OBJECT-ORIENTED PROGRAMMING

Lesson 5

Inheritance: Extending Classes

Inheritance: Extending Classes

1. Introduction
2. Defining Derived Classes
3. Single Inheritance
4. Making a Private Member Inheritable
5. Multilevel Inheritance
6. Multiple Inheritance
7. Hierarchical Inheritance
8. Hybrid Inheritance
9. Virtual Base Classes
10. Abstract Classes
11. Constructors in Derived Classes
12. Member Classes: Nesting of Classes

Key Concepts

- Reusability
- Inheritance
- Single inheritance
- Multiple inheritance
- Multilevel inheritance
- Hybrid inheritance
- Hierarchical inheritance
- Defining a derived class
- Inheriting private members
- Virtual base class
- Direct base class
- Indirect base class
- Abstract class
- Defining derived class constructors
- Nesting of classes

1 Introduction

- Reusability
  - Another important feature of OOP.
  - It is always nice if we could reuse something that already exists rather than trying to create the same all over again.
  - It would not only save time and money but also reduce frustration and increase reliability.
  - For instance, the reuse of a class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

1 Introduction

- Reusability
  - C++ strongly supports the concept of reusability.
  - The C++ classes can be reused in several ways.
  - Once a class has been written and tested, it can be adapted by other programmers to suit their requirements.
  - This is basically done by creating new classes, reusing the properties of the existing ones.

1 Introduction

- Inheritance (Derivation)
  - The mechanism of deriving a new class from an old one.
- Base class
  - Refer to the old class.
- Derived class (Subclass)
  - Refer to the new class.
  - The derived class inherits some or all of the traits from the base class.

1 Introduction

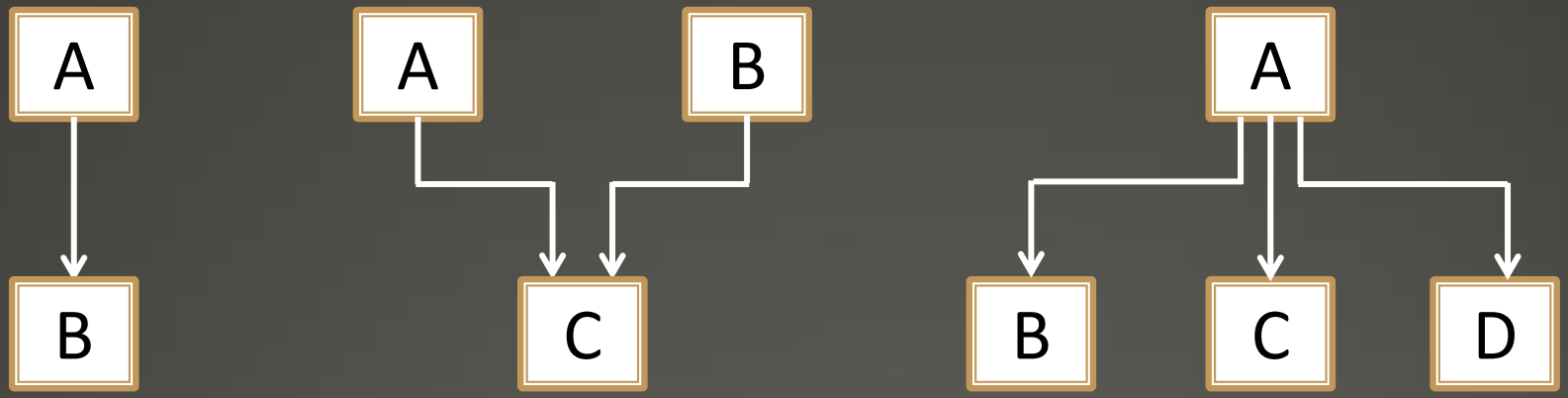
- A class can inherit properties from more than one class or from more than one level.

1 Introduction

- Single inheritance
  - A derived class with only one base class.
- Multiple inheritance
  - A derived class with several base classes.
- Hierarchical inheritance
  - The traits of one class may be inherited by more than one class.
- Multilevel inheritance
  - Deriving a class from another ‘derived class’.

1 Introduction

- Various forms of inheritance that could be used for writing extensible programs.



(a)Single Inheritance

(b)Multiple Inheritance

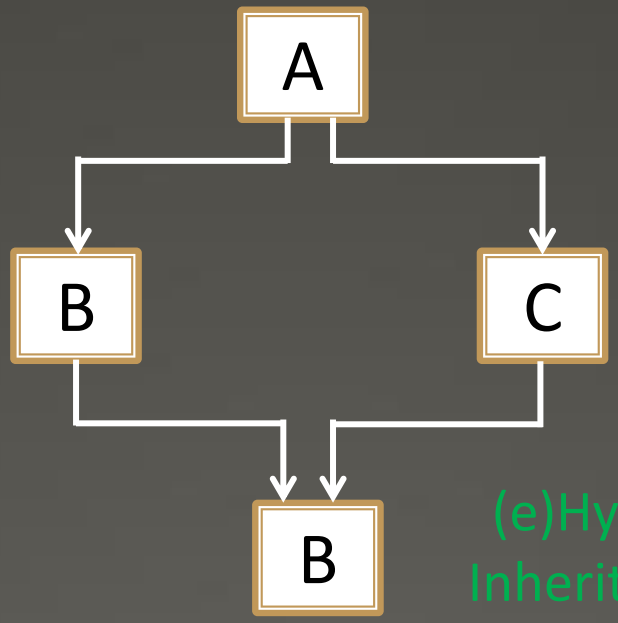
(c)Hierarchical Inheritance

Forms of inheritance

1 Introduction

- Various forms of inheritance that could be used for writing extensible programs.

(d)Multilevel Inheritance



(e)Hybrid Inheritance

Forms of inheritance

2 Defining Derived Classes

- A derived class can be defined by specifying its relationship with the base class in addition to its own details.

2 Defining Derived Classes

- The general form of defining a derived class is

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

  - The colon indicates that the *derived-class-name* is derived from the *base-class-name*.
  - The *visibility-mode* is optional and, if present, may be either private or public.(By default it is *private*.)
  - Visibility mode specifies whether the features of the base class are *privately derived* or publicly derived.

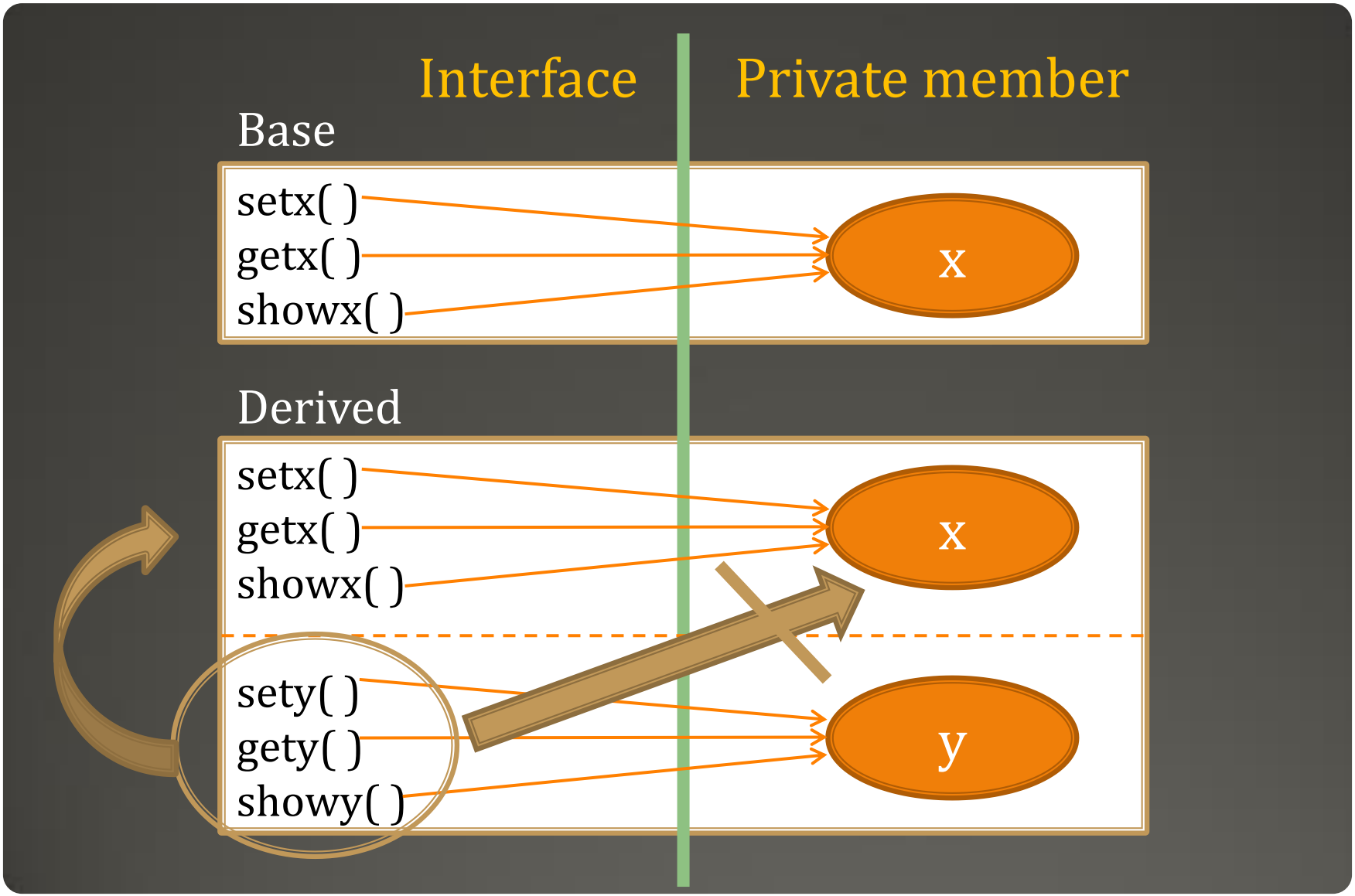
2 Defining Derived Classes

- Examples:

```
class ABC: private XYZ          //private derivation
{
        members of ABC
};
class ABC: public XYZ           //public deviation
{
        members of ABC
};
class ABC: XYZ                  //private derivation by default
{
        members of ABC
};
```

