TATA ELXSI

OBJECT ORIENTED PROGRAMMING USING C++
Module 6

Learning & Development Team

Inheritance

Objectives

- In this lesson, you will learn to:
- Derive a class from an existing class
- Need for Derived classes
- Implementing Single, Multiple, Multilevel Inheritance
- Role of constructors and destructors in inheritance

Inheritance[**is-a relationship**]

- Reusing already existing class is known as Inheritance.
- A concept does not exist in isolation. It coexists with related concepts and derives much of its power from relationships with related concepts.
- For example, while explaining what a car is, wheels, engines, drivers, pedestrians, trucks, ambulances, roads, oil, tickets, motels etc., come in picture.
- Classes are used to represent concepts.
- How to represent relationship between concepts?
- The notion of derived classes and its associated language mechanisms are provided to express hierarchical relationships between classes.
    - Example: triangle and square have the concept of shape in common.

Inheritance

Derived classes:

```
class Employee
{
   string name;
   Date hiring_data;
   short dept;
   ……..
};

class Manager
{
   Employee emp;
   char *deptName;
   int totalExp;
   ……..
};
```

Inheritance

- A manager is an Employee. The employee details is stored in the emp member of a Manager.
- It is understood by the user, but it does not tell the compiler and other tools that manager is also an employee.
- We cannot put a Manager onto a list of employees without writing special code. (explicit type conversion) or put the address of emp member of Manager onto a list of employees.
- Not a proper solution.
- Right approach is to state that Manager **is an** Employee.

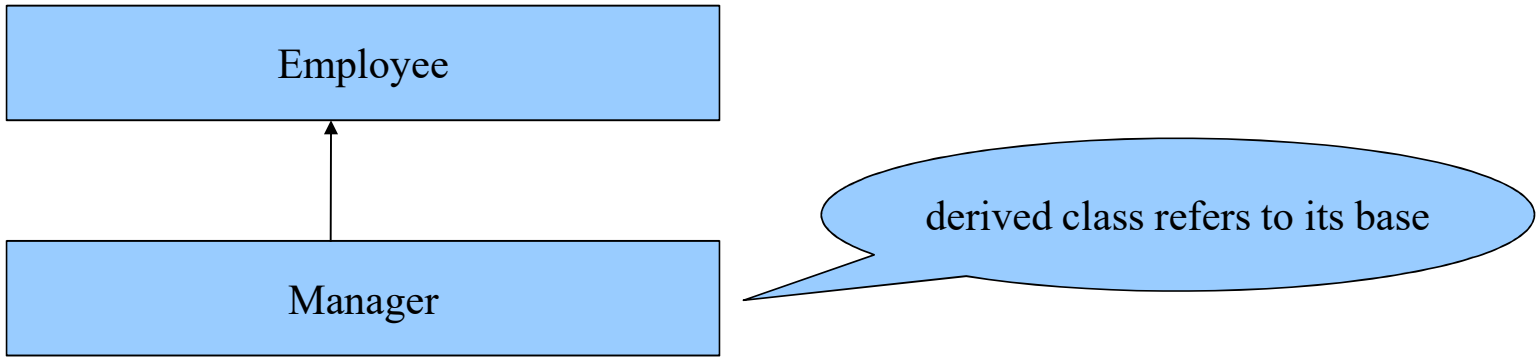
Code Syntax for Inheritance

- Class inheritance uses this general form:
  - class derived-class-name : **access** base-class-name
- The access status of the base class members inside the derived class is determined by **access**
- The base class access specifier must either be **public, private, or protected.**
- If no access specifier is present, the access specifier is private by default.

