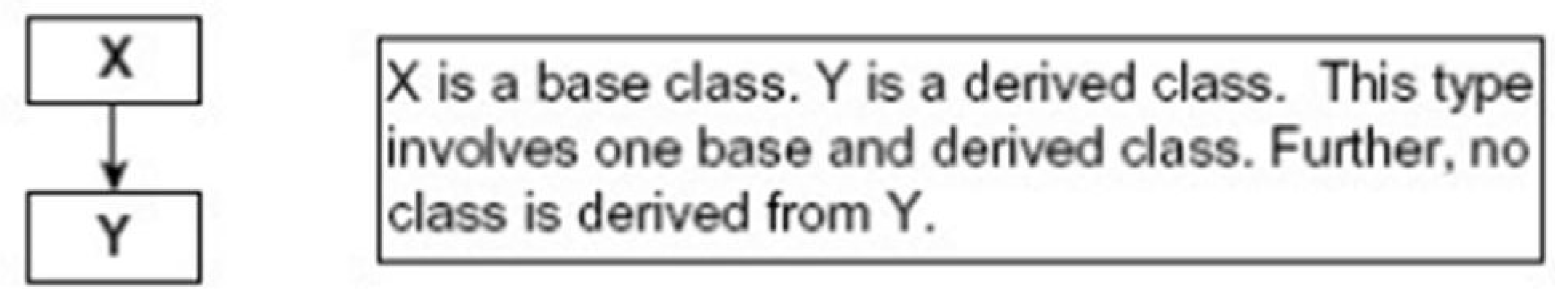
Types of Inheritance

The Inheritance is classified as follows:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance
- Multi-path Inheritance

Single Inheritance (Definition)

- This occurs when only one base class is used for the derivation of a derived class.
- Further, derived class is not used as a base class, such a type of inheritance that has one base and derived class is known as single inheritance.



X is a base class. Y is a derived class. This type involves one base and derived class. Further, no class is derived from Y.

Single Inheritance (Example 1)

```
class Publisher
{ string pname;
 string place;
public:
void getdata()
 {  cout<<"Enter name
and place of
publisher:"<<endl;
 cin>>pname>>place;
 }
void show()
 {  cout<<"Publisher
Name:"<<pname<<endl;
cout<<"Place:"<<place<
<endl; } };
```

```
class Book:public
Publisher
{  string title;
float price;int pages;
public:
void getdata()
 {  Publisher::getdata();
 cout<<"Enter Book Title,
Price and No. of
pages"<<endl;
 in>>title>>price>>pages;
 }
```

```
void show()
 {  Publisher:: show ();
cout<<"Title:"<<title<<e
ndl;
cout<<"Price:"<<price<
<endl;
 cout<<"No. of
Pages:"<<pages<<endl;
 } };
//Main Function
int main() {
 Book b;
 b.getdata();
 b.show(); return 0; }
```

Single Inheritance (Example 2)

```
class Animal{
public: void
eat(){cout<<"eating..."<<endl;}
};

class Dog: public Animal{
void
bark(){cout<<"barking..."<<endl;
}
};
```

```
//Main Function
int main() {
Dog d;
d.bark();
d.eat();
return 0;
}
```

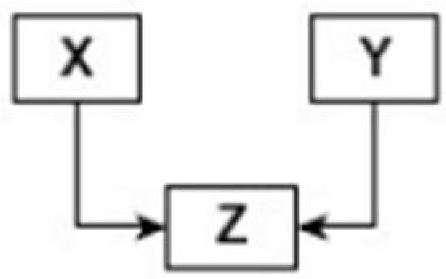
Output:

```
barking...
eating...
```

Multiple Inheritance (Definition)

5 Inheritance

- When two or more base classes are used for the derivation of a class, it is called multiple inheritance.



X and Y are base classes. Z is a derived class. Class Z inherits properties of both X and Y. Further, Z is not used as a base class.