2 Defining Derived Classes

- Publicly inherited
  - When the base class is publicly inherited, 'public members' of the base class become 'public members' of the derived class.
  - And therefore the public members of the base class are accessible to the objects of the derived class.
  - Example (Eg4-2)



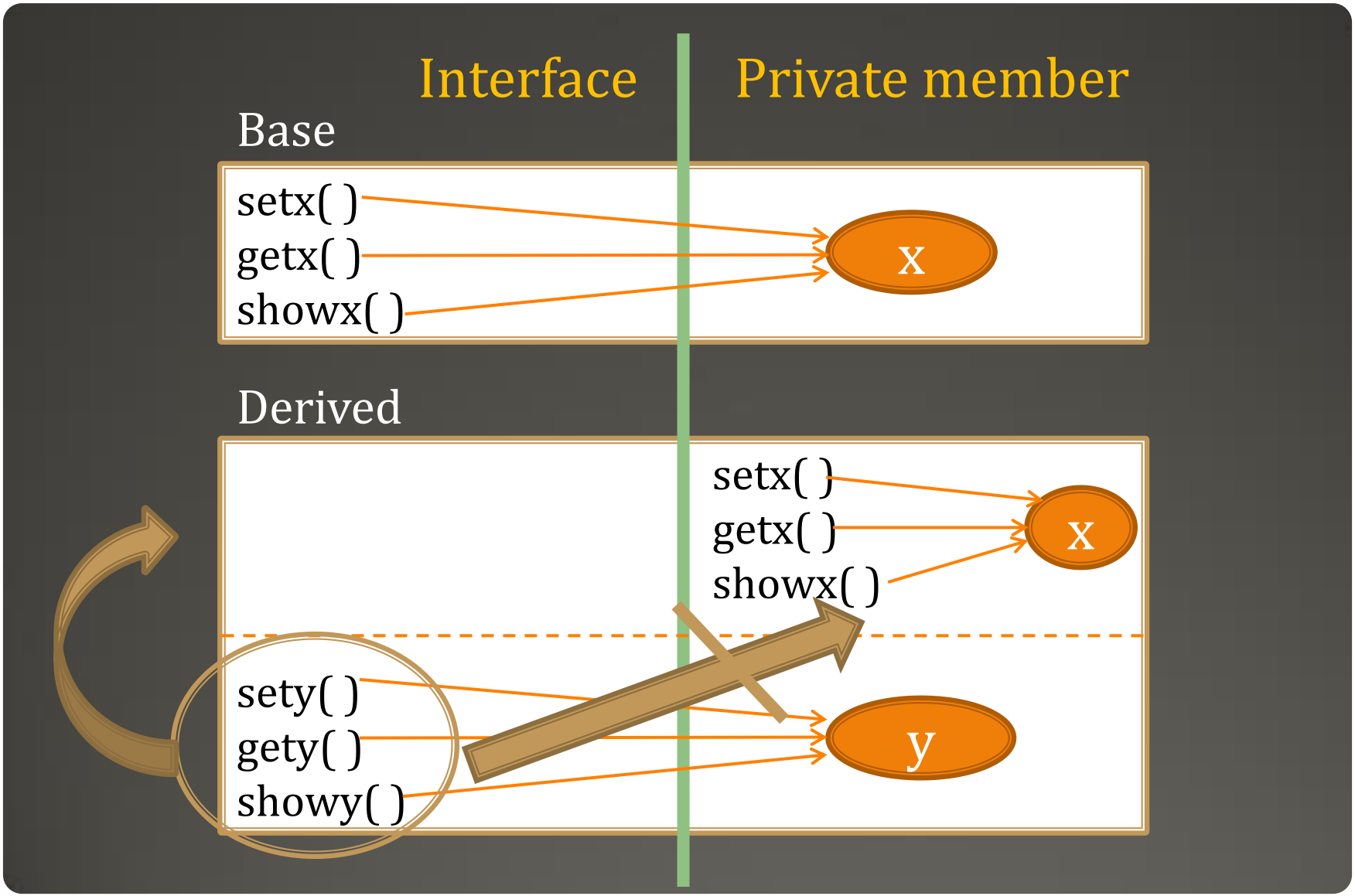
2 Defining Derived Classes

- Privately inherited
  - When a base class is privately inherited by a derived class, 'public members' of the base class become 'private members' of the derived class.
  - And therefore the public members of the base class can only be accessed by the member functions of the derived class.
  - They are inaccessible to the objects of the derived class.
  - Remember, a public member of a class can be accessed by its own objects using the **dot operator**.
  - The result is that no member of the base class is accessible to the objects of the derived class.

2 Defining Derived Classes

- Privately inherited
  - Example (Eg4-2)

```
class  Derived : private  Base
{
            .....
};
```



2 Defining Derived Classes

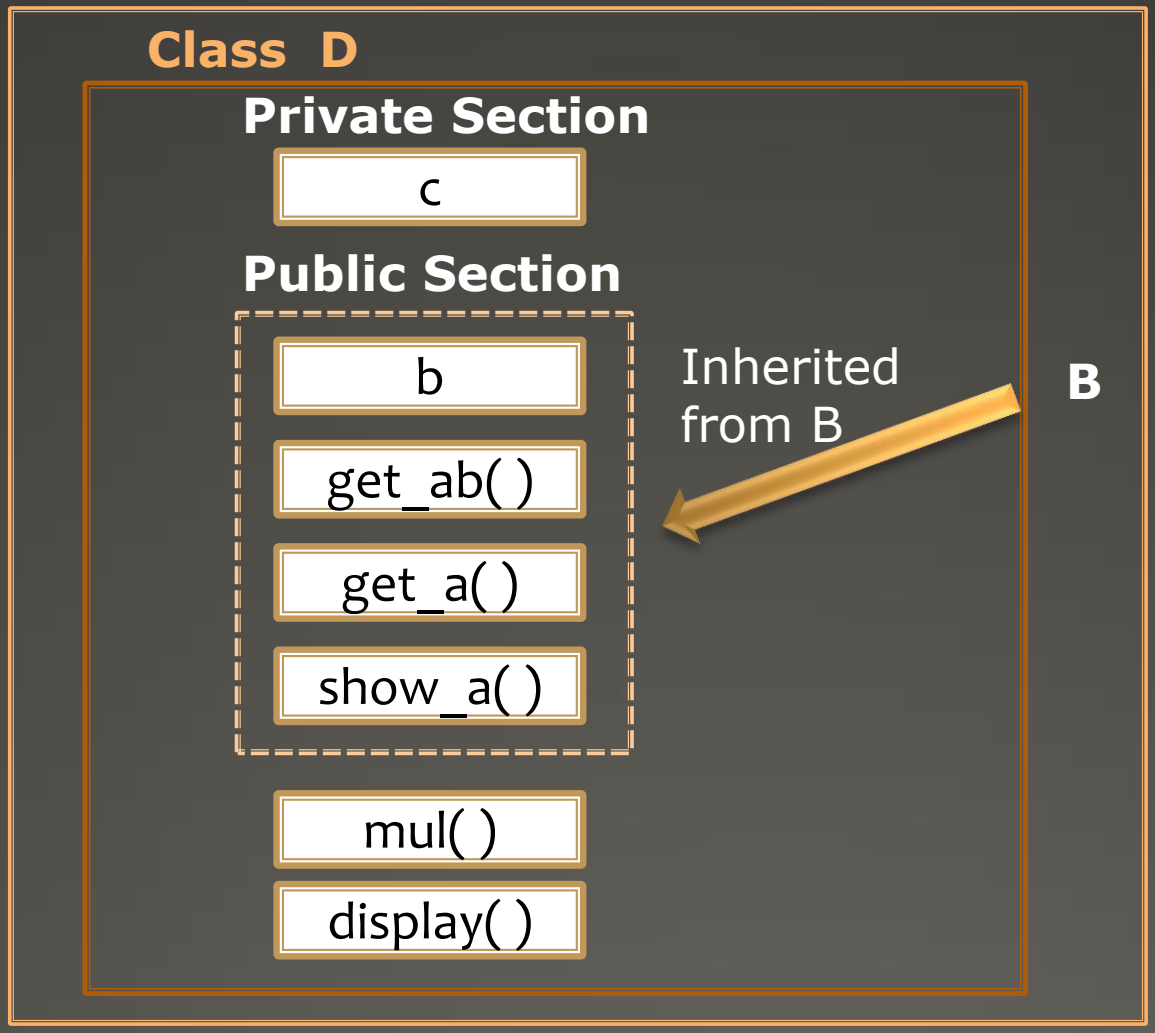
- Privately inherited & publicly inherited
  - In both the cases, the private members are not inherited.
  - And therefore, the private members of a base class will never become the members of its derived class.

2 Defining Derived Classes

- In 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.

3 Single Inheritance

- Example (Program 8.1)
  - Shows a base class B and a derived class D.
  - The class B contains one private data member, one public data member, and three public member functions.
  - The class D contains one private data member and two public member functions.



Adding more members to a class (by public derivation)

3 Single Inheritance

- Example (Program 8.1)
  - The objects of class **D** have access to all the public members of **B**.