Inheritance

```
class Manager: public Employee
{
   char *deptName;
   int totalExp;
   ........
};
```

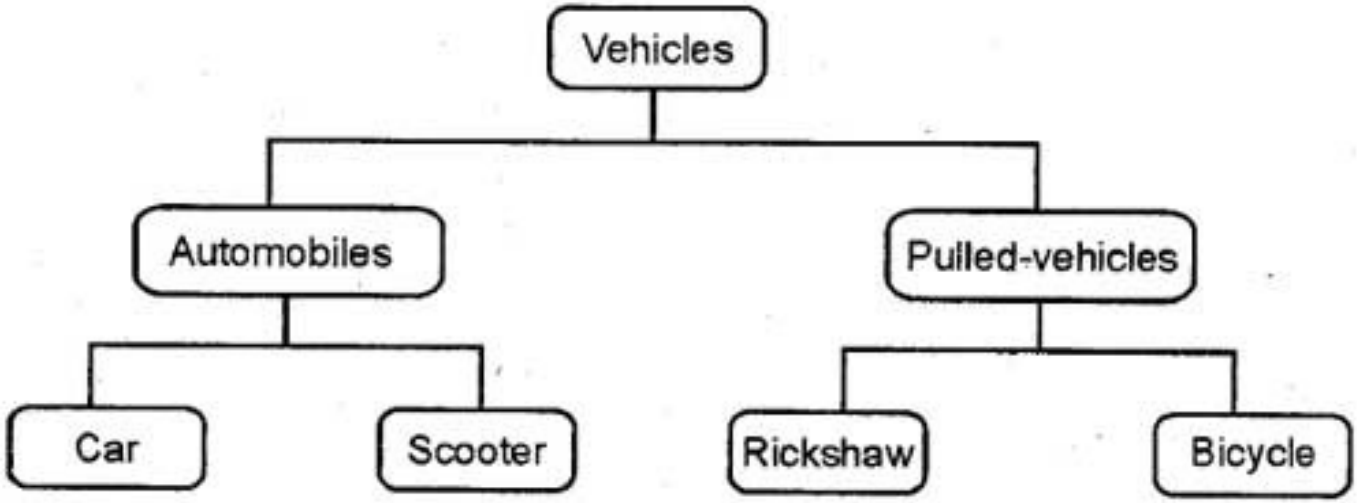
- Employee is the base class for manager. Or in other words, Manager is derived from Employee.
- The class Manager has members of class Employee + its own members.



Inheritance

- A derived class inherits properties from its base, and its relationship is called inheritance.
- Base class is also called super class and derived class is called sub class.
- Derived class is a superset of the data of an object of its base class. It is larger than its base class.
- Deriving Manager from Employee in this way makes Manager a subtype of Employee so that Manager can be used wherever an Employee is acceptable.
  - We can create a list of employees who are managers.
    (Manager * can be used as Employee *)
  - However Employee * cannot be used as a Manager *
    (needs explicit type conversion).