Multiple Inheritance (Example 1/2)

```
class Publisher
{ string pname;
 string place;
public:
void getdata()
 {  cout<<"Enter name
and place of
publisher:"<<endl;
 cin>>pname>>place;
 }
void show()
 {  cout<<"Publisher
Name:"<<pname<<endl;
cout<<"Place:"<<place<
<endl; } };
```

```
class Author
{
 string aname;
public:
void getdata()
 {
 cout<<"Enter Author
name:"<<endl;
 cin>>aname;
 }
void show ()
 { cout<<"Author
Name:"<<aname<<endl;
 } };
```

Multiple Inheritance (Example 2/2)

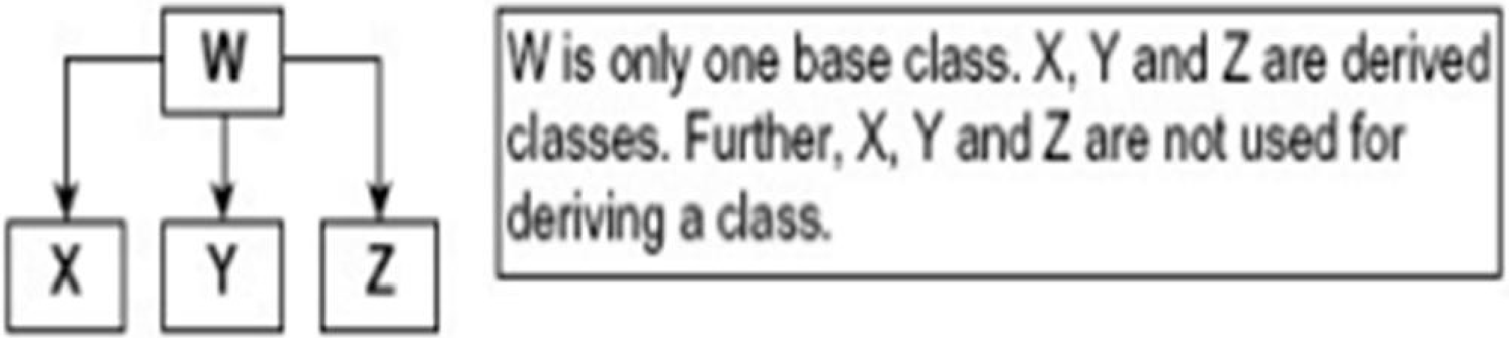
```
class Book:public
Publisher, public Author
{  string title;
float price;
int pages;
public:
void getdata()
 { Publisher::getdata();
 Author::getdata();
 cout<<"Enter Book
Title, Price and No. of
pages"<<endl;
in>>title>>price>>pages
;
 }
```

```
void show()
 {
 Publisher:: show ();
 Author:: show ();
cout<<"Title:"<<title<<en
dl;
cout<<"Price:"<<price<<
endl;
 cout<<"No. of
Pages:"<<pages<<endl;
 }
};
```

```
int main() {
 Book b;
 b.getdata();
b.show();
return 0;
}
```

Hierarchical Inheritance (Definition)

- When a single base class is used for the derivation of two or more classes, it is known as hierarchical inheritance.



W is only one base class. X, Y and Z are derived classes. Further, X, Y and Z are not used for deriving a class.

Hierarchical Inheritance (Example 1/2)

```
class Account
{ int act_no;
 string cust_name;
public:
void getdata()
 {  cout<<"Enter Accout number
and Customer name:"<<endl;
cin>>act_no>>cust_name;
 }
void show ()
 {  cout<<"Account
Number:"<<act_no<<endl;
 cout<<"Customer
Name:"<<cust_name<<endl; }};
```

```
class SB_Act: public Account
{ float roi;
 public:
void getdata()
 { Account::getdata();
 cout<<"Enter Rate of
Interest"<<endl;
 cin>>roi;
 }
void show ()
 {
 Account:: show ();
 cout<<"Rate of
Interest:"<<roi<<endl; } };
```

Hierarchical Inheritance (Example 2/2)