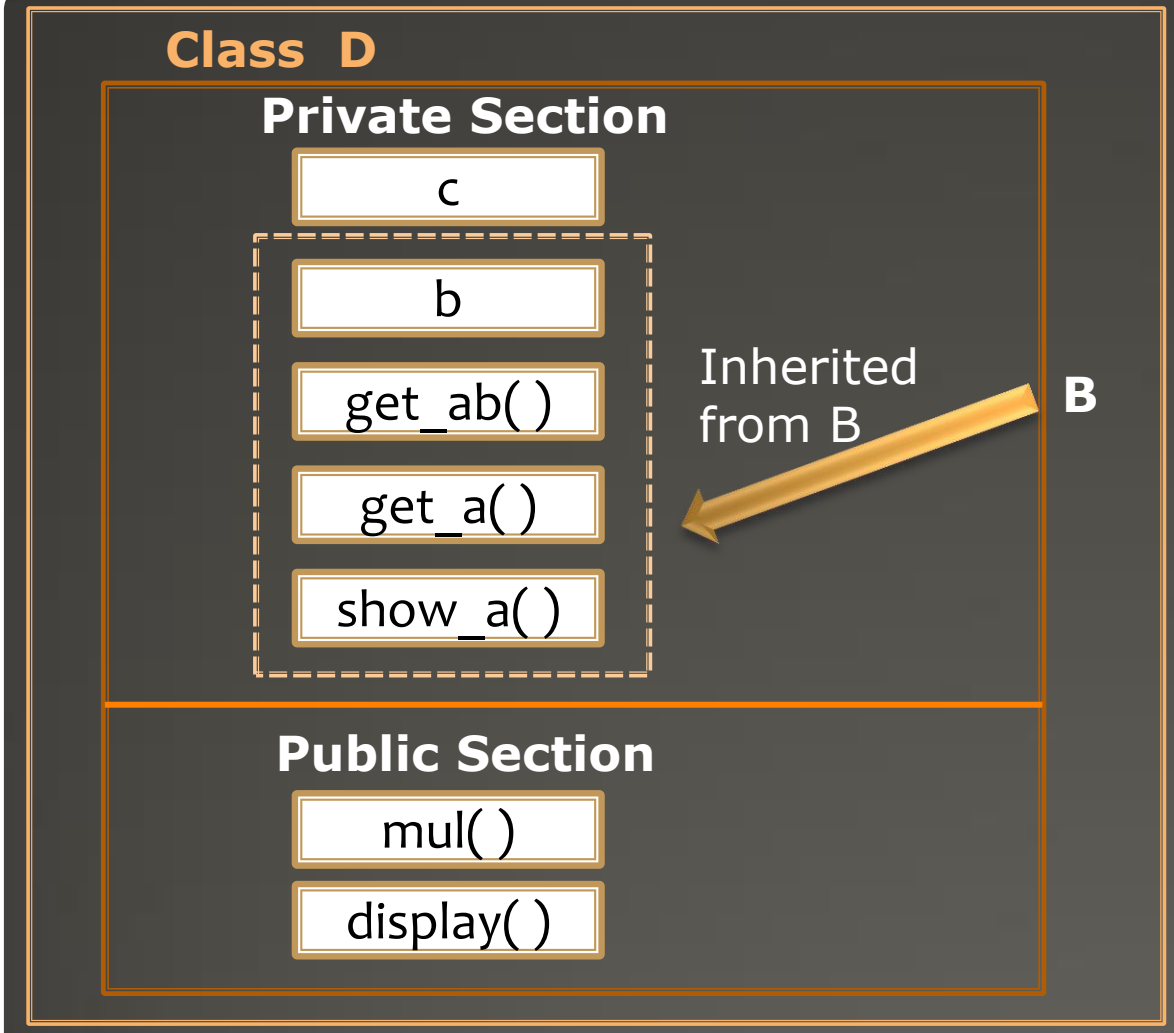
```
void  show_a( )
{
        cout << "a = " << a << "\n";
}
void  mul( )
{
        c = b * get_a( );              //c = b * a
}
```

  - Although the data member **a** is private in **B** and cannot be inherited, objects of **D** are able to access it through an inherited member function of **B**.

3 Single Inheritance

- The case of private derivation
  - The public members of the base class become private members of the derived class.

```
class  B
{
        int  a;
public:
        int  b;
        void  get_ab( );
        void  get_a( );
        void  show_a( );
};
class  D  :  private  B        //private derivation
{
        int  c;
public:
        void  mul( );
        void  display( );
};
```



The objects of D can not have direct access to the public member functions of B.

Adding more members to a class (by public derivation)

3 Single Inheritance

- The statement below will not work.

```
d.get_ab( );        //get_ab( ) is private
d.get_a( );         //so also get_a( )
d.show_a( );        //and show_a( )
```

3 Single Inheritance

- However, these functions can be used inside **mul()** and **display()** like the normal functions.

```
void  mul( )
{
        get_ab( );
        c = b * get_a( );
}
void display()
{
        show_a( );                    //outputs value of 'a'
        cout << "b = " << b << "\n" << "c = " << c << "\n\n";
}
```

3 Single Inheritance

- Example (Program 8.2)
  - It incorporates these modifications for private derivation.
  - Compare this with program 8.1

3 Single Inheritance

- Suppose
  - A base class and a derived class define a function of the same name.
  - What will happen when a derived class object invokes the function?
  - In such cases, the derived class function supersedes the base class definition.
  - The base class function will be called only if the derived class does not redefine the function.

4 Making a Private Member Inheritable

- What do we do if the private data needs to be inherited by a derived class?
  - This can be accomplished by modifying the visibility limit of the private member by making it public.
  - This would make it accessible to all the other functions of the program, thus taking away the advantages of data hiding.

4 Making a Private Member Inheritable

- Protected
  - A third visibility modifier, provided by C++, which serve a limited purpose in inheritance.
  - 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.

4 Making a Private Member Inheritable

- A class can now use all the three visibility modes.

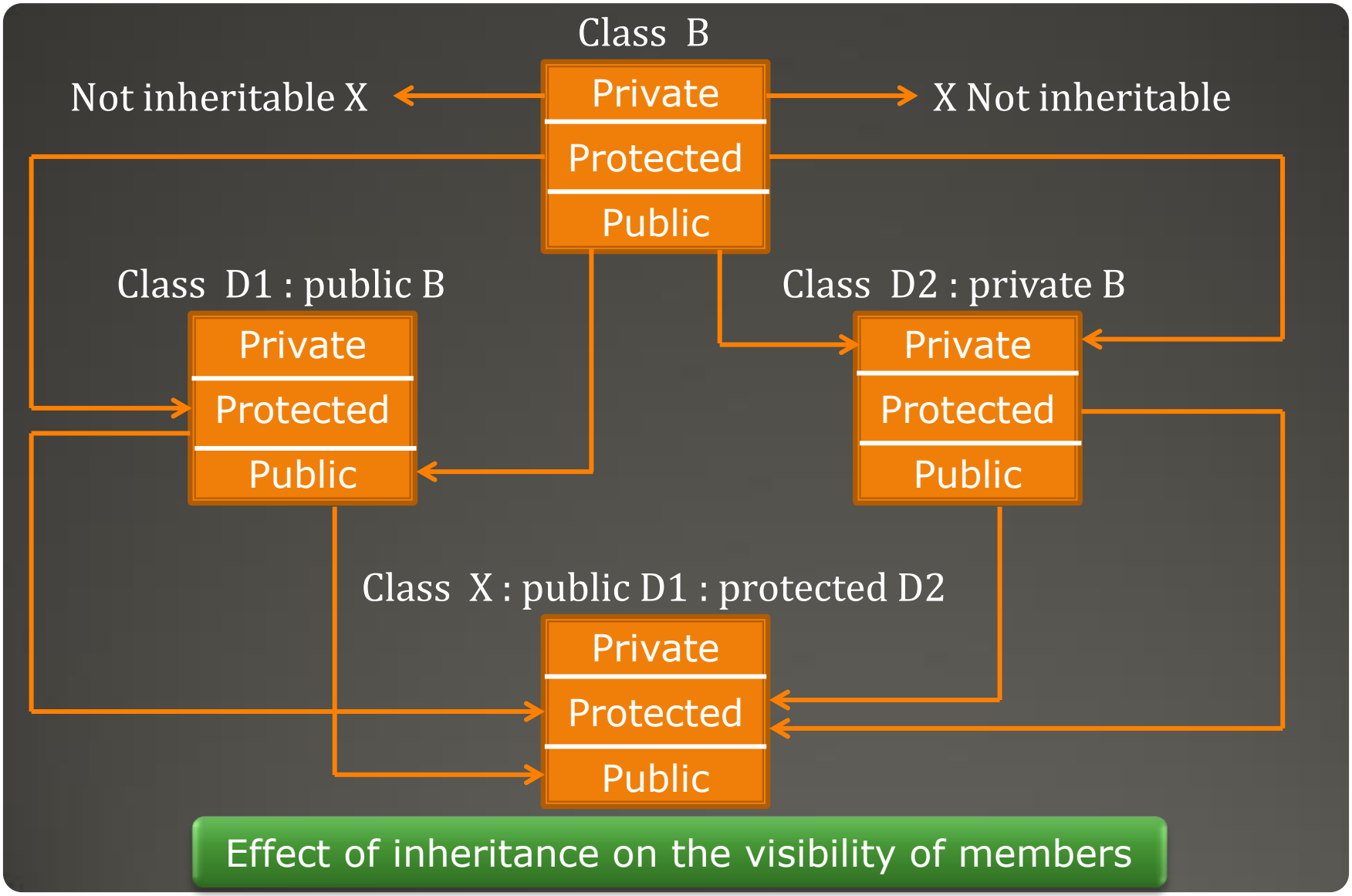
```
class   alpha
{
private:              //optional
        ……    //visible to member functions within its class
protected:
        ……    //visible to member functions
        ……    //of its own and derived class
public:
        ……    //visible to all functions in the program
};
```

4 Making a Private Member Inheritable

- When a protected member is inherited in public mode
  - It becomes protected in the derived class too and therefore is accessible by the member functions of the derived class.
  - It is also ready for further inheritance.