Example of Inheritance

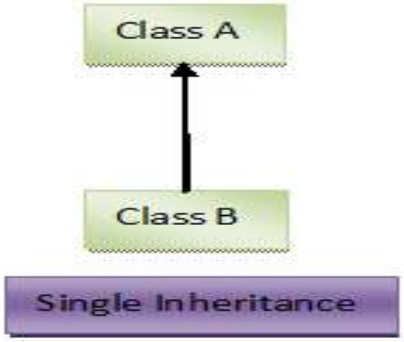


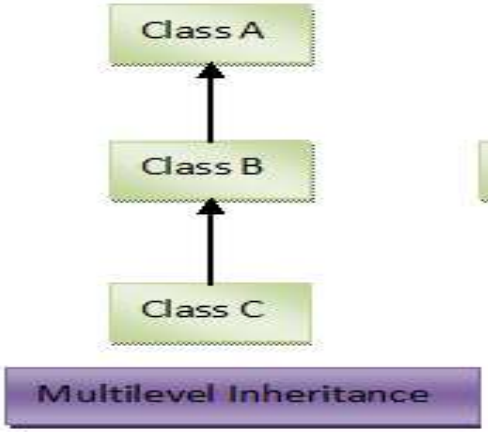
Inheritance

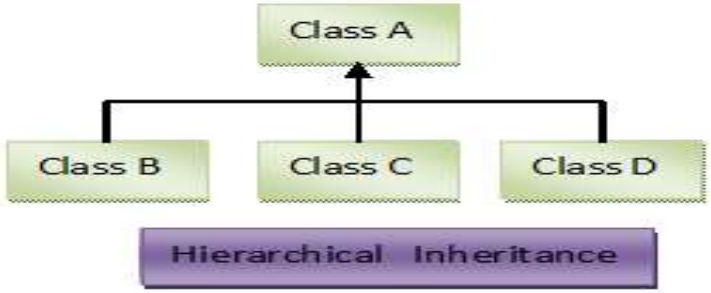
Generalization/Specialization

- In keeping with C++ terminology, a class that is inherited is referred to as a base class. The class that does the inheriting is referred to as the derived class.
- Each instance of a derived class includes all the members of the base class. The derived class inherits all the properties of the base class.
- Therefore, the derived class has a large set of properties than its base class. However, a derived class may override some of the properties of the base class.

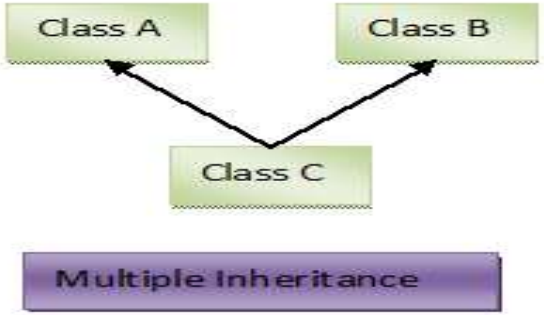
Implementing Single, Multiple, Multilevel, Hierarchical & Hybrid Inheritance.

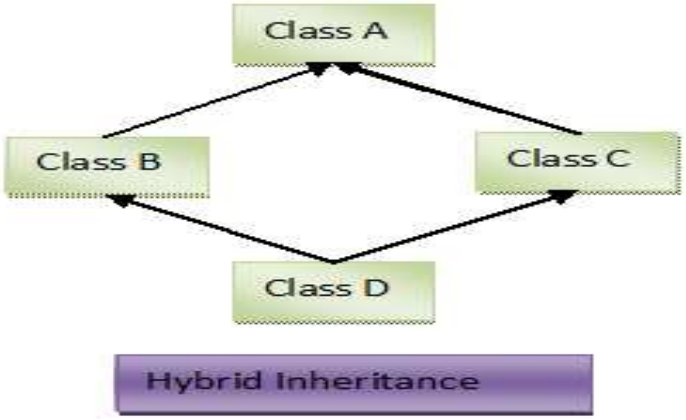






* *Arrows should always be pointing towards Base class*





Inheritance types

- Single Inheritance
  - Class inherits from one base class.

- Multiple Inheritance
  - Class inherits from multiple base classes.

- Three types of inheritance:
  - public: Derived objects can access the base class members.
  - private: Derived objects cannot access the base class members.
  - protected: Derived classes and friends can access protected members of the base class.

Defining Derived classes

- When a class inherits another, the members of the base class become members of the derived class.

```
class derived_class_name : access base_class_name
{
   // body of class
};
```

- Access status of the base class members inside the derived class is determined by access.
- The access specifier can be : public, private or protected.

public access

- When the access specifier for a base class is public, all public members of the base class become public members of the derived class
- All protected members of the base class become protected members of the derived class.
- The private elements of the base class remains private to the base and not accessible by the derived class.

public access

```
class base
{
   int i,j;
  public:
   void set (int a, int b)
   {
      i = a ;
      j = b;
   }
   void show()
   {
      cout << i << j;
   }
};
```

```
class derived:public base
{
   int k;
  public:
   derived(int x)
   {k = x;}
   void showk()
   { cout << k; }
};
int main()
{
   derived ob(3);
   ob.set(1,2);
   ob.show();
   ob.showk();
   return 0;
}
```

Private access

- When the base class is inherited using the private access specifier, all public and protected members of the base class become private members of the derived class.