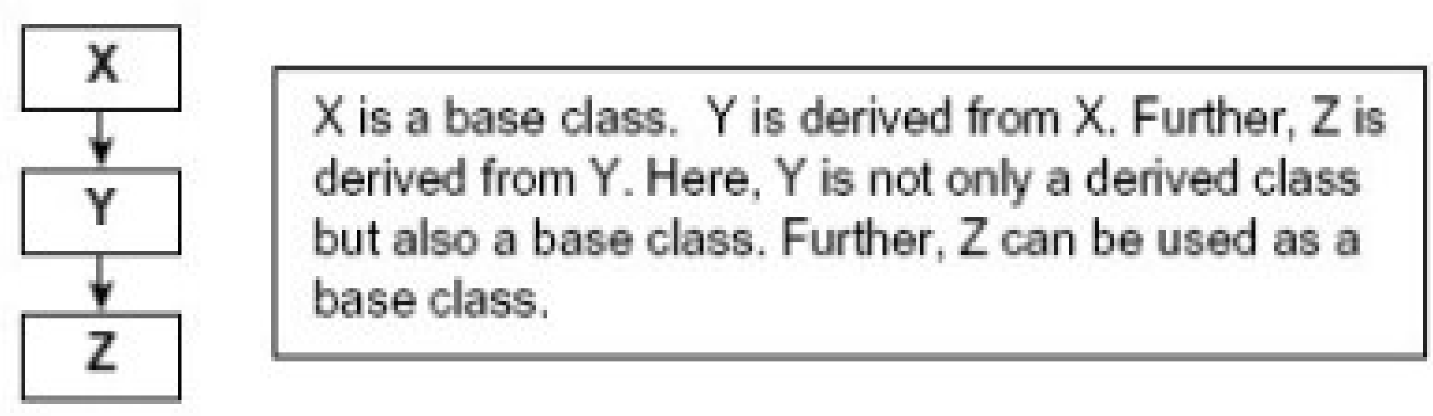
```
class Current_Act: public
Account
{ float roi;
public:
void getdata()
 { Account::getdata();
 cout<<"Enter Rate of
Interest"<<endl;
 cin>>roi;
 }
void show ()
 {  Account:: show ();
 cout<<"Rate of
Interest:"<<roi<<endl;  } };
```

```
int main() {
 SB_Act s;
 s.getdata();
 s. show ();
  Current_Act c;
 c.getdata();
 c. show ();
return 0;
}
```

5 Inheritance

Multilevel Inheritance (Definition)

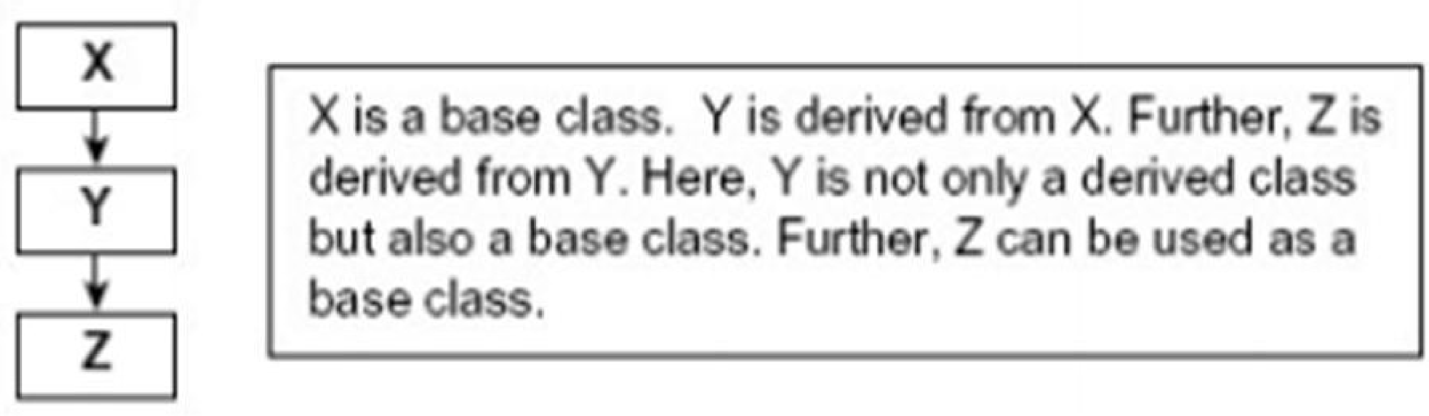
- When a class is derived from another derived class, that is, the derived class acts as a base class, such a type of inheritance is known as multilevel inheritance.



