8

Inheritance: Extending Classes

Key Concepts

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

8.1 Introduction

Reusability is yet 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.

Fortunately, 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. The mechanism of deriving a new class from an old one is called *inheritance (or derivation)*. The old class is referred to as the *base class* and the new one is called the *derived class or subclass*.

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base class, is called *single inheritance* and one with several base classes is called *multiple inheritance*. On the other hand, the traits of one class may be inherited by more than one class. This process is known as *hierarchical inheritance*. The mechanism of deriving a class from another 'derived class' is known as *multilevel inheritance*. Figure 8.1 shows various forms of inheritance that could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance. (Some authors show the arrow in opposite direction meaning "inherited from".)

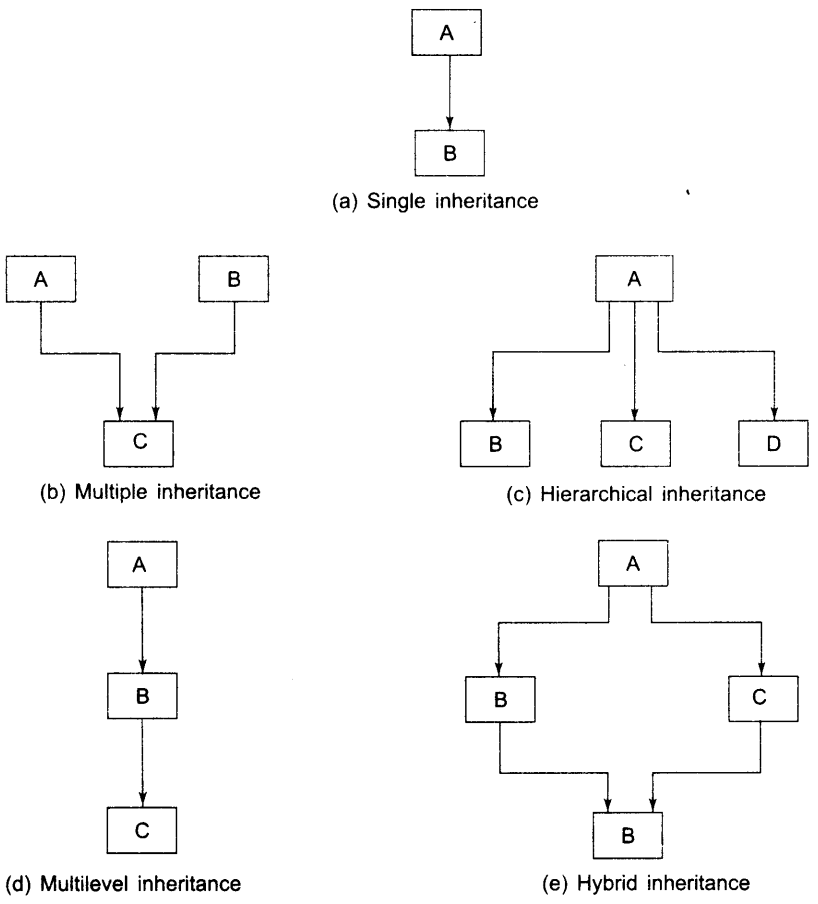


Fig. 8.1 ⇔ *Forms of inheritance*

8.2 Defining Derived Classes

A derived class can be defined by specifying its relationship with the base class in addition to its own details. 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**. The default visibility-mode is **private**. Visibility mode specifies whether the features of the base class are *privately derived or publicly derived.*

Examples:

```
class ABC: private XYZ         // private derivation
{
        members of ABC
};

class ABC: public XYZ          // public derivation
{
        members of ABC
};

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

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.

On the other hand, when the base class is *publicly inherited*, 'public members' of the base class become 'public members' of the derived class and therefore they are accessible to the objects of the derived class. 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.

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.

8.3 Single Inheritance

Let us consider a simple example to illustrate inheritance. 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.

SINGLE INHERITANCE : PUBLIC

```
#include <iostream>

using namespace std;

class B
{
      int a;                          // private; not inheritable
  public:
      int b;                          // public; ready for inheritance
      void get_ab();
      int  get_a(void);
      void show_a(void);
};

class D : public B                    // public derivation
{
       int c;
  public:
       void mul(void);
       void display(void);
};
//------------------------------------------------------------
void B :: get_ab(void)
{
       a = 5; b = 10;
}
int B :: get_a()
{
       return a;
}
void B :: show_a()
{
```

(Contd)

```
        cout << "a = " << a << "\n";
}
void D :: mul()
{
        c = b * get_a();
}
void D :: display()
{
        cout << "a = " << get_a() << "\n";
        cout << "b = " << b << "\n";
        cout << "c = " << c << "\n\n";
}
//----------------------------------------------------
int main()
{
        D d;

        d.get_ab();
        d.mul();
        d.show_a();
        d.display();

        d.b = 20;
        d.mul();
        d.display();

        return 0;
}
```

PROGRAM 8.1

Given below is the output of Program 8.1:

```
a = 5
a = 5
b = 10
c = 50

a = 5
b = 20
c = 100
```

The class **D** is a public derivation of the base class **B**. Therefore, **D** inherits all the **public** members of **B** and retains their visibility. Thus a **public** member of the base class **B** is also a public member of the derived class **D**. The **private** members of **B** cannot be inherited by **D**. The class **D**, in effect, will have more members than what it contains at the time of declaration as shown in Fig. 8.2.

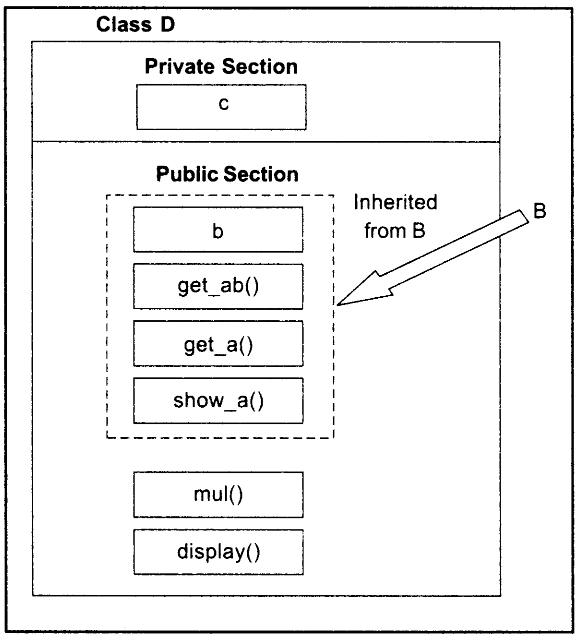


Fig. 8.2 ⇔ *Adding more members to a class (by public derivation)*

The program illustrates that the objects of class **D** have access to all the public members of **B**. Let us have a look at the functions **show_a()** and **mul()**:

```
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**.

Let us now consider the case of private derivation.

```
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 membership of the derived class **D** is shown in Fig. 8.3. In **private** derivation, the **public** members of the base class become **private** members of the derived class. Therefore, the objects of **D** can not have direct access to the public member functions of **B**.

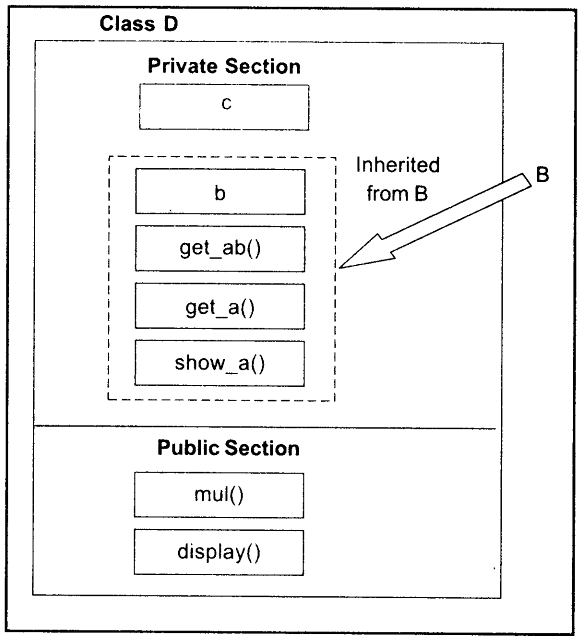


Fig. 8.3 ⇔ *Adding more members to a class (by private derivation)*

The statements such as

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

will not work. However, these functions can be used inside **mul()** and **display()** like the normal functions as shown below:

```
void mul()
{
     get_ab();
     c = b * get_a();
}

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

Program 8.2 incorporates these modifications for private derivation. Please compare this with Program 8.1.

SINGLE INHERITANCE : PRIVATE

```
#include <iostream>

using namespace std;

class B
{
       int a;          // private; not inheritable
     public:
       int b;          // public; ready for inheritance
       void get_ab();
       int  get_a(void);
       void show_a(void);
};

class D : private B             // private derivation
{
      int c;
```

(Contd)

```
   public:
      void mul(void);
      void display(void);
};

//----------------------------------------------------

void B :: get_ab(void)
{
     cout << "Enter values for a and b:";
     cin >> a >> b;
}

int B :: get_a()
{
     return a;
}

void B :: show_a()
{
      cout << "a = " << a << "\n";
}

void D :: mul()
{
      get_ab();
      c = b * get_a();        // 'a' cannot be used directly
}

void D :: display()
{
      show_a();                // outputs value of 'a'

      cout << "b = " << b << "\n"
           << "c = " << c << "\n\n";
}

//----------------------------------------------------

int main()
{
     D d;

     // d.get_ab();  WON'T WORK
     d.mul();
     // d.show_a();  WON'T WORK
     d.display();
```

(Contd)

```
    // d.b = 20;      WON'T WORK; b has become private
    d.mul();
    d.display();

    return 0;
}
```

PROGRAM 8.2

The output of Program 8.2 would be:

```
Enter values for a and b:5 10
a = 5
b = 10
c = 50
Enter values for a and b:12 20
a = 12
b = 20
c = 240
```

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.

8.4 Making a Private Member Inheritable

We have just seen how to increase the capabilities of an existing class without modifying it. We have also seen that a private member of a base class cannot be inherited and therefore it is not available for the derived class directly. 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 advantage of data hiding.

C++ provides a third *visibility modifier*, **protected**, 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. A class can now use all the three visibility modes as illustrated below:

```
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
};
```

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. A **protected** member, inherited in the **private** mode derivation, 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). Figure 8.4 is the pictorial representation for the two levels of derivation.