Private access

```
class base
{
  int i,j;
 public:
  void set (int a, int b)
  {
    i = a;
    j = b;
  }
  void show()
  {
    cout << i << j;
  }
};
```

```
class derived:private base
{
  int k;
 public:
  derived(int x)
  { k = x;}
  void showk()
  { cout << k; }
};
int main()
{
  derived ob(3);
  ob.set(1,2); //wont work
  ob.show();   // wont work
  ob.showk();
  return 0;
}
```

protected access

- Protected keyword is included in C++ to provide greater flexibility in inheritance mechanism.
- When members of a class is declared as protected, it is not accessible by non-members of the program. (same as private).
- However, when the class is inherited using the protected access specifier, all public and protected members of the base class become protected members of the derived class.
- Hence it is possible to create class members that are private to their class but that can still be inherited and accessed by a derived class.

protected access

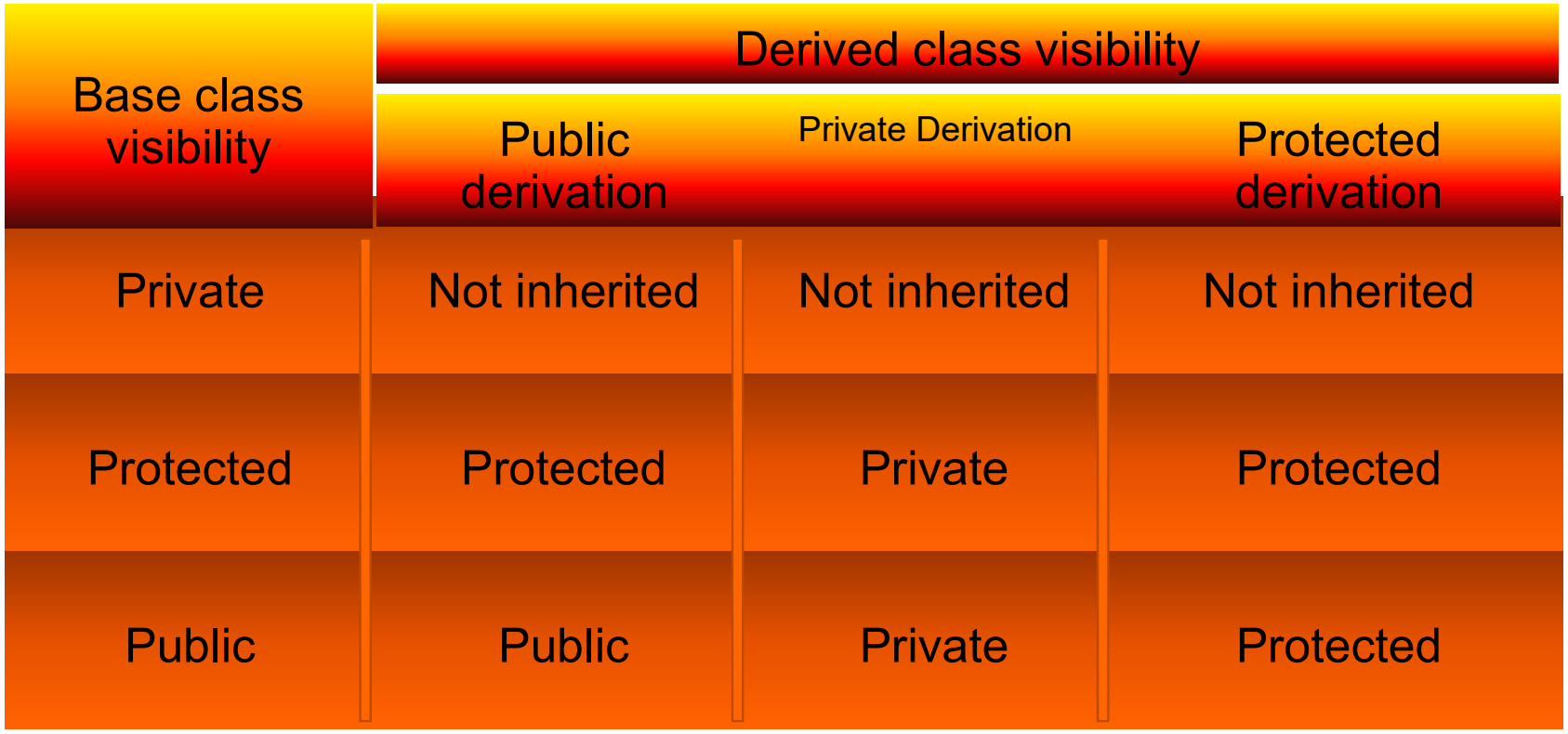
```
class B
{
  protected:
   int i,j;
  public:
   void setij(int a,int b)
   {
     i = a;
     j = b;
   }
   void showij()
   {
     cout << i << j;
   }
};
```

```
class D:protected B
{
   int k;
  public:
   void setk()
   {
     setij(2,3);
     k = i * j;
   }
   void showall()
   {
     cout << k;
     showij();
   }
};
```