4 Making a Private Member Inheritable

- When a protected member is inherited in private mode
  - It becomes private in the derived class.
  - Although it is available to the member functions of the derived class, it is not available for further inheritance (since private members cannot be inherited).



4 Making a Private Member Inheritable

- The keywords private, protected, and public may appear in any order and any number of times in the declaration of a class.
  - Example is a valid class definition.

```
class beta
{
        protected:
                ……
        public:
                ……
        private:
                ……
        public:
                ……
};
```

4 Making a Private Member Inheritable

- However, the normal practice is to use as follows

```
class beta
{
                    ……                    //private by default
                    ……
          protected:
                    ……
          public:
                    ……
};
```

- Example (Eg 4-1)

4 Making a Private Member Inheritable

- Protected derivation
  - Inherit a base class in protected mode.
  - Both the public and protected members of the base class become protected members of the derived class.

4 Making a Private Member Inheritable

Visibility of inherited members

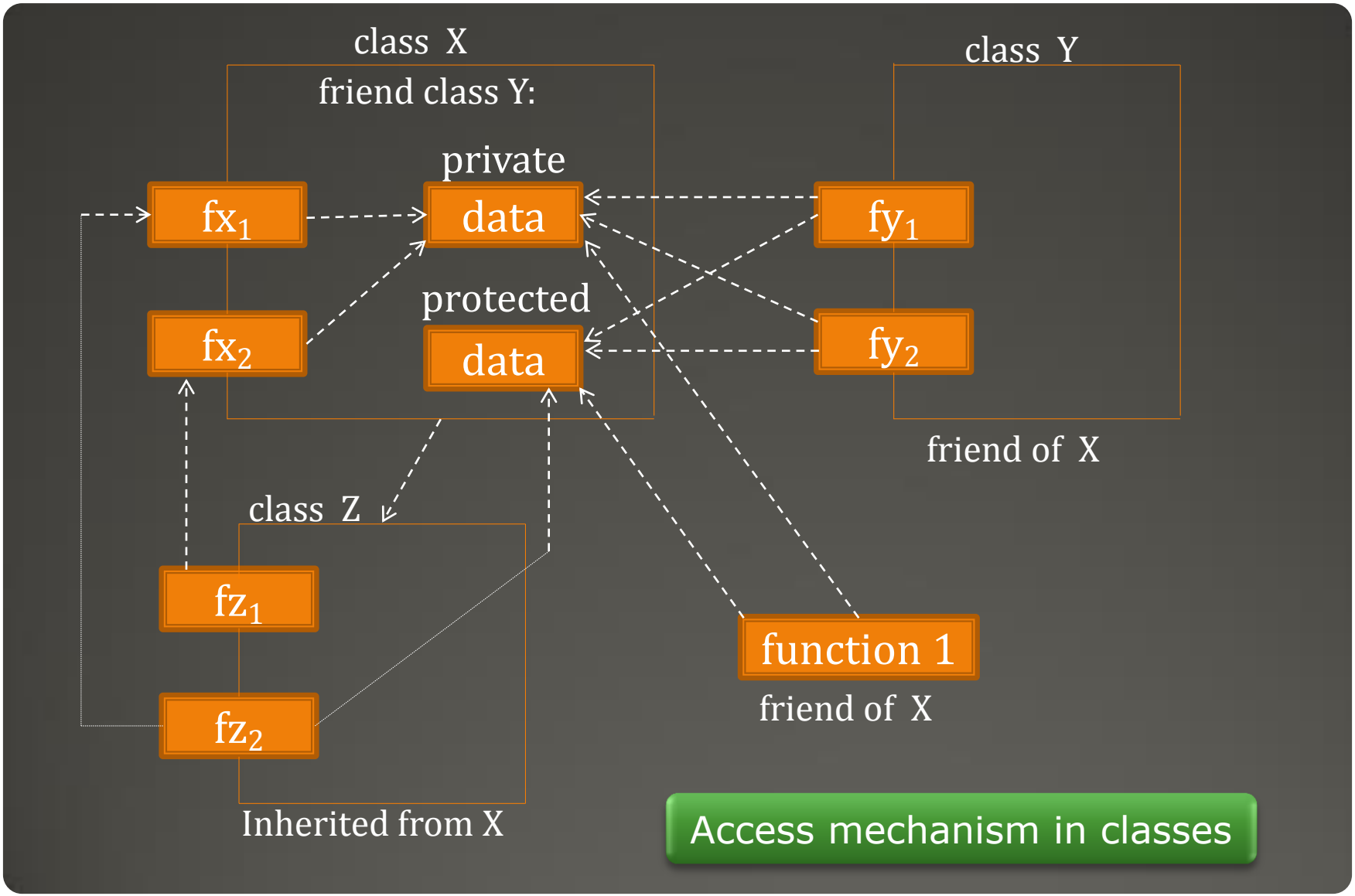
Base class visibility	Derived class visibility		
	Public derivation	Private derivation	Protected derivation
Private →	Not inherited	Not inherited	Not inherited
Protected →	Protected	Private	Protected
Public →	Public	Private	Protected

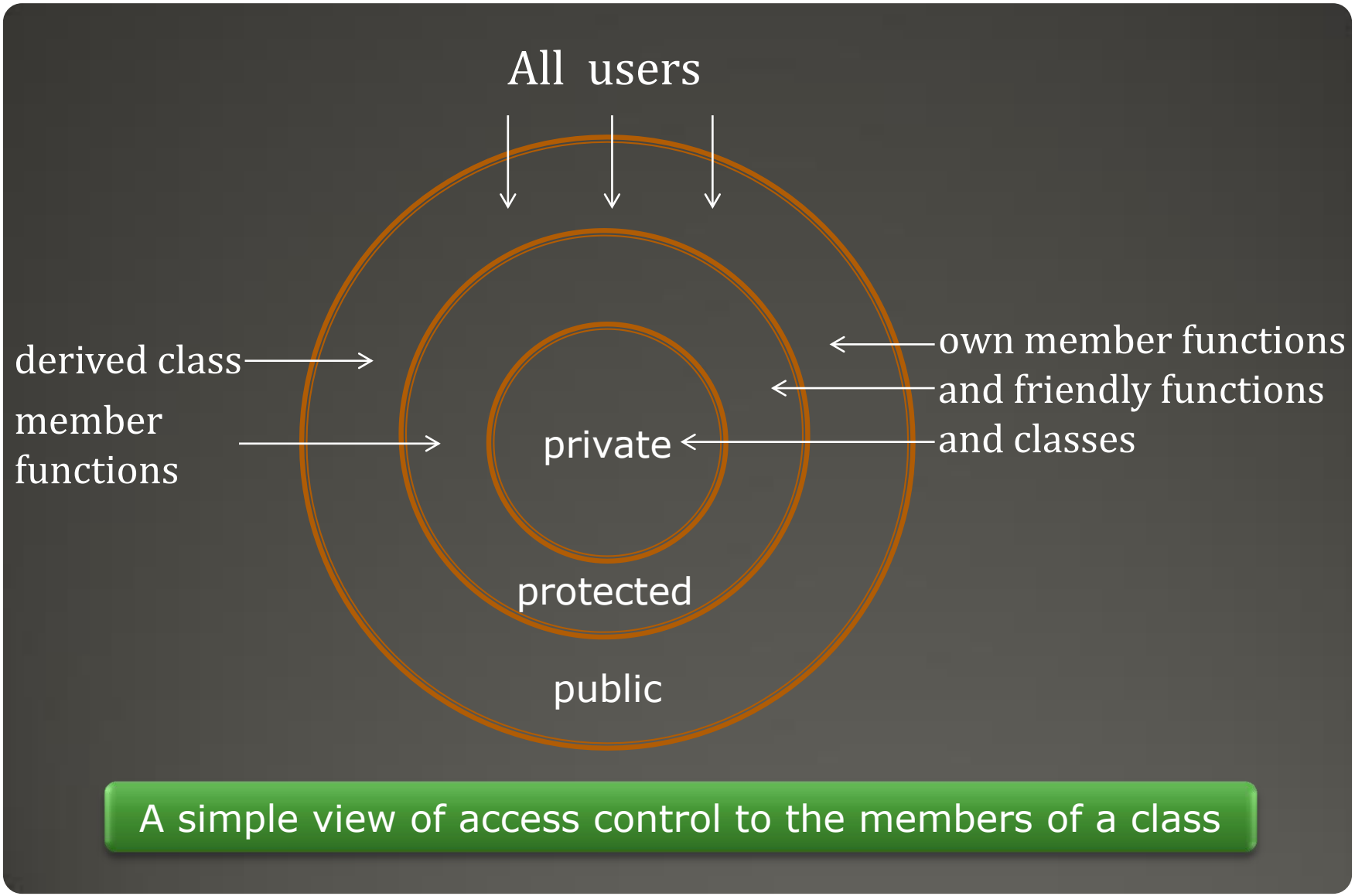
4 Making a Private Member Inheritable

- What are the various functions that can have access to the private and protected members of a class?
  1. A function that is a friend of the class.
  2. A member function of a class that is a friend of the class.
  3. A member function of a derived class.

4 Making a Private Member Inheritable

- Note
  - The friend functions and the member functions of a friend class can have direct access to both the private and protected data.
  - While the member functions of a derived class can directly access only the protected data, they can access the private data through the member functions of the base class.