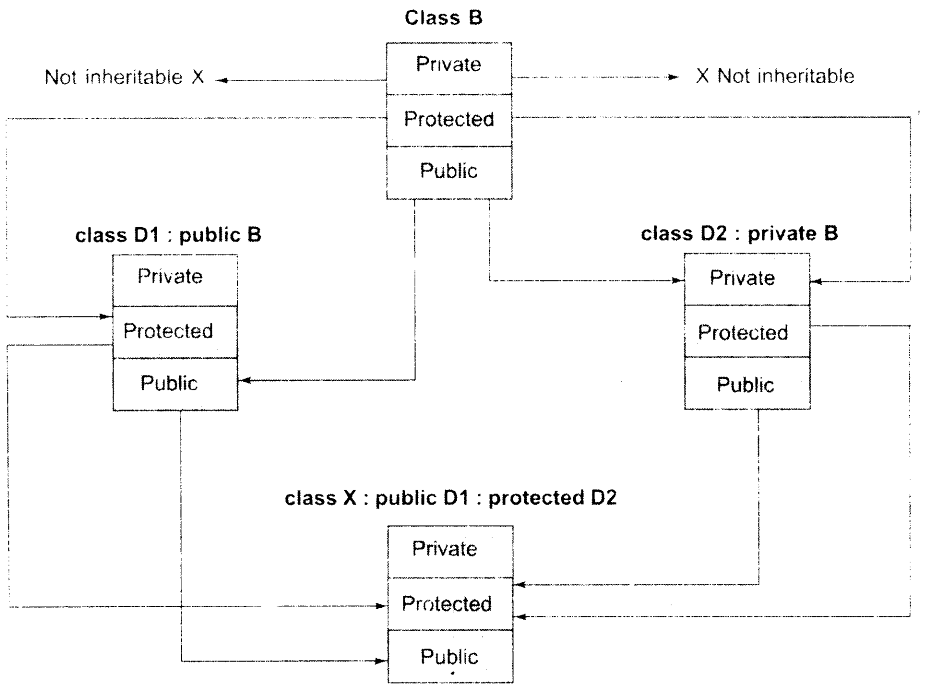


Fig. 8.4 ⇔ *Effect of inheritance on the visibility of members*

The keywords **private**, **protected**, and **public** may appear in any order and any number of times in the declaration of a class. For example,

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

is a valid class definition.

However, the normal practice is to use them as follows:

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

It is also possible to inherit a base class in **protected** mode (known as *protected derivation*). In protected derivation, both the **public** and **protected** members of the base class become **protected** members of the derived class. Table 8.1 summarizes how the visibility of base class members undergoes modifications in all the three types of derivation.

Now let us review the access control to the **private** and **protected** members of a class. What are the various functions that can have access to these members? They could be:

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.

While the friend functions and the member functions of a friend class can have direct access to both the **private** and **protected** data, the member functions of a derived class can directly access only the **protected** data. However, they can access the **private** data through the member functions of the base class. Figure 8.5 illustrates how the access control mechanism works in various situations. A simplified view of access control to the members of a class is shown in Fig. 8.6.

Table 8.1 *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

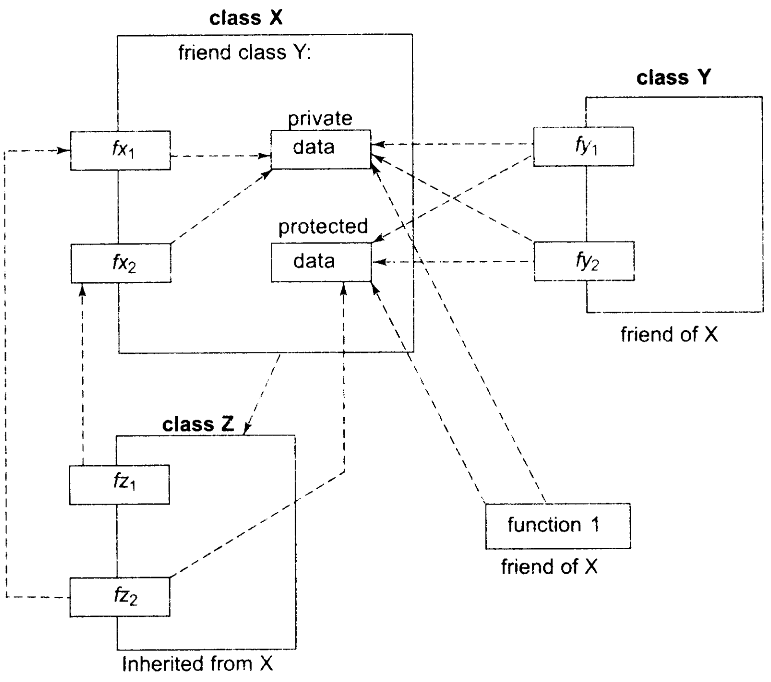


Fig. 8.5 ⇔ *Access mechanism in classes*

8.5 Multilevel Inheritance

It is not uncommon that a class is derived from another derived class as shown in Fig. 8.7. 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*.

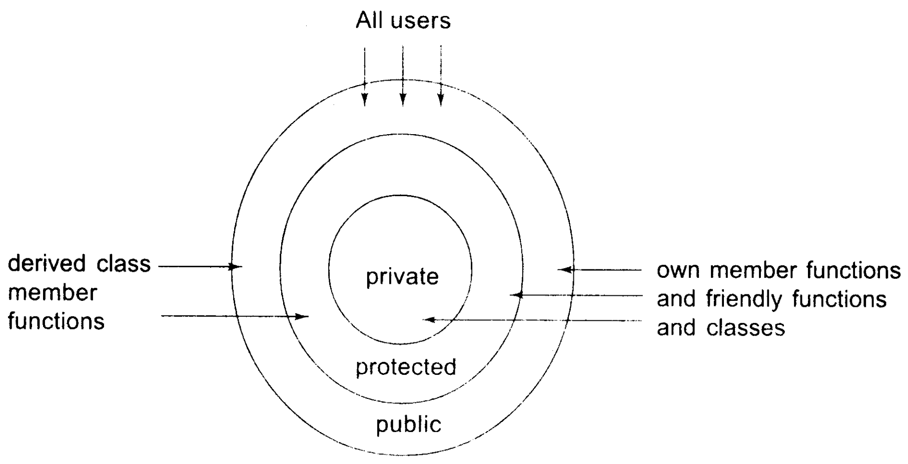

All users
derived class member functions
private
protected
public
own member functions and friendly functions and classes