X is a base class. Y is derived from X. Further, Z is derived from Y. Here, Y is not only a derived class but also a base class. Further, Z can be used as a base class.

Multilevel Inheritance (Definition)

- When a class is derived from another derived class, that is, the derived class acts as a base class, such a type of inheritance is known as multilevel inheritance.



X is a base class. Y is derived from X. Further, Z is derived from Y. Here, Y is not only a derived class but also a base class. Further, Z can be used as a base class.

Multilevel Inheritance (Example 1/2)

```
class Publisher
{  string pname;
 string place;
public:
void getdata()
 {  cout<<"Enter name and place of publisher:"<<endl;
 cin>>pname>>place;
 }
void show ()
 {  cout<<"Publisher Name:"<<pname<<endl;
 cout<<"Place:"<<place<<endl;
 } };
```

```
class Author:public Publisher
{  string aname;
public:
void getdata()
 {  Publisher::getdata();
 cout<<"Enter Author name:"<<endl;
 cin>>aname;
 }
void show ()
 {  Publisher:: show ();
 cout<<"Author Name:"<<aname<<endl;
 }};
```

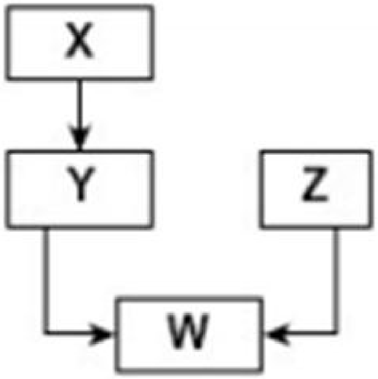
Multilevel Inheritance (Example 2/2)

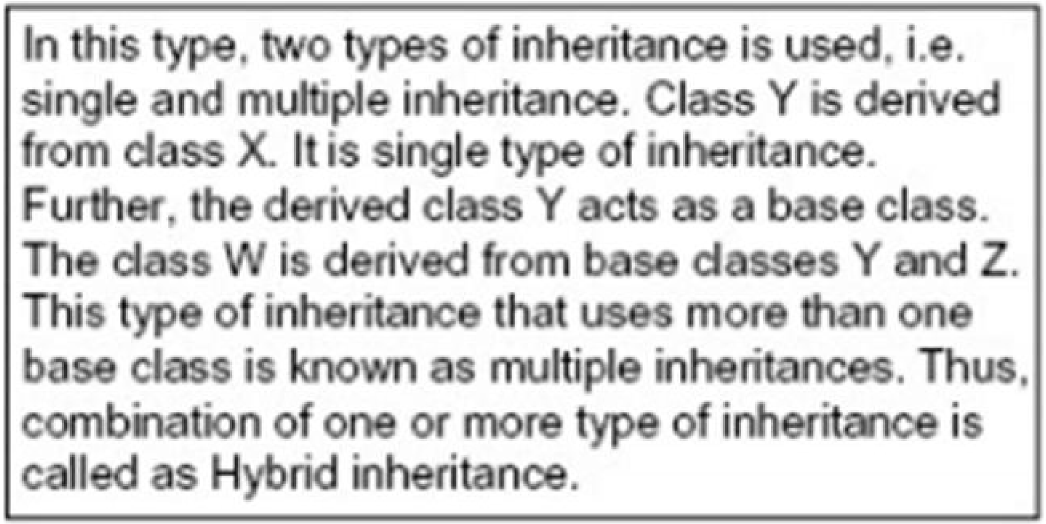
```
class Book:public Author
{  string title;
float price;
int pages;
public:
void getdata()
 {  Author::getdata();
 cout<<"Enter Book Title, Price
and No. of pages"<<endl;
 cin>>title>>price>>pages;
 }
```

```
void show()
 {  Author:: show ();
 cout<<"Title:"<<title<<endl;
 cout<<"Price:"<<price<<endl;
 cout<<"No. of
Pages:"<<pages<<endl;
 }
};
//Main Function
int main() {
 Book b;
 b.getdata();
 b.show();
return 0;
}
```

Hybrid Inheritance (Definition)

- A combination of one or more types of inheritance is known as hybrid inheritance.