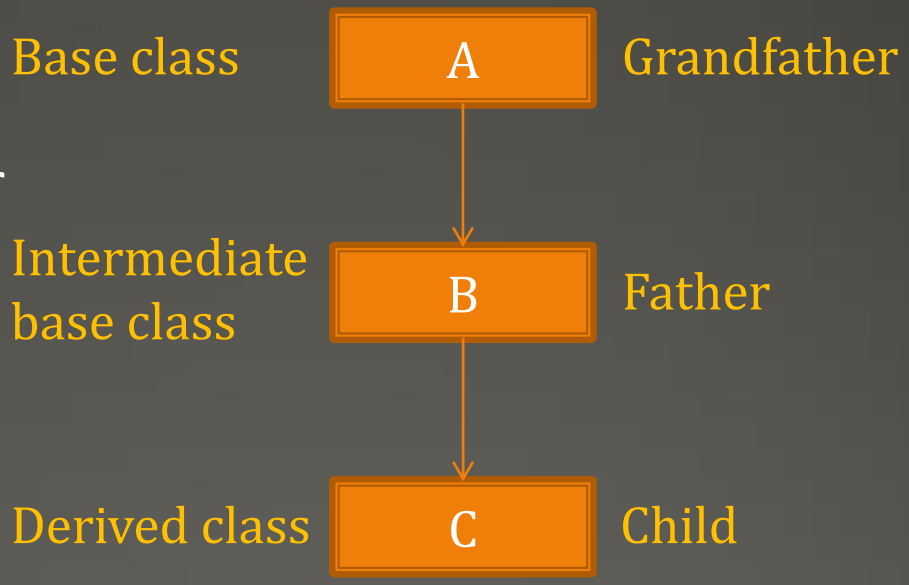




A simple view of access control to the members of a class

5 Multilevel Inheritance

- A class is derived from another derived class
  - The class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.
  - The class B is known as intermediate base class since it provides a link for the inheritance between A and C.
  - The chain ABC is known as inheritance path.



Multilevel inheritance

5 Multilevel Inheritance

- A derived class with multilevel inheritance is declared as follows:

```
class  A {……};                    //Base  class
class  B:  public  A {……};         //B derived from A
class  C:  public  B {……};         //C derived from B
```

  - This process can be extended to any number of levels.

5 Multilevel Inheritance

- Example
  - Assume that the test results of a batch of students are stored in three different classes.
    - Class **student** stores the roll-number.
    - Class **test** stores the marks obtained in two subjects.
    - Class **result** contains the total marks obtained in the test.
    - The class **result** can inherit the details of the marks obtained in the test and the roll-number of students through multilevel inheritance.

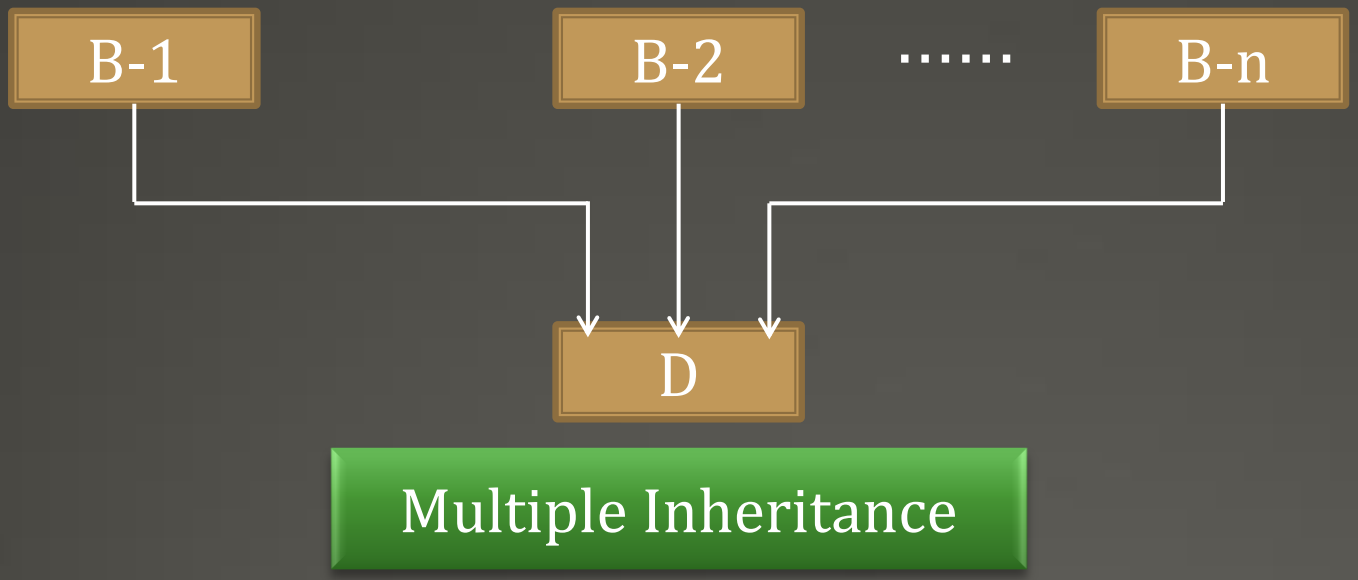
5 Multilevel Inheritance

- Example (Program 8.3)
  - The class result, after inheritance from 'grandfather' through 'father', would contain the following numbers

```
private:
        float  total;                   //own member
protected:
        int  roll_number;               //inherited from student via test
        float  sub1;                    //inherited from test
        float  sub2;                    //inherited from test
public:
        void  get_number(int);          //from student via test
        void  put_number(void);         //from student via test
        void  get_marks(float, float);  //from test
        void  put_marks(void);          //from test
        void  display(void);            //own member
```

6 Multiple Inheritance

- Multiple inheritance
  - A class can inherit the attributes of two or more classes.



  - It allows us to combine the features of several existing classes as a starting point for defining new classes.

6 Multiple Inheritance

- Multiple inheritance
  - The syntax of a derived class with multiple base classes

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

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

6 Multiple Inheritance

- Multiple inheritance
  - Example (Program 8.4)
  - Example (Eg 4-9)

6 Multiple Inheritance

- Ambiguity Resolution in Inheritance
  - Problem
    - When a function with the same name appears in more than one base class.

6 Multiple Inheritance

- Ambiguity Resolution in Inheritance
  - Which display() function is used by the derived class when we inherit these two classes?

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

6 Multiple Inheritance

- Ambiguity Resolution in Inheritance
  - We can solve this problem by defining a **named instance** within the derived class, using the class resolution operator with the function:

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

6 Multiple Inheritance

- Ambiguity Resolution in Inheritance
  - We can now use the derived class:

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

6 Multiple Inheritance

- Ambiguity Resolution in Inheritance
  - Ambiguity may also arise in **single inheritance** applications.
  - For instance, 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.

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

6 Multiple Inheritance

- Ambiguity Resolution in Inheritance
  - However, we may invoke the function defined in **A** by using the scope resolution operator to specify the class.

```
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;
}
```

  - Output

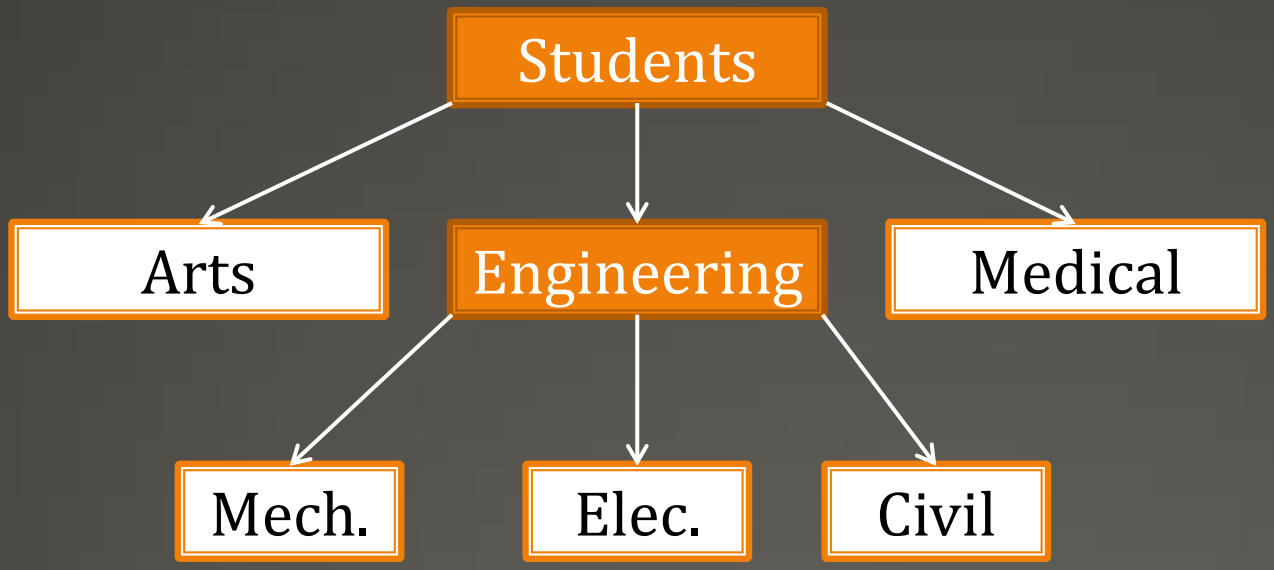
```
B
A
B
```

7 Hierarchical Inheritance

- The interesting application of inheritance is to use it as a support to the hierarchical design of a program.
  - Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level.