Fig. 8.6 ⇔ *A simple view of access control to the members of a class*

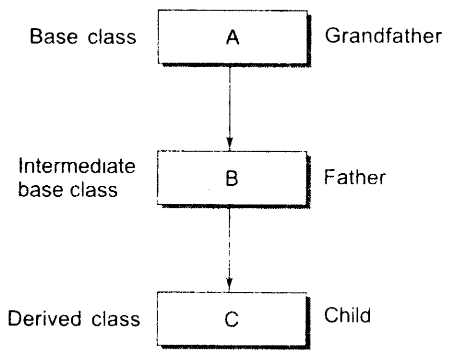


Fig. 8.7 ⇔ *Multilevel inheritance*

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.

Let us consider a simple 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 and 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. Example:

```
class student
{
        protected:
        int roll_number;
    public:
        void get_number(int);
        void put_number(void);
};
void student :: get_number(int a)
{
     roll_number = a;
}
void student :: put_number()
{
        cout << "Roll Number: " << roll_number << "\n";
}

class test : public student          // First level derivation
{
  protected:
      float sub1;
      float sub2;
  public:
      void get_marks(float, float);
      void put_marks(void);
};
void test :: get_marks(float x, float y)
{
      sub1 = x;
      sub2 = y;
}
void test :: put_marks()
{
      cout << "Marks in SUB1 = " << sub1 << "\n";
      cout << "Marks in SUB2 = " << sub2 << "\n";
}
class result : public test           // Second level derivation
{
      float total;              // private by default
  public:
      void display(void);
};
```

The class **result**, after inheritance from 'grandfather' through 'father', would contain the following members:

```
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
```

The inherited functions **put_number()** and **put_marks()** can be used in the definition of **display()** function:

```
void result :: display(void)
{
     total = sub1 + sub2;
     put_number();
     put_marks();
     cout << "Total = " << total << "\n";
}
```

Here is a simple **main()** program:

```
int main()
{
     result student1;              // student1 created
     student1.get_number(111);
     student1.get_marks(75.0, 59.5);
     student1.display();

  return 0;
}
```

This will display the result of **student1**. The complete program is shown in Program 8.3.

MULTILEVEL INHERITANCE

```
#include <iostream>

using namespace std;

class student
```

(Contd)

```
{
  protected:
         int roll_number;
  public:
         void get_number(int);
         void put_number(void);
};

void student :: get_number(int a)
{
         roll_number = a;
}

void student :: put_number()
{
         cout << "Roll Number: " << roll_number << "\n";
}

class test : public student            // First level derivation
{
  protected:
     float sub1;
     float sub2;
  public:
        void get_marks(float, float);
        void put_marks(void);
};

void test :: get_marks(float x, float y)
{
     sub1 = x;
     sub2 = y;
}

void test :: put_marks()
{
        cout << "Marks in SUB1 = " << sub1 << "\n";
        cout << "Marks in SUB2 = " << sub2 << "\n";
}

class result : public test       // Second level derivation
{
      float total;               // private by default
  public:
      void display(void);
};

void result :: display(void)
{
```

(Contd)

```
            total = sub1 + sub2;
            put_number();
            put_marks();
            cout << "Total = " << total << "\n";
    }

    int main()
    {
            result student1:          // student1 created

            student1.get_number(111);
            student1.get_marks(75.0, 59.5);

            student1.display();

        return 0;
    }
```

PROGRAM 8.3

Program 8.3 displays the following output:

```
Roll Number: 111
Marks in SUB1 = 75
Marks in SUB2 = 59.5
Total = 134.5
```

8.6 Multiple Inheritance

A class can inherit the attributes of two or more classes as shown in Fig. 8.8. This is known as *multiple inheritance*. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent and the intelligence of another.

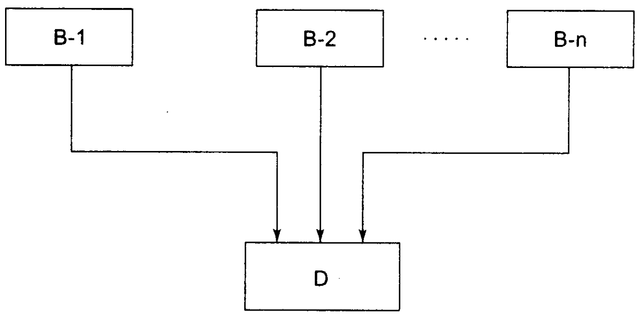


Fig. 8.8 ⇔ *Multiple inheritance*

The syntax of a derived class with multiple base classes is as follows:

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

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

Example:

```
class P : public M, public N
{
  public:
      void display(void);
};
```

Classes M and N have been specified as follows:

```
class M
{
  protected:
      int m;
  public:
      void get_m(int);
};
void M :: get_m(int x)
{
      m = x;
}
class N
{
  protected:
      int n;
  public:
      void get_n(int);
};
void N :: get_n(int y)
  {
        n = y;
    }
```

The derived class **P**, as declared above, would, in effect, contain all the members of **M** and **N** in addition to its own members as shown below:

```
class P
{
  protected:
      int m;                          // from M
      int n;                          // from N
  public:
      void get_m(int);                // from M
      void get_n(int);                // from N
      void display(void);             // own member
};
```

The member function display() can be defined as follows:

```
void P :: display(void)
{
     cout << "m = " << m << "\n";
     cout << "n = " << n << "\n";
     cout << "m*n =" << m*n << "\n";
};
```

The main() function which provides the user-interface may be written as follows:

```
main()
{
     P p;
     p.get_m(10);
     p.get_n(20);
     p.display();
}
```

Program 8.4 shows the entire code illustrating how all the three classes are implemented in multiple inheritance mode.