protected access

```
int main()
{
  D ob;
  ob.setij(2,3);  //wont work
  ob.setk;
  ob.showall();
  ob.showij();    //wont work
  return 0;
}
```

Inheritance visibility



protected access

```
class B
{
  protected:
   int i,j;

  public:
   void setij(int a,int b)
   {
      i = a;
      j = b;
   }
   void showij()
   {
      cout << i << j;
   }
};
```

```
class D: public B
{
   int k;

  public:
   void setk()
   {
      setij(2,3);
      k = i * j;
   }
   void showall()
   {
      cout << k;
      showij();
   }
};
```

protected access

```
int main()
{
    D ob;
    ob.setij(2,3);
    ob.setk();
    ob.showall();
    ob.showij();
    return 0;
}
```

Constructors, Destructors and inheritance

- A base class , a derived class or both can contain constructor or destructor functions.
- Base's constructor is executed followed by derived's when an object of a derived type is created.
- Derived's destructor is executed before base's destructor when the object is deleted.
- In case of multiple inheritance, constructors functions are executed in their order of derivation and destructor functions are executed in reverse order of derivation.

Constructors and Destructors

```
class base
{
  public:
   base(){cout << “constructing base\n”;}
   ~base() { cout <<  “destructing base\n”;}
};
class derived: public base
{
  public:
   derived() {cout << “constructing derived\n”;}
   ~derived() { cout <<  “destructing derived\n”;}
};
int main()
{
      derived ob;
      return 0;
}
```

Constructors and Destructors

```
class base
{
  public:
   base() { cout << “constructing base\n”; }
   ~base() { cout <<  “destructing base\n”; }
};

class derived1: public base
{
  public:
   derived1() { cout << “constructing derived1\n”; }
   ~derived1() { cout <<  “destructing derived1\n”;}
};
```

Constructors and Destructors

```
class derived2: public derived1
{
  public:
   derived() { cout << “constructing derived2\n”; }
   ~derived() { cout <<  “destructing derived2\n”; }
};

int main()
{
   derived2 ob;
   return 0;
}
```

Constructors and Destructors

```
class base1
{
  public:
   base1() { cout << “constructing base1\n”; }
   ~base1() { cout <<  “destructing base1\n”; }
};

class base2
{
  public:
   base2() { cout << “constructing base2\n”; }
   ~base2() { cout <<  “destructing base2\n”; }
};
```