7 Hierarchical Inheritance

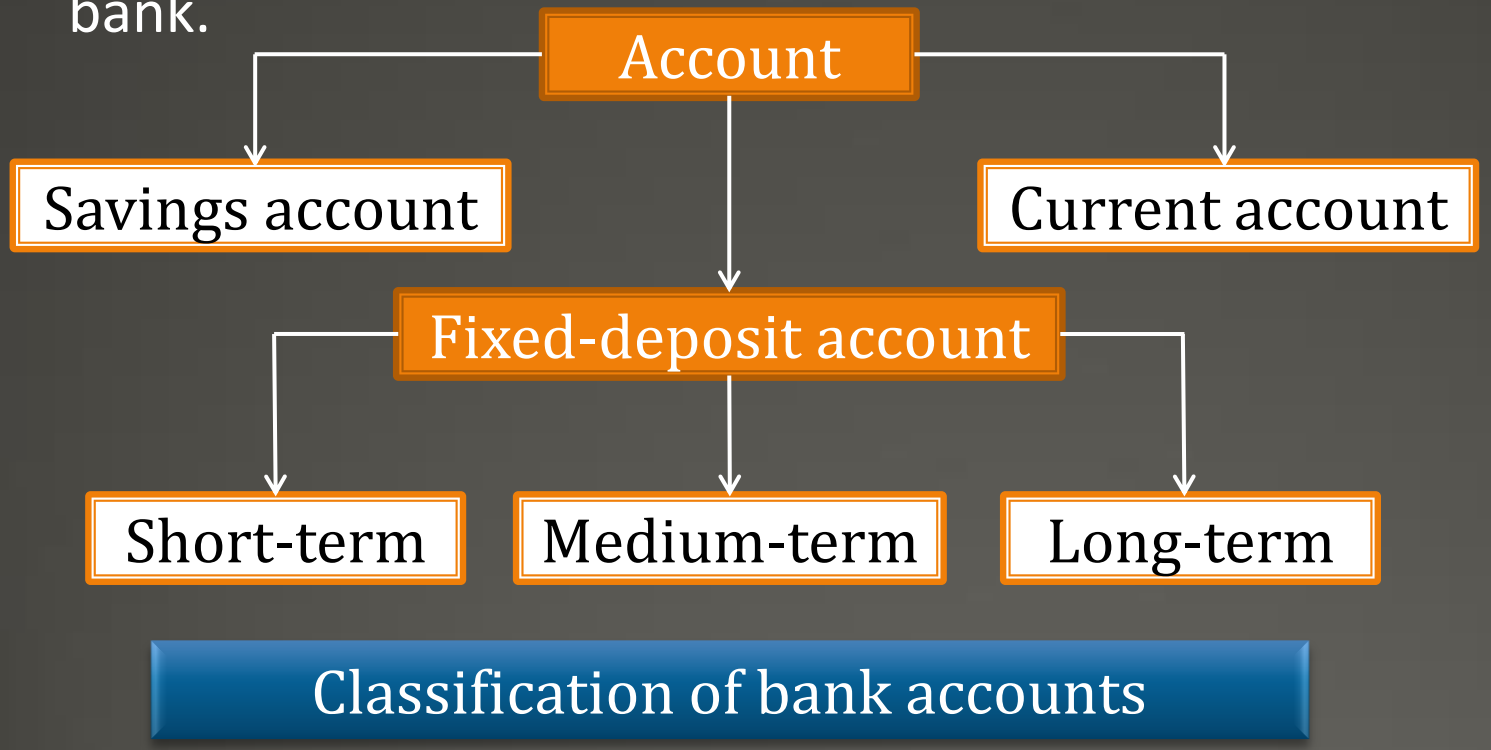
- Example
  - A hierarchical classification of students in a university.



Hierarchical classification of students

7 Hierarchical Inheritance

- Example
  - A hierarchical classification of accounts in a commercial bank.



7 Hierarchical Inheritance

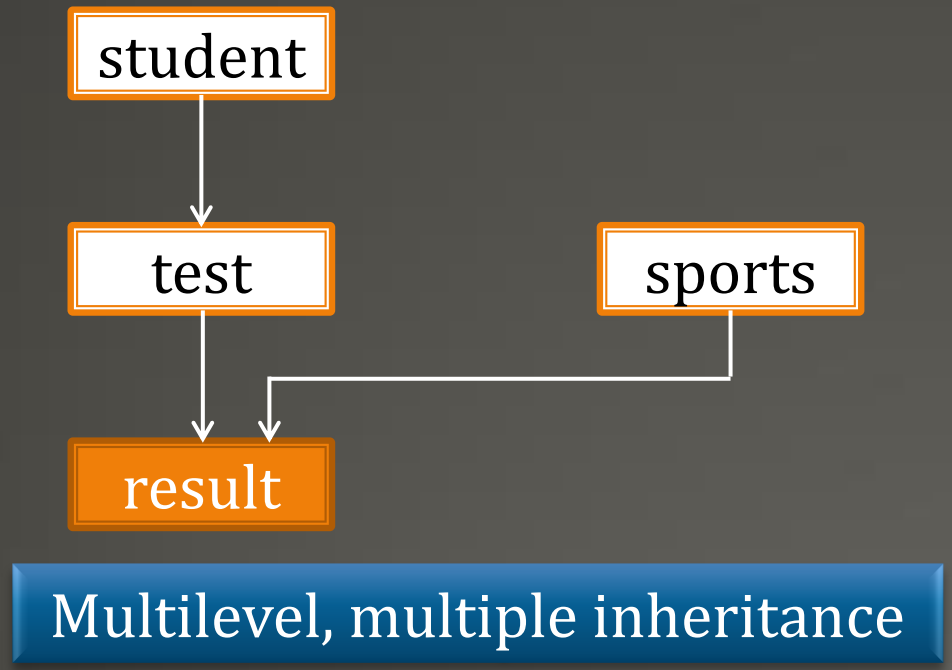
- Example
  - All the students have certain things in common.
  - Similarly, all the accounts possess certain common features.
  - In C++, such problems can be easily converted into class hierarchies.

7 Hierarchical Inheritance

- The base class will include all the features that are common to the subclasses.
- A subclass can be constructed by inheriting the properties of the base class.
- A subclass can serve as a base class for the lower level classes and so on.

8 Hybrid Inheritance

- There could be situations where we need to apply two or more types of inheritance to design a program.
  - Example (Program 8.5)

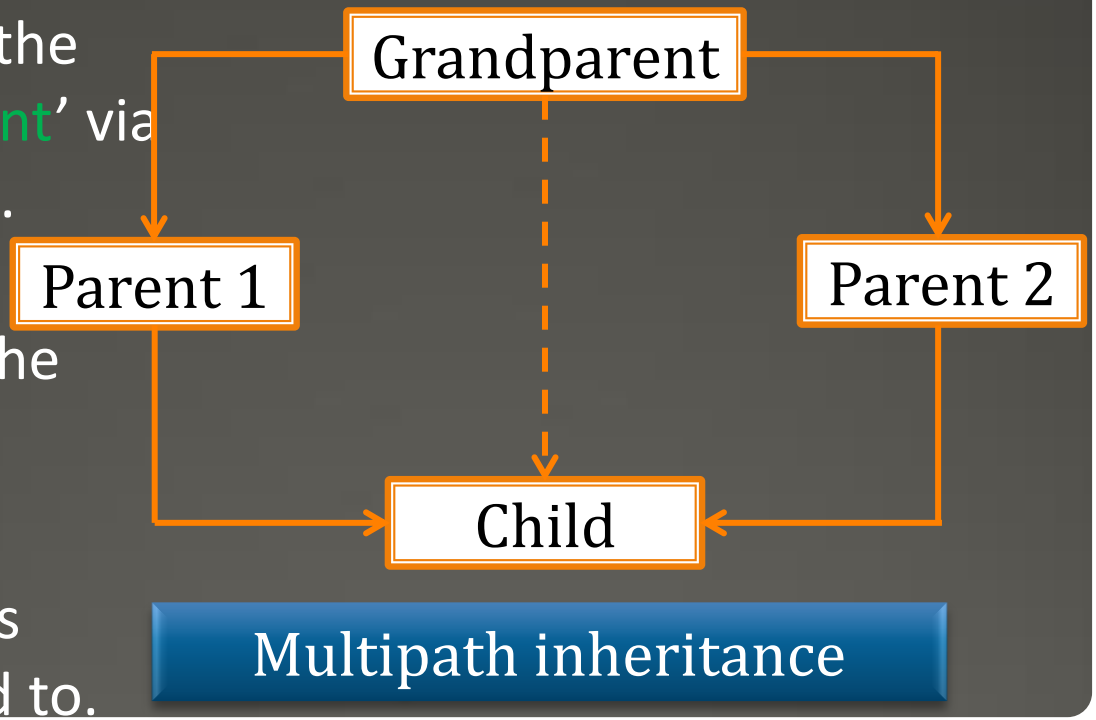


9 Virtual Base Classes

- Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved.