MULTIPLE INHERITANCE

```
#include <iostream>

using namespace std;

class M
{
  protected:
        int m;
  public:
        void get_m(int);
};

class N
{
  protected:
     int n;
  public:
     void get_n(int);
};

class P : public M, public N
{
  public:
     void display(void);
};

void M :: get_m(int x)
{
     m = x;
}

void N :: get_n(int y)
{
     n = y;
}

void P :: display(void)
{
     cout << "m = " << m << "\n";
     cout << "n = " << n << "\n";
     cout << "m*n = " << m*n << "\n";
}

int main()
{
```

(Contd)

```
          P p;

          p.get_m(10);
          p.get_n(20);
          p.display();

          return 0;
     }
```

PROGRAM 8.4

The output of Program 8.4 would be:

```
m = 10
n = 20
m*n = 200
```

Ambiguity Resolution in Inheritance

Occasionally, we may face a problem in using the multiple inheritance, when a function with the same name appears in more than one base class. Consider the following two classes.

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

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

Which **display()** function is used by the derived class when we inherit these two classes? We can solve this problem by defining a named instance within the derived class, using the class resolution operator with the function as shown below:

```
class P : public M, public N
```

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

We can now use the derived class as follows:

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

Ambiguity may also arise in single inheritance applications. For instance, consider the following situation:

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

In this case, 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. However, we may invoke the function defined in **A** by using the scope resolution operator to specify the class.

Example:

```
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;
    }
```

This will produce the following output:

```
B
A
B
```

8.7 Hierarchical Inheritance

We have discussed so far how inheritance can be used to modify a class when it did not satisfy the requirements of a particular problem on hand. Additional members are added through inheritance to extend the capabilities of a class. Another 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.

As an example, Fig. 8.9 shows a hierarchical classification of students in a university. Another example could be the classification of accounts in a commercial bank as shown in Fig. 8.10. All the students have certain things in common and, similarly, all the accounts possess certain common features.

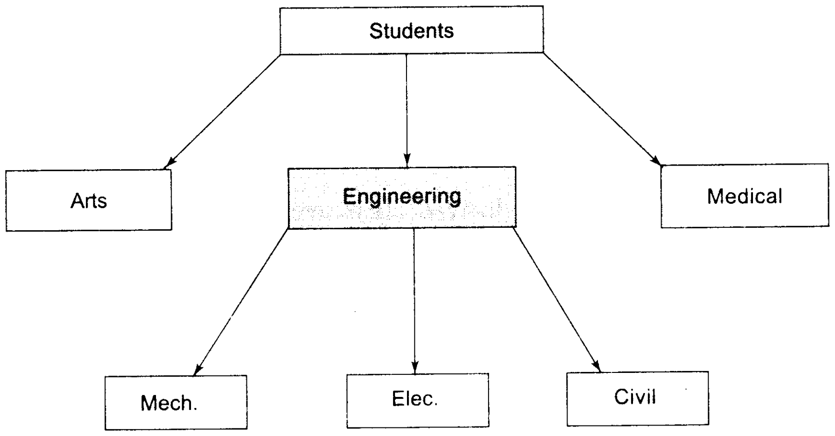


Fig. 8.9 ⇔ *Hierarchical classification of students*

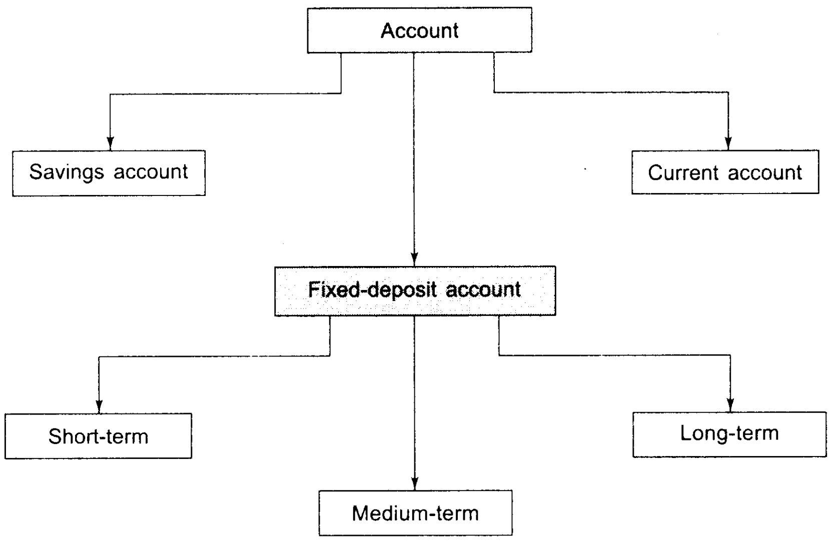


Fig. 8.10 ⇔ *Classification of bank accounts*

In C++, such problems can be easily converted into class hierarchies. 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.