Constructors and Destructors

```
class derived: public base1, public base2
{
  public:
   derived() { cout << “constructing derived\n”; }
   ~derived() { cout <<  “destructing derived\n”; }
};

int main()
{
   derived ob;
   return 0;
}
```

Constructors and Destructors

```
class base1
{
  public:
   base1() { cout << “constructing base1\n”; }
   ~base1() { cout << “destructing base1\n”; }
};

class base2
{
  public:
   base2() { cout << “constructing base2\n”; }
   ~base2() { cout << “destructing base2\n”; }
};
```

Constructors and Destructors

```
class derived: public base2, public base1
{
  public:
   derived() {cout << “constructing derived\n”;}
   ~derived() { cout <<  “destructing derived\n”;}
};

int main()
{
   derived ob;
   return 0;
}
```

Passing parameters to Base class constructor

- The derived class must declare both the parameter(s) that it requires as well as any required by the base class. The parameters required by the base class are passed to it in the derived class argument list specified into the colon.
- Even if a derived class constructor does not require any arguments, it is necessary to declare one if the base class requires it.
- A derived class' constructor function is free to make use of any and all parameters it is passing to a base class.
- The argument can consist of valid expressions, variables and function calls(dynamic initialization).

Passing parameters to Base class constructor

```
derived_constructor(arg_list) : base1(arg_list), base2(arg_list),...., basen(arg_list)
{
   ........
}
```

Passing parameters

```
class base1{
 public: int x;
   base1(int a){x = a; cout << “constructing base1\n”;}
   ~base1(){cout << “destructing base1\n”;}
};
class derived: public base1{
  int y;
  public:
   derived(int i, int j) :base(j)
   {y = i; cout << “constructing derived\n”;}
   ~derived() { cout << “destructing derived\n”;}
   show(){ cout << x << y;}
};
int main(){
   derived ob(1,2); ob.show(); return 0;
}
```

Passing parameters

```
class base1{
   int x;
  public:
   base1(int a){x = a;cout << “constructing base1\n”;}
   ~base1() { cout << “destructing base1\n”;}
};
class derived: public base1{
  public:
   derived(int i) :base1(i){
      cout << “constructing derived\n”;
   }
   ~derived() { cout << “destructing derived\n”;}
   show(){ cout << x ;}
};
int main(){
      derived ob(1); ob.show();  return 0;
}
```

Passing parameters(1)

```
class base1{
  protected:
   int x;
  public:
   base1(int a){x=a; cout << “constructing base1\n”;}
   ~base1() { cout << “destructing base1\n”; }
};

class  base2{
  protected:
   int y;
  public:
   base2(int b){y=b; cout << “constructing base2\n”;}
   ~base2() { cout << “destructing base2\n”; }
};
```

Inheriting multiple base classes

- It is possible for a derived class to inherit two or more base classes.
- Two or more base classes are inherited, using a comma-separated list.
- An access-specifier for each base class must be specified.

Multiple Inheritance

```
class base1
{
   int i,j;

 public:

   int addij (int a, int b)  {
      i = a;
      j = b;
      return a+b;
   }

   void showij()
   { cout << i << j;}
};
```

```
class base2
{
   int k;

  public:

    void setk()
     {k=10;}

    void showk()
     {cout << k;}
};
```

Multiple Inheritance

```
class derived: public base1, public base2
{
   int m;
  public:
   void setm()
   { m = addij(4,5); }

   void showm ()
   { cout << m;}
};
```

Multiple Inheritance

```
int main()
{
   derived ob;
   int x;
   x = ob.addij(2,3);
   ob.showij();
   ob.setk();
   ob.showk();
   ob.setm();
   ob.showm();
   return 0;
}
```