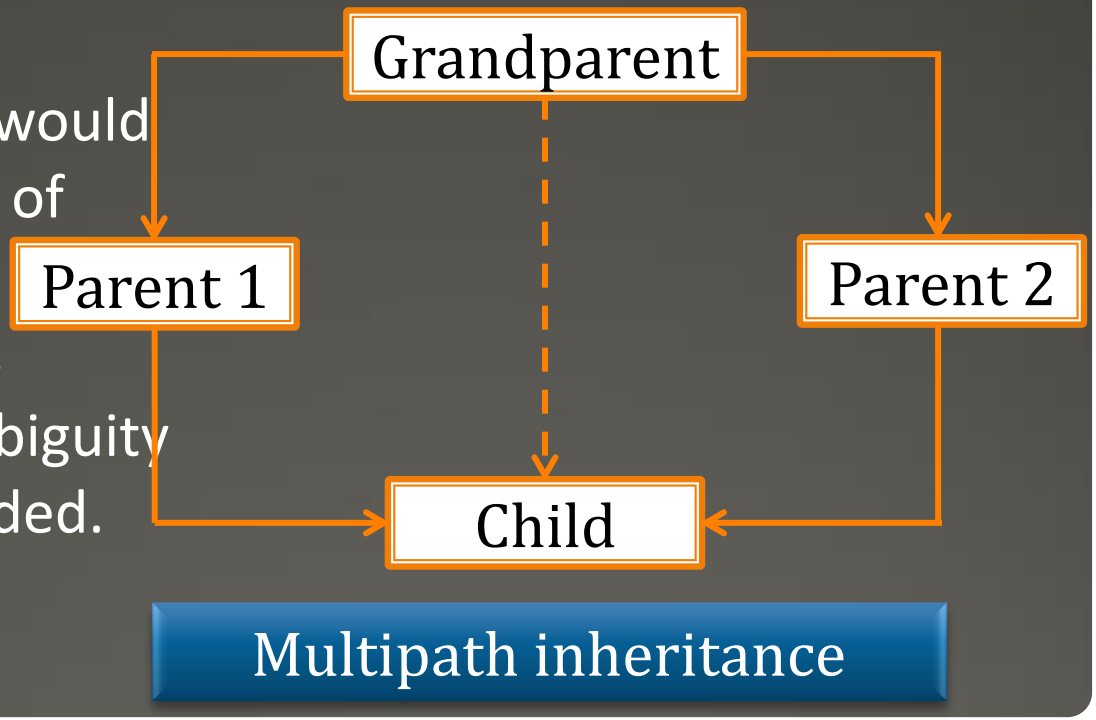
9 Virtual Base Classes

- Direct base classes
  - 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.
  - The 'child' can also inherit directly by the broken line.
- Indirect base class
  - The 'grandparent' is sometimes referred to.



9 Virtual Base Classes

- Inheritance by the 'child' 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 members inherited from 'grandparent'.
  - This introduces ambiguity and should be avoided.



9 Virtual Base Classes

- Virtual base class
  - 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.

9 Virtual Base Classes

```
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
};
```

9 Virtual Base Classes

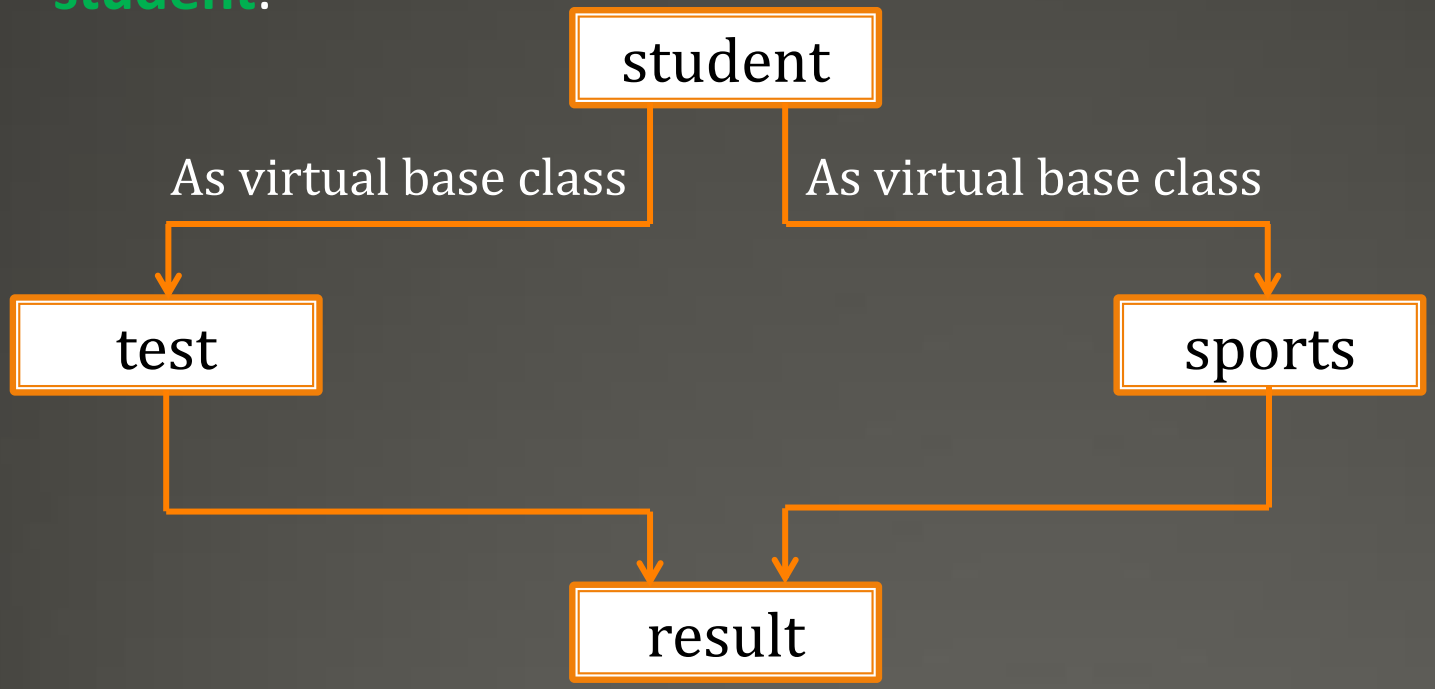
- Note
  - The keywords virtual and public may be used in either order.

9 Virtual Base Classes

- When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.

9 Virtual Base Classes

- Example (Program 8.6)
  - The class **sports** derives the **roll_number** from the class **student**.



10 Abstract Classes

- Abstract class
  - One that is not used to create objects.
  - It is designed only to act as a base class (to be inherited by other classes).
  - It is a design concept in program development and provides a base upon which other classes may be built.
  - In the previous example, the **student** class is an abstract class since it was not used to create any objects.

11 Constructors in Derived Classes

- Note
  - As long as no base class constructor takes any arguments, the derived class need not have a constructor function.
  - However, if any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors.

11 Constructors in Derived Classes

- Remember
  - While applying inheritance, we usually create objects using the derived class.
  - Thus, it makes sense for the derived class to pass arguments to the base class constructor.
  - When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

11 Constructors in Derived Classes

- In case of multiple inheritance
  - The base classes are constructed in the order in which they appear in the declaration of the derived class.
- In case of multilevel inheritance
  - The constructors will be executed in the order of inheritance.

11 Constructors in Derived Classes

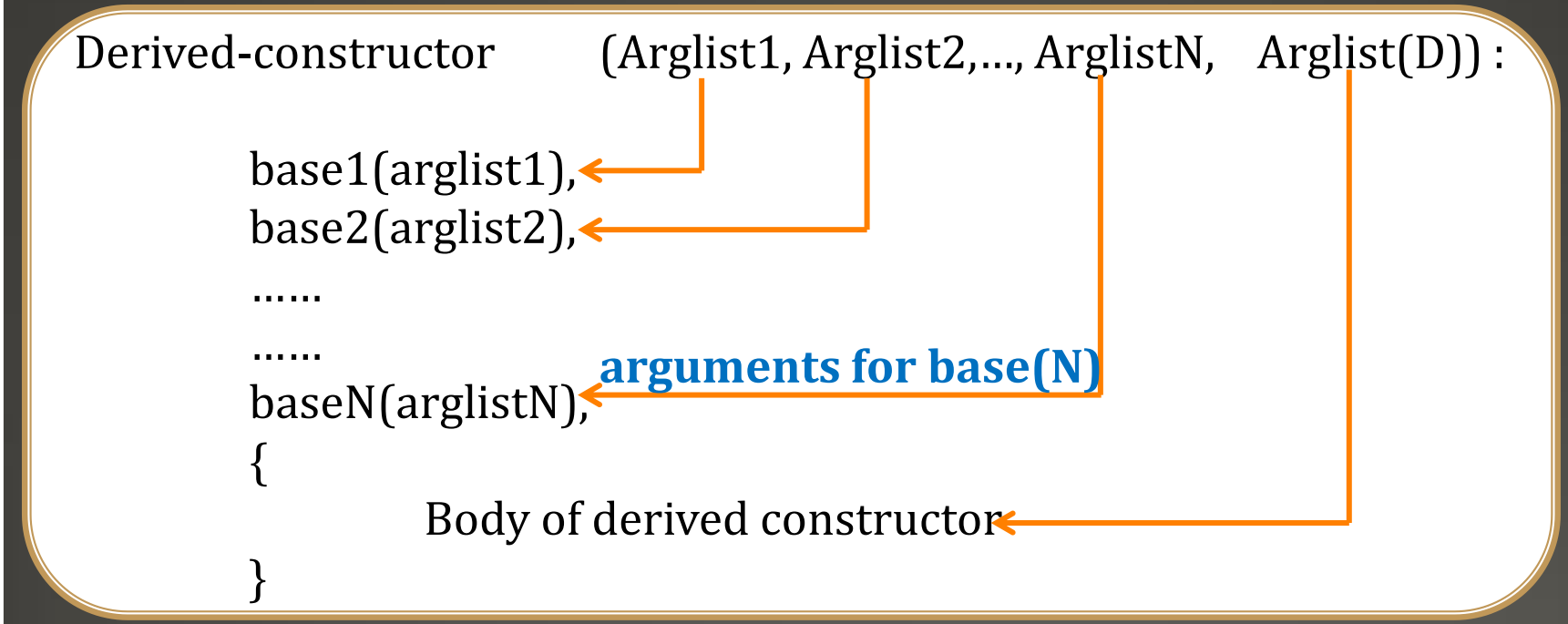
- Since the derived class takes the responsibility of supplying the initial values to its base classes, we supply the initial values that are required by all the classes together, when a derived class object is declared.

11 Constructors in Derived Classes

- How the initial values are passed to the base class constructors so that they can do their job?
  - C++ supports a special argument passing mechanism.
  - The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class.
  - The base constructors are called and executed before executing the statements in the body of the derived constructor.