8.8 Hybrid Inheritance

There could be situations where we need to apply two or more types of inheritance to design a program. For instance, consider the case of processing the student results discussed in Sec. 8.5. Assume that we have to give weightage for sports before finalising the results. The weightage for sports is stored in a separate class called **sports**. The new inheritance relationship between the various classes would be as shown in Fig. 8.11.

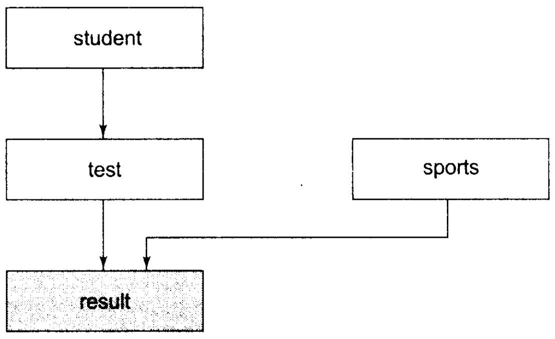


Fig. 8.11 ⇔ *Multilevel, multiple inheritance*

The **sports** class might look like:

```
class sports
{
  protected:
     float  score;
  public:
     void get_score(float);
     void put_score(void);
};
```

The result will have both the multilevel and multiple inheritances and its declaration would be as follows:

```
class result : public test, public sports
{
      .....
      .....
};
```

Where test itself is a derived class from student. That is

```
class test : public student
{
      .....
      .....
};
```

Program 8.5 illustrates the implementation of both multilevel and multiple inheritance.

HYBRID INHERITANCE

```
#include <iostream>

using namespace std;

class student
{
  protected:
      int  roll_number;
  public:
      void get_number(int a)
      {
           roll_number = a;
```

(Contd)

```
        }
        void put_number(void)
        {
              cout << "Roll No: " << roll_number << "\n";
        }
};

class test : public student
{
  protected:
        float part1, part2;
  public:
        void get_marks(float x, float y)
        {
              part1 = x;  part2 = y;
        }
        void put_marks(void)
        {
              cout << "Marks obtained: " << "\n"
                   << "Part1 = " << part1 << "\n"
                   << "Part2 = " << part2 << "\n";
        }
};

class sports
{
   protected:
        float score;
   public:
        void get_score(float s)
        {
              score = s;
        }
        void put_score(void)
        {
              cout << "Sports wt: " << score << "\n\n";
        }
};

class result : public test, public sports
{
        float total;
  public:
        void display(void);
```

(Contd)

```
};

void result :: display(void)
{
     total = partl + part2 + score;

     put_number();
     put_marks();
     put_score();

     cout << "Total Score: " << total << "\n";
}

int main()
{
     result student_1;
     student_1.get_number(1234);
     student_1.get_marks(27.5, 33.0);
     student_1.get_score(6.0);
     student_1.display();

     return 0;
}
```

PROGRAM 8.5

Here is the output of Program 8.5:

```
Roll No: 1234
Marks obtained:
Partl = 27.5
Part2 = 33
Sports wt: 6

Total Score: 66.5
```

8.9 Virtual Base Classes

We have just discussed a situation which would require the use of both the multiple and multilevel inheritance. Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved. This is illustrated in Fig. 8.12. The 'child' has two *direct base classes* 'parentl' and 'parent2' which themselves have a common base class 'grandparent'. The 'child' inherits the traits of 'grandparent' via two separate paths. It can also inherit directly as shown by the broken line. The 'grandparent' is sometimes referred to as *indirect base class*.

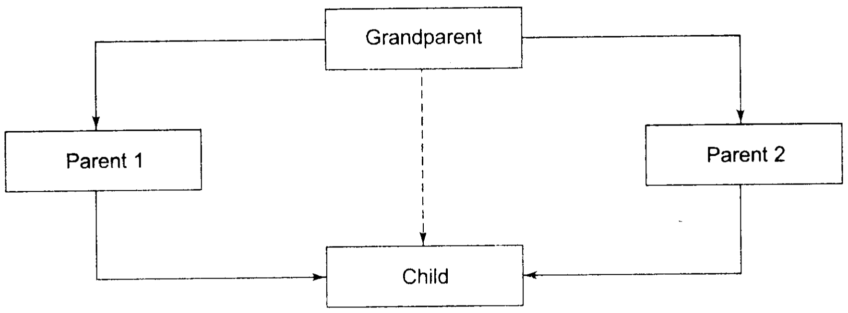


Fig. 8.12 ⇔ *Multipath inheritance*

Inheritance by the 'child' as shown in Fig. 8.12 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 the members inherited from 'grandparent'. This introduces ambiguity and should be avoided.

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 as shown below:

```
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
};
```

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.

note

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

For example, consider again the student results processing system discussed in Sec. 8.8. Assume that the class **sports** derives the **roll_number** from the class **student**. Then, the inheritance relationship will be as shown in Fig. 8.13.