In this type, two types of inheritance is used, i.e. single and multiple inheritance. Class Y is derived from class X. It is single type of inheritance. Further, the derived class Y acts as a base class. The class W is derived from base classes Y and Z. This type of inheritance that uses more than one base class is known as multiple inheritances. Thus, combination of one or more type of inheritance is called as Hybrid inheritance.

Hybrid Inheritance (Example 1/3)

```
class Publisher
{  string pname;
 string place;
public:
void getdata()
 {  cout<<"Enter name and place
of publisher:"<<endl;
 cin>>pname>>place;
 }
void show ()
 {  cout<<"Publisher
Name:"<<pname<<endl;
 cout<<"Place:"<<place<<endl;
 } };
```

```
class Author:public Publisher
{  string aname;
public:
void getdata()
 {  Publisher::getdata();
 cout<<"Enter Author
name:"<<endl;
 cin>>aname;
 }
void show ()
 {  Publisher:: show ();
 cout<<"Author
Name:"<<aname<<endl;
 }};
```

Hybrid Inheritance (Example 2/3)

```
class Distributor
{  string dname;
public:
void getdata()
 {  cout<<"Enter Distributor
name:"<<endl;
cin>>dname;
 }
void show ()
 {  cout<<"Distributor
Name:"<<dname<<endl;
 } };
```

```
class Book:public Author, public
Distributor
{
 string title;
float price;
int pages;
public:
void getdata()
 {
 Author::getdata();
 Distributor::getdata();
 cout<<"Enter Book Title, Price
and No. of pages"<<endl;
 cin>>title>>price>>pages;  }
```

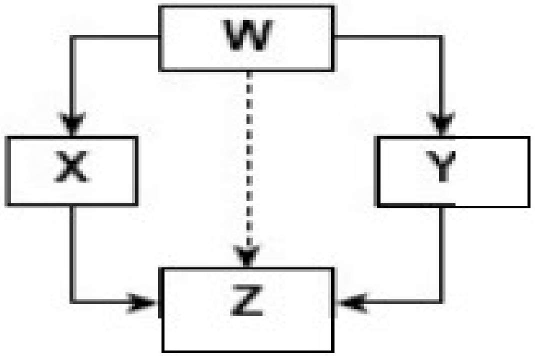
Hybrid Inheritance (Example 3/3)

```
// class Book:public Author,
public Distributor
void show()
 {
 Author:: show ();
 Distributor:: show ();
 cout<<"Title:"<<title<<endl;
 cout<<"Price:"<<price<<endl;
 cout<<"No. of
Pages:"<<pages<<endl;
 }
};
```

```
int main() {
 Book b;
 b.getdata();
 b.show();
return 0;
}
```

Multipath Inheritance (Definition)

- When a class is derived from **two or more classes**, those are derived from the same base class. Such a type of inheritance is known as multipath inheritance.
- Multipath inheritance also consists of many types of inheritance, such as multiple, multilevel, and hierarchical, as shown in the figure.



=> Its disadvantage is the ambiguity in classes.

Multipath Inheritance (Example 1/2)

```
class A1
{
 protected:
 int a1;
};
class A2 : public A1
{
 protected:
 int a2;
};
```

```
class A3: public A1
{
 protected:
 int a3;
};
class A4: public A2,A3
{
 int a4;
};
```

Multipath Inheritance (Example 2/2)

- In the above example, classes **A2** and **A3** are derived from class **A1**; that is, their base class is similar to class **A1** (hierarchical inheritance). Both classes **A2** and **A3** can access the variable **a1** of class **A1**. The class **A4** is derived from classes **A2** and **A3** by multiple inheritance. If we try to access the variable **a1** of class **A1**, the compiler shows error.

- To overcome the ambiguity occurring due to multipath inheritance, the C++ provides the keyword **virtual**.

