INHERITANCE

Chapter 7

OBJECTIVES

- Base class access control
- Using **protected** members
- Visibility of base class members in derived class
- Constructors, destructors, and inheritance
- Multiple inheritance
- Virtual base classes

Base Class Access Control

- class derived-class-name : ***access*** base-class-name { … };
- Here ***access*** is one of three keywords
  - public
  - private
  - protected
- Use of ***access*** is optional
  - It is private by default if the derived class is a **class**
  - It is public by default if the derived class is a **struct**

USING PROTECTED MEMBERS

- Cannot be directly accessed by non-related classes and functions
- But can be directly accessed by the derived classes
- Can also be used with structures

VISIBILITY OF BASE CLASS MEMBERS IN DERIVED CLASS

- When a class (derived) inherits from another (base) class, the visibility of the members of the base class in the derived class is as follows.

Member access specifier in base class	Member visibility in derived class		
	Type of Inheritance		
	Private	Protected	Public
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Private	Protected	Protected
Public	Private	Protected	Public

Constructors, Destructors, and Inheritance

- Both base class and derived class can have constructors and destructors.
- Constructor functions are executed in the order of derivation.
- Destructor functions are executed in the reverse order of derivation.
- While working with an object of a derived class, the base class constructor and destructor are always executed no matter how the inheritance was done (private, protected or public).

Constructors, Destructors, and Inheritance (contd.)

```
class base {
public:
   base() {
      cout << “Constructing base class\n”;
   }
   ~base() {
      cout << “Destructing base class\n”;
   }
};
class derived : public base {
public:
   derived() {
      cout << “Constructing derived class\n”;
   }
   ~derived() {
      cout << “Destructing derived class\n”;
   }
};
```

```
void main() {
   derived obj;
}
```

Output:

```
Constructing base class
Constructing derived class
Destructing derived class
Destructing base class
```

Constructors, Destructors, and Inheritance (contd.)

- If a base class constructor takes parameters then it is the responsibility of the derived class constructor(s) to collect them and pass them to the base class constructor using the following syntax -
  - derived-constructor(arg-list) : base(arg-list) { … }
  - Here “base” is the name of the base class
- It is permissible for both the derived class and the base class to use the same argument.
- It is also possible for the derived class to ignore all arguments and just pass them along to the base class.

CONSTRUCTORS, DESTRUCTORS, AND INHERITANCE (CONTD.)

```
class MyBase {
public:
   int x;
   MyBase(int m) { x = m; }
};
class MyDerived : public MyBase {
public:
   int y;
   MyDerived() : MyBase(0) { y = 0; }
   MyDerived(int a) : MyBase(a)
   {
      y = 0;
   }
   MyDerived(int a, int b) : MyBase(a)
   {
      y = b;
   }
};
```

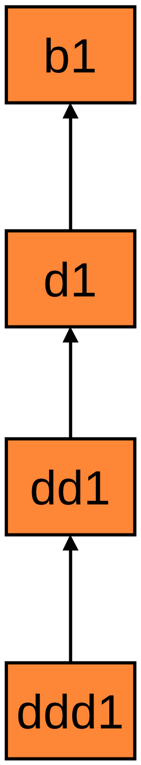
```
void main() {
   MyDerived o1; // x = 0, y = 0
   MyDerived o2(5); // x = 5, y = 0
   MyDerived o3(6, 7); // x = 6, y = 7
}
```

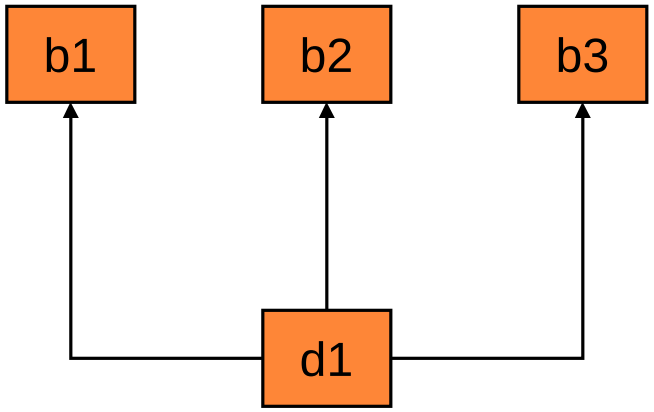
- As “MyBase” does not have a default (no argument) constructor, every constructor of “MyDerived” must pass the parameters required by the “MyBase” constructor.