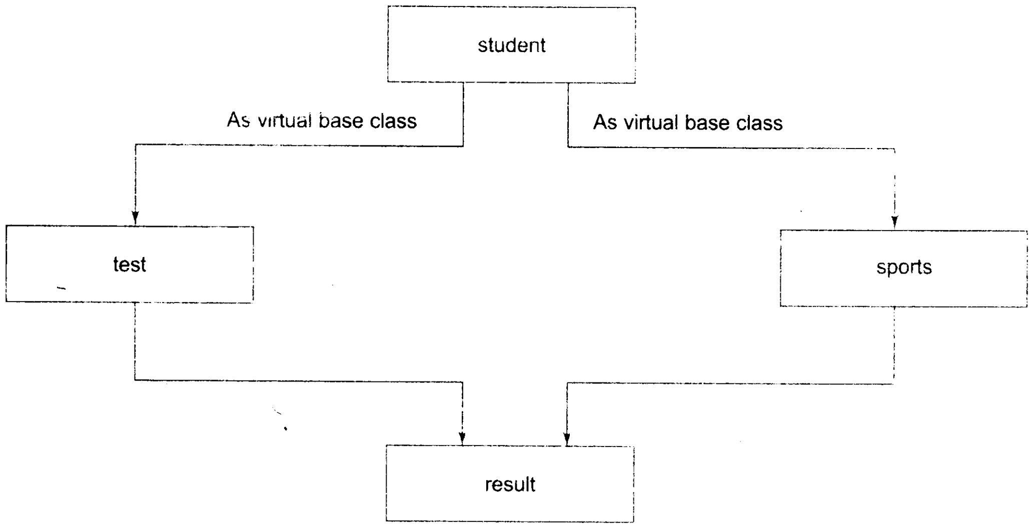


Fig. 8.13 ⇔ *Virtual base class*

A program to implement the concept of virtual base class is illustrated in Program 8.6.

VIRTUAL BASE CLASS

```
#include <iostream>

using namespace std;

class student
{
  protected:
    int roll_number;
  public:
    void get_number(int a)
    {
```

(Contd)

```
            roll_number = a;
    }
    void put_number(void)
    {
            cout << "Roll No: " << roll_number << "\n";
    }
};

class test : virtual public student
{
    protected:
      float part1, part2;
    public:
      void get_marks(float x, float y)
      {
           part1 = x;  part2 = y;
      }
      void put_marks(void)
      {
           cout << "Marks obtained: " << "\n"
                << "Part1 = " << part1 << "\n"
                << "Part2 = " << part2 << "\n";
      }
};

class sports : public virtual student
{
    protected:
      float score;
    public:
      void get_score(float s)
    {
           score = s;
    }
    void put_score(void)
    {
           cout << "Sports wt: " << score << "\n\n";
    }
};

class result : public test, public sports
{
      float total;
    public:
           void display(void);
};
```

(Contd)

```
void result :: display(void)
{
        total = part1 + part2 + score;

        put_number();
        put_marks();
        put_score();

        cout << "Total Score: " << total << "\n";
}

int main()
{
        result student_1;
        student_1.get_number(678);
        student_1.get_marks(30.5, 25.5);
        student_1.get_score(7.0);
        student_1.display();

        return 0;
}
```

PROGRAM 8.6

The output of Program 8.6 would be

```
Roll No: 678
Marks obtained:
Part1 = 30.5
Part2 = 25.5
Sport wt: 7

Total Score: 63
```

8.10 Abstract Classes

An *abstract class* is one that is not used to create objects. An abstract class 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.

8.11 Constructors in Derived Classes

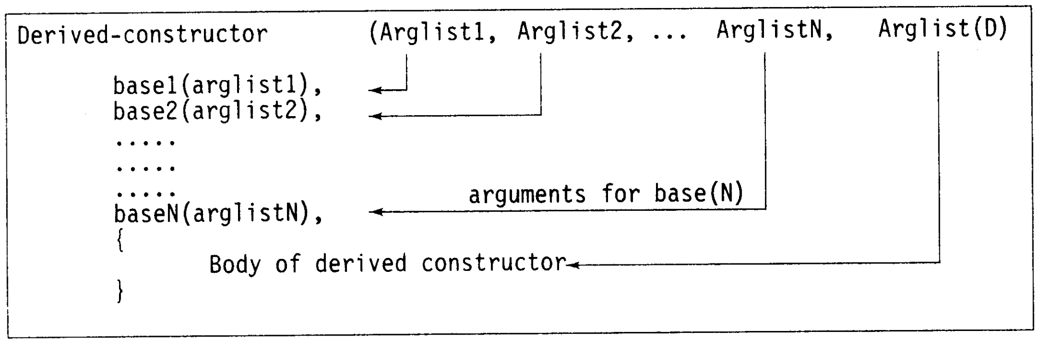
As we know, the constructors play an important role in initializing objects. We did not use them earlier in the derived classes for the sake of simplicity. One important thing to note here is that, as long as no base class constructor takes any arguments, the derived class need not have a constructor 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. 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.

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

Since the derived class takes the responsibility of supplying 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. How are they passed to the base class constructors so that they can do their job? C++ supports a special argument passing mechanism for such situations.

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.

The general form of defining a derived constructor is:



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

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

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

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. *Arglist1* through *ArglistN* are the argument declarations for base constructors *base1* through *baseN*. *ArglistD* provides the parameters that are necessary to initialize the members of the derived class.

Example:

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

A(a1, a2) invokes the base constructor **A()** and B(b1, b2) invokes another base constructor **B()**. The constructor **D()** supplies the values for these four arguments. In addition, it has one argument of its own. The constructor **D()** has a total of five arguments. **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()** as follows:

5	⟶	al
12	⟶	a2
2.5	⟶	bl
7.54	⟶	b2
30	⟶	dl

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. See Table 8.2.

Table 8.2 *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

Program 8.7 illustrates how constructors are implemented when the classes are inherited.