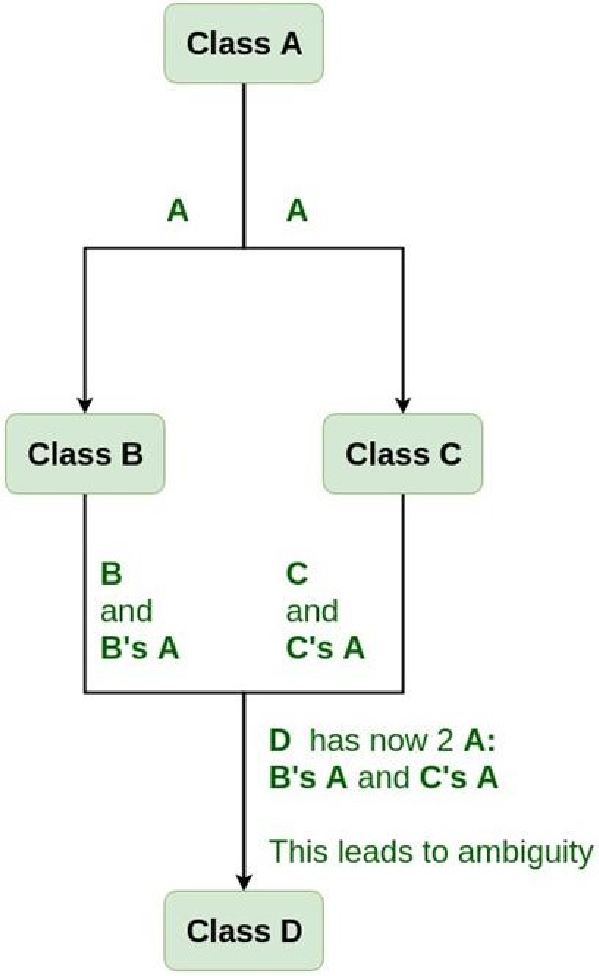
- The keyword virtual declares the specified classes virtual.

Virtual Base Classes (Definition)

- **Virtual base classes** are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.

- *Need for Virtual Base Classes*: Consider the situation where we have one class **A** . This class **A** is inherited by two other classes **B** and **C**. Both these classes are inherited into another in a new class **D** as shown in the ***illustrative example (Next slide)***.

Virtual Base Classes (*Illustrative Example*)



Data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.

Virtual Base Classes (*technical overview*)

```
#include <iostream>
using namespace std;
class A {
public:
void show() {
cout << « I am from class A \n"; }
};
class B : public A { };
class C : public A { };
class D : public B, public C { };

// Main function
int main() {
D obj1;
obj1.show();
}
```

Compile Errors

```
prog.cpp: In function 'int main()':
prog.cpp:29:9: error: request for
member 'show' is ambiguous
object.show();
^
prog.cpp:8:8: note: candidates are:
void A::show()
void show()
^
prog.cpp:8:8: note: void A::show()
```

Virtual Base Classes (Advantage)

To resolve this ambiguity when class **A** is inherited in both class **B** and class **C**, it is declared as **virtual base class** by placing a keyword **virtual** as :

Syntax for Virtual Base Classes:

Syntax 1:

```
class B : virtual public A
{ ……..};
```

Syntax 2:

```
class C : public virtual A
{……};
```

- Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the **virtual base class**.
- Virtual base classes offer a way to **save space** and **avoid ambiguities** in class hierarchies that use multiple inheritances.
- A **single copy** of its data members is shared by all the base classes that use virtual base.

Virtual Base Classes (Advantage)

```
#include <iostream>
using namespace std;
class A {
public:
void show() {
cout << « I am from class A \n"; }
};
class B : public  virtual A { };
class C : public virtual A { };
class D : public B, public C { };

// Main function
int main() {
D obj1;
obj1.show();
}
```

Output

```
I am from class A
```

Why? The class **A** has just one data member **show()** which is **public**. This class is virtually inherited in class **B** and class **C**. Now class **B** and class **C** use the virtual base class **A** and no duplication of data member **show()** is done; Classes **B** and **C** share a single copy of the members in the virtual base class **A**.