MULTIPLE INHERITANCE

- A derived class can inherit more than one base class in two ways.
  - Option-1: By a chain of inheritance
    - b1 -> d1 -> dd1 -> ddd1 -> …
    - Here b1 is an indirect base class of both dd1 and ddd1
    - Constructors are executed in the order of inheritance
    - Destructors are executed in the reverse order
  - Option-2: By directly inheriting more than one base class
    - class d1 : *access* b1, *access* b2, …, *access* bN { … }
    - Constructors are executed in the order, left to right, that the base classes are specified
    - Destructors are executed in the reverse order

Multiple Inheritance (contd.)



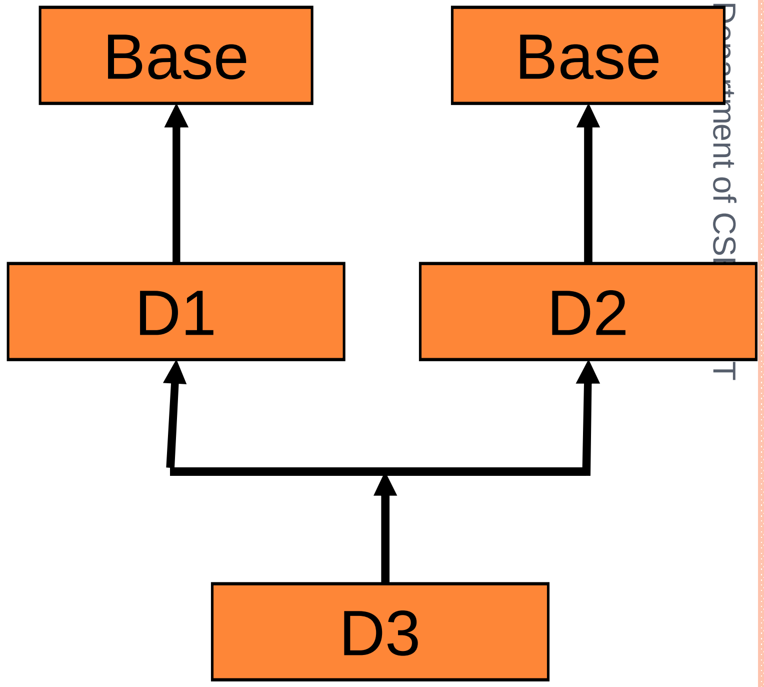


Option - 1

Option - 2

VIRTUAL BASE CLASSES

- Consider the situation shown.
- Two copies of *Base* are included in *D3*.
- This causes ambiguity when a member of *Base* is directly used by *D3*.



Virtual Base Classes (contd.)

```
class Base {
public:
   int i;
};
class D1 : public Base {
public:
   int j;
};
class D2 : public Base {
public:
   int k;
};
```

```
class D3 : public D1, public D2 {
  // contains two copies of 'i'
};
void main() {
   D3 obj;
   obj.i = 10; // ambiguous, compiler error
   obj.j = 20; // no problem
   obj.k = 30; // no problem
   obj.D1::i = 100; // no problem
   obj.D2::i = 200; // no problem
}
```

Virtual Base Classes (contd.)

- class Base {
- public:
-    int i;
- };
- class D1 : **virtual** public Base {
- public:
-    int j;
- }; // activity of D1 not affected
- class D2 : **virtual** public Base {
- public:
-    int k;
- }; // activity of D2 not affected

- class D3 : public D1, public D2 {
-   // contains only one copy of 'i'
- }; // no change in this class definition
- void main() {
-    D3 obj;
-    obj.i = 10; // no problem
-    obj.j = 20; // no problem
-    obj.k = 30; // no problem
-    obj.D1::i = 100; // no problem, overwrites '10'
-    obj.D2::i = 200; // no problem, overwrites '100'
- }

LECTURE CONTENTS

- Teach Yourself C++
  - Chapter 7 (Full, with exercise)
  - Study the examples from the book carefully