CONSTRUCTORS IN DERIVED CLASS

```
#include <iostream>

using namespace std;

class alpha
{
      int x;
    public:
      alpha(int i)
      {
           x = i;
           cout << "alpha initialized \n";
      }
      void show_x(void)
      { cout << "x = " << x << "\n"; }
};

class beta
{
      float y;
    public:
      beta(float j)
      {
           y = j;
           cout << "beta initialized \n";
      }
      void show_y(void)
      { cout << "y = " << y << "\n"; }
};

class gamma: public beta, public alpha
{
      int m, n;
    public:
      gamma(int a, float b, int c, int d):
           alpha(a), beta(b)
      {
           m = c;
           n = d;
           cout << "gamma initialized \n";
      }
```

(Contd)

```
    void show_mn(void)
    {
        cout << "m = " << m << "\n"
             << "n = " << n << "\n";
    }
};

int main()
{
    gamma g(5, 10.75, 20, 30);
    cout << "\n";
    g.show_x();
    g.show_y();
    g.show_mn();

    return 0;
}
```

PROGRAM 8.7

The output of Program 8.7 would be:

```
beta initialized
alpha initialized
gamma initialized

x = 5
y = 10.75
m = 20
n = 30
```

note

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. Also, note that **alpha(a)** and bet**a(b)** are function calls. Therefore, the parameters should not include types.

C++ supports another method of initializing the class objects. This method uses what is known as initialization list in the constructor function. This takes the following form:

```
constructor (arglist) : intialization-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. The part immediately following the colon is known as the *initialization section*. We can use this section to provide initial values 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 separated by commas. This list is known as *initialization list*. Consider a simple example:

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

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

This program will initialize **a** to 2 and **b** to 6. Note how the data members are initialized, just by using the variable name followed by the initialization value enclosed in the parenthesis (like a function call). 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. 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. Remember, **a** which has been declared first is initialized first and then its value is used to initialize **b**. 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 statements 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. Program 8.8 illustrates the use of initialization lists in the base and derived constructors.

[illegible] LIST IN CONSTRUCTORS

```
#include <iostream>

using namespace std;

class alpha
{
    int x;
  public:
    alpha(int i)
    {
         x = i;
         cout << "\n alpha constructed";
    }

    void show_alpha(void)
    {
         cout << " x = " << x << "\n";
    }
};

class beta
{
    float p, q;
  public:
    beta(float a, float b): p(a), q(b+p)
    {
         cout << "\n beta constructed";
    }
    void show_beta(void)
    {
         cout << " p = " << p << "\n";
         cout << " q = " << q << "\n";
    }
};
class gamma : public beta, public alpha
{
    int u,v;
  public:
```

(Contd)

```
          gamma(int a, int b, float c):
          alpha(a*2), beta(c,c), u(a)
          { v = b; cout << "\n gamma constructed"; }

          void show_gamma(void)
          {
          cout << " u = " << u << "\n";
          cout << " v = " << v << "\n";
          }
     };

     int main()
     {
          gamma g(2, 4, 2.5);

          cout << "\n\n Display member values " << "\n\n";

          g.show_alpha();
          g.show_beta();
          g.show_gamma();

          return 0;
     };
```

PROGRAM 8.8

The output of Program 8.8 would be:

```
beta constructed
alpha constructed
gamma constructed

Display member values

x = 4
p = 2.5
q = 5
u = 2
v = 4
```

note

The argument list of the derived construcctor **gamma** contains only three parameters **a**, **b** and **c** which are used to initialize the five data members contained in all the three classes.

8.12 Member Classes: Nesting of Classes

Inheritance is the mechanism of deriving certain properties of one class into another. We have seen in detail how this is implemented using the concept of derived classes. C++ supports yet another way of inheriting properties of one class into another. This approach 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 as shown below:

```
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*. 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. On the other hand, a nested object is created in two stages. First, the member objects are created using their respective constructors and then 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.

Example:

```
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
      }
};
```

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**. *arglist1* is the argument list for the constructor of **a** and *arglist2* is the argument list for the constructor of **b**. *arglist1* and *arglist2* may or may not use the arguments from *arglist*. 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 other members)
}
```

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.

SUMMARY

- ⇔ The mechanism of deriving a new class from an old class is called inheritance. Inheritance provides the concept of reusability. The C++ classes can be reused using inheritance.
- ⇔ The derived class inherits some or all of the properties of the base class.
- ⇔ A derived class with only one base class is called single inheritance.
- ⇔ A class can inherit properties from more than one class which is known as multiple inheritance.
- ⇔ A class can be derived from another derived class which is known as multilevel inheritance.
- ⇔ When the properties of one class are inherited by more than one class, it is called hierarchical inheritance.
- ⇔ A private member of a class cannot be inherited either in public mode or in private mode.
- ⇔ A protected member inherited in public mode becomes protected, whereas inherited in private mode becomes private in the derived class.
- ⇔ A public member inherited in public mode becomes public, whereas inherited in private mode becomes private in the derived class.
- ⇔ The friend functions and the member functions of a friend class can directly access the private and protected data.

```
class Student {
      char* name;
      int rollNumber;
private:
      Student() {
              name = "AlanKay";
              rollNumber = 1025;
      }
      void setNumber(int no) {
              rollNumber = no;
      }
      int getRollNumber() {
              return rollNumber;
      }
};

class AnualTest: Student {
      int mark1, mark2;
public:
      AnualTest(int m1, int m2)
              :mark1(m1), mark2(m2) {
      }
      int getRollNumber() {
              return Student::getRollNumber();
      }
};

void main()
{
      AnualTest test1(92, 85);
      cout << test1.getRollNumber();
}
```

8.2 Identify the error in the following program.

```
#include <iostream.h>
class A
{
public:
      A()
      {
```