Multiple Inheritance With different access specifiers for base classes

```
class base
{
   int i,j;
  public:
   int addij (int a, int b)
   {
      i = a;
      j = b;
      return a+b;
   }

   void showij()
   {cout << i << j;}
};
```

```
class base1
{
   int k;

  public:

   void setk()
    {k=10;}

   void showk()
    {cout << k;}
};
```

Multiple Inheritance

```
class derived: public base, private base1
{
        int m;
  public:
        void setm()
        { m = addij(4,5); }

        void showm ()
        {cout << m;}
};
```

Passing parameters(2)

```
class derived: public base1, public base2
{
   int z;
  public:
   derived(int a, int b, int c): base1(b), base2(c)
   { z = a; cout << "constructing derived\n"; }
   ~derived() { cout << "destructing derived\n"; }
   void show() { cout << x << y << z; }
};
int main()
{
   derived ob(1,2,3); ob.show();
   return 0;
}
```

Passing parameters(3)

```
class derived: public base1, public base2
{
   int z;
  public:
   derived(int b, int c) : base1(b), base2(c)
   { z = b+c; cout << “constructing derived\n”; }
   ~derived(){ cout << “destructing derived\n”; }
   void show(){ cout << x << y << z;}
};

int main()
{
   derived ob(2,3);
   ob.show();
   return 0;
```

Replicated base class

```
class base{ public: int i; };
class derived1 : public base{ public: int j;};
class derived2 : public base{  public:int k;};
class derived3 : public derived1, public derived2
{ public: int sum;};
int main()
{
   derived3 ob;
   ob.i = 50; // which i??
   sum = ob.i + ob.j + ob.k;
   cout << sum;
};
```

Replicated base classes

- derived3 inherits the traits of base via two separate paths.
- derived3 will have duplicate sets of members inherited from the base. This creates ambiguity.
- In such a case, where more than one copy of the base is visible, the reference must be explicitly qualified.
- If the base class has a public member i, they can be accessed by specifying derived1::i, derived2::i.

Replicated base class

```
class base{ public: int i; };
class derived1 : public base{ public: int j; };
class derived2 : public base{ public: int k; };
class derived3 : public derived1, public derived2
{
  public:
   int sum;
};
int main()
{
   derived3 ob;
   ob.derived1::i = 50;
   sum = ob.derived2::i + ob.j + ob.k;
   cout << sum;
};
```

Replicated base classes

- The duplication of inherited members due to these multiple paths can be avoided by making the common base class as ***“Virtual Base Class”***.
- This is done by preceding the base class name with the keyword 'virtual' when it is inherited.

Virtual base classes

- When a class is made a virtual base class, only one copy of the class is inherited, regardless of many inheritance paths existing between base and derived classes.

Virtual base class

```
class base{public: int i;};
class derived1:virtual public base{public:int j;};
class derived2:virtual public base{public:int k;};
class derived3:public derived1, public derived2
{ public: int sum; };

int main()
{
   derived3 ob;
   ob.i = 50; ob.j = 20; ob.k = 30;
   sum = ob.i + ob.j + ob.k;
   cout << sum;
};
```

Are you ready to solve…

1. Multilevel Inheritance has got no ambiguity problem .
   a. True
   b. False

   Ans: **a. True**

2. Which member specifier is especially designed for Inheriting classes only.

   a. public b. protected c. private d. None of them

   Ans: **b. Protected**

End of Module 6

Disclaimer

- Some examples and concepts have been sourced from the below links and are open source material
  - **http://cppreference.com**
  - **www.cplusplus.com**

- References:
  - *C++: The Complete Reference* - 4th Edition by Herbert Schildt, Tata McGraw-Hill publications.
  - *The C++ Programming Language* - by Bjarne Stroustrup.
  - *Practical C++ Programming* - by Steve Oualline, O'Reilly publications.

Thank You

Learning & Development, Tata Elxsi, Bangalore.

ITPB Road Whitefield

Bangalore 560 048 India
campus_elxsi@tataelxsi.co.in

www.tataelxsi.com