11 Constructors in Derived Classes

- The general form of defining a derived constructor is



11 Constructors in Derived Classes

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

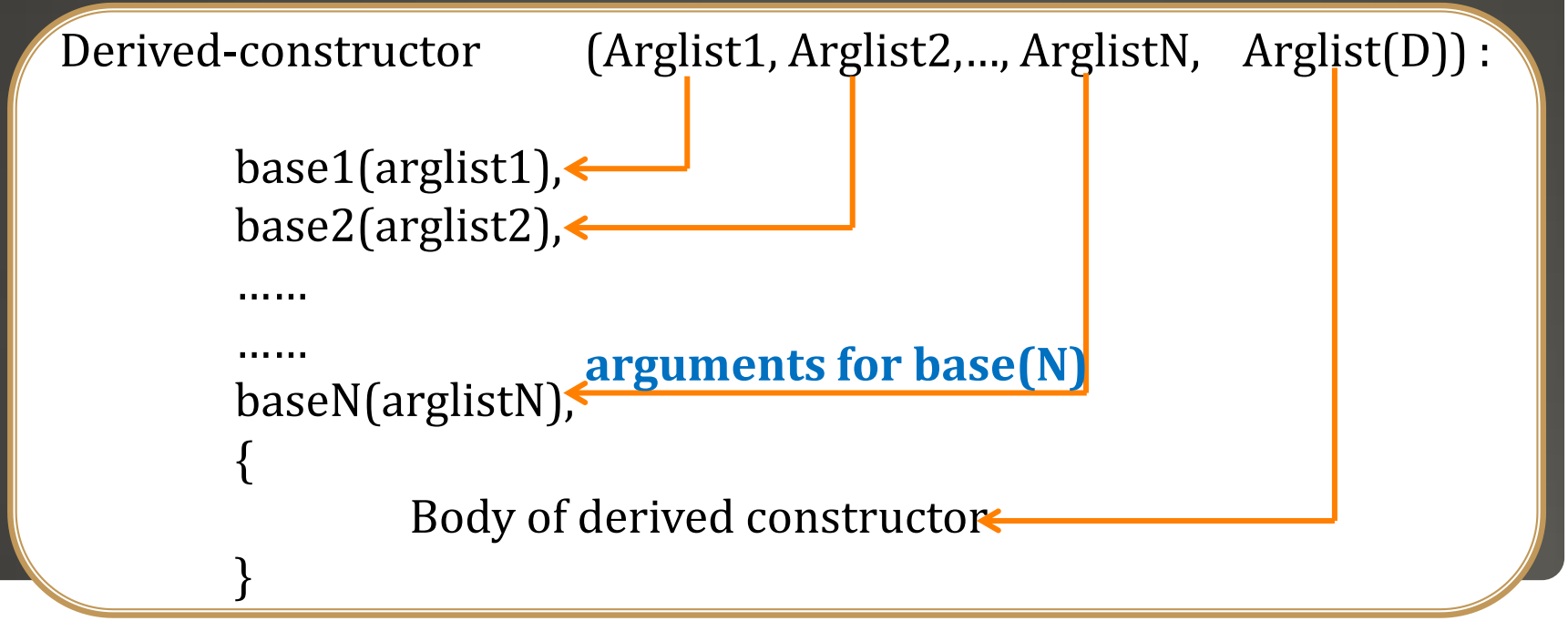
```
Derived-constructor      (Arglist1, Arglist2,..., ArglistN,   Arglist(D)) :

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

arguments for base(N)

11 Constructors in Derived Classes

- *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.



11 Constructors in Derived Classes

- *Arglist1* through *arglistN* are the argument declaration for base constructor *base1* through *baseN*.
- *ArglistD* provides the parameters that are necessary to initialize the members of the derived class.

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

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

arguments for base(N)

11 Constructors in Derived Classes

- Example

```
D(int a1, int a2, float b1, float b2, int d1):
A(a1, a2),          /*call to constructor A */
B(b1, b2)           /*call to constructor B */
{
        d=d1;       //executes its own body
}
```

  - A(a1, a2) invokes the base constructor A().
  - B(b1, b2) invokes another base constructor B().
  - The constructor D() supplies the values for these four arguments.
  - The constructor D() has one argument of its own.
  - The constructor D() has a total of five arguments.

11 Constructors in Derived Classes

- Example

```
D(int a1, int a2, float b1, float b2, int d1):
A(a1, a2),          /*call to constructor A */
B(b1, b2)           /*call to constructor B */
{
        d=d1;       //executes its own body
}
```

  - D() may be invoked as follows

```
……
D objD(5, 12, 2.5, 7.54, 30);
……
```

  - These values are assigned to various parameters by the constructor D().

5	→	a1
12	→	a2
2.5	→	b1
7.54	→	b2
30	→	d1

11 Constructors in Derived Classes

- The constructors for virtual base classes are invoked before any non-virtual base classes.
- If there are multiple virtual base classes, they are invoked in the order in which they are declared.
- Any non-virtual bases are then constructed before the derived class constructor is executed.

11 Constructors in Derived Classes

- Execution of base class constructors

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

11 Constructors in Derived Classes

- Example (Program 8.7)
  - Illustrates how constructors are implemented when the classes are inherited.
  - 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.
  - Note that alpha(a) and beta(b) are function calls.
  - Therefore, the parameters should not include types.

11 Constructors in Derived Classes

- Another method supported by C++ to initialize the class object.
  - The method uses what is known as initialization list in the constructor function.

11 Constructors in Derived Classes

- Another method supported by C++ to initialize the class object.
  - It takes the form

```
constructor  (arglist) : initialization-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.

11 Constructors in Derived Classes

- Another method supported by C++ to initialize the class object.
  - It takes the form

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

  - The initialization-section is used to provide initial value to the base constructors and also to initialize its own class members.
  - This means that we can use either of the sections to initialize the data members of the constructors class.
  - The initialization section basically contains a list of initializations, known as initialization list, separated by commas.

11 Constructors in Derived Classes

- Example
  - The program initialize **a** to 2 and **b** to 6.
  - The data members are initialized by using the variable name followed by the initialization value enclosed in the parenthesis(like a function call).

```
class  XYZ
{
        int  a;
        int  b;
public:
        XYZ(int  i, int  j) : a(i), b(2 * j) { }
};

main( )
{
        XYZ  x(2, 3);
}
```

11 Constructors in Derived Classes

- Any of the parameters of the argument list may be used as the initialization value and the items in the list may be in any order.
  - For example, the constructor XYZ may also be written as

```
XYZ(int i, int j) : b(i), a(i + j) { }
```

  - In this case, **a** will be initialized to 5 and **b** to 2.

11 Constructors in Derived Classes

- Remember
  - The data members are initialized in the order of declaration, independent of the order in the initialization list.
  - This enables us to have statements such as

```
XYZ(int i, int j) : a(i), b(a * j) { }
```

  - Here **a** is initialized to 2 and **b** to 6.

11 Constructors in Derived Classes

- However, the following will not work:

```
XYZ(int i, int j) : b(i), a(b * j) { }
```

  - Because the value of **b** is not available to **a** which is to be initialized first.

- The following statement are also valid:

```
XYZ(int i, int j) : a(i) {b = j; }
XYZ(int i, int j) : {a = i ; b = j; }
```

- We can omit either section, if it is not needed.

11 Constructors in Derived Classes

- Example (Program 8.8)
  - Illustrates the use of initialization lists in the base and derived constructors.

12 Member Classes : Nesting of Classes

- Another way of inheriting properties of one class into another.
  - It takes a view that an object can be a collection of many other objects.
  - That is, a class can contain objects of other classes as its members.

12 Member Classes : Nesting of Classes

- Example

```
class  alpha{...};
class  beta{...};
class  gamma
{
        alpha  a;          //a is an object of alpha class
        beta  b;           //b is an object of beta class
        ......
};
```

  - All objects of **gamma** class will contain the objects **a** and **b**.
  - This kind of relationship is called ***containership*** or ***nesting***.

12 Member Classes : Nesting of Classes

- Creation of an object that contains another object is very different than the creation of an independent object.
  - An independent object is created by its constructor when it is declared with arguments.
  - A nested object is created in two stages.
    1. The member objects are created using their respective constructors.
    2. The other 'ordinary' members are created.
    - ➢ This means, constructors of all the member objects should be called before its own constructor body is executed.
    - ➢ This is accomplished using an initialization list in the constructor of the nested class.

12 Member Classes : Nesting of Classes

- Example
  - ***arglist*** is the list of arguments that is to be supplied when a **gamma** object is defined.
  - These parameters are used for initializing the members of gamma.

```
class  gamma
{
            ……
            alpha  a;            //a is object of alpha
            beta  b;             //b is object of beta
public:
           gamma(arglist): a(arglist1), b(arglist2)
            {
                                 //constructor body
            }
};
```

12 Member Classes : Nesting of Classes

- Example
  - ***arglist1*** is the argument list for the constructor of **a**
  - ***arglist2*** is the argument list for the constructor of **b**.
  - ***arglist 1*** and ***arglist2*** may or may not use the arguments from ***arglist***.

```
class  gamma
{
        ……
        alpha  a;          //a is object of alpha
        beta  b;           //b is object of beta
public:
       gamma(arglist): a(arglist1), b(arglist2)
        {
                           //constructor body
        }
};
```

12 Member Classes : Nesting of Classes

- Remember
  - **a**(*arglist1*) and **b**(*arglist2*) are function calls and therefore the arguments do not contain the data types.
  - They are simply variables or constants.
- Example

```
gamma (int  x, int  y, float  z) : a(x), b(x,z)
{
        Assignment section(for ordinary members)
}
```

12 Member Classes : Nesting of Classes

- We can use as many member objects as are required in a class.
  - For each member object we add a constructor call in the initializer list.
  - The constructors of the member objects are called in the order in which they are declared in the nested class.