```
    void f2()
    {
    // .. can refer to members a, b, c, and functions f1, f2, and f3
    }
  public:
    int c;
    void f3()
    {
    // .. can refer to members a, b, c, and functions f1, f2, and f3
    }
};
```

The data member a is *private* to class X and is accessible only to members of its own class, that is, member functions f1(), f2(), f3() can access a directly. However, statements outside and even member functions of the derived class are not allowed to access a directly. In addition, the member function f1() can be called only by other members of class X. The statements outside the class cannot call f1(), which is exclusively a private property of the class X.

The data member b and the member function f2() are *protected*. These members are accessible to other member functions of the class X and member functions in a derived class. However, outside the class, protected members have private status. The statements outside the class cannot directly access members b or f2() using the class.

The data member c and the member function f3() are *public*, and may be accessed directly by all the members of the class X, or by members in a derived class, or by objects of the class. *Public members* are always accessible to all users of the class.

The following statements,

```
X objx;            // objx is an object of class X
int d;         // temporary variable d
```

define the object objx of the class X and the integer variable d. The member access privileges are illustrated by the following statements referring to the object objx.

1. Accessing private members of the class X

```
d = objx.a; // Error: 'X::a' is not accessible
objx.f1();  // Error: 'X::f1()' is not accessible
```

Both the statements are invalid because the private members of a class are inaccessible to the object objx.

2. Accessing protected members of the class X

```
d = objx.b; // Error: 'X::b' is not accessible
objx.f2();  // Error: 'X::f2()' is not accessible
```

Both the statements are invalid because the protected members of a class are inaccessible since they are private to the class X.

3. Accessing public members of the class X

```
d = objx.c; // OK
objx.f3();  // OK
```

Both the statements are valid because the public members of a class are accessible to statements outside the scope of the class.

The program bag.cpp uses the access modifier *protected* to hold data members, instead of using the *private* access specifier. It indicates that the protected members are inheritable to derived classes. However, they have the same status as private members in the base class.

```
// bag.cpp: Bag into which fruits can be placed
#include <iostream.h>
enum boolean { FALSE, TRUE };
// Maximum number of items that a bag can hold
const int MAX_ITEMS = 25;
class Bag
{
   protected:                       // Note: not private
      int contents[MAX_ITEMS];      // bag memory area
      int ItemCount;                // Number of items present in a bag
   public:
      Bag()    // no-argument constructor
      {
         ItemCount = 0;    // When you purchase a bag, it will be empty
      }
      void put( int item ) // puts item into bag
      {
         contents[ ItemCount++ ] = item; // item into bag, counter update
      }
      boolean IsEmpty()    // 1, if bag is empty, 0, otherwise
      {
         return ItemCount == 0 ? TRUE : FALSE;
      }
      boolean IsFull()     // 1, if bag is full, 0, otherwise
      {
         return ItemCount == MAX_ITEMS ? TRUE : FALSE;
      }
      boolean IsExist( int item );
      void show();
};
// returns 1, if item is in bag, 0, otherwise
boolean Bag::IsExist( int item )
{
   for( int i = 0; i < ItemCount; i++ )
      if( contents[i] == item )
         return TRUE;
   return FALSE;
}
// display contents of a bag
void Bag::show()
{
   for( int i = 0; i < ItemCount; i++ )
      cout << contents[i] << " ";
   cout << endl;
}
```

```
void main()
{
   Bag bag;
   int item;
   while( TRUE )
   {
      cout << "Enter Item Number to be put into the bag <0-no item>: ";
      cin >> item;
      if( item == 0 )   // end of an item, break
         break;
      bag.put( item );
      cout << "Items in Bag: ";
      bag.show();
      if( bag.IsFull() )
      {
         cout << "Bag Full, no more items can be placed";
         break;
      }
   }
}
```

Run

```
Enter Item Number to be put into the bag <0-no item>: 1
Items in Bag: 1
Enter Item Number to be put into the bag <0-no item>: 2
Items in Bag: 1 2
Enter Item Number to be put into the bag <0-no item>: 3
Items in Bag: 1 2 3
Enter Item Number to be put into the bag <0-no item>: 3
Items in Bag: 1 2 3 3
Enter Item Number to be put into the bag <0-no item>: 1
Items in Bag: 1 2 3 3 1
Enter Item Number to be put into the bag <0-no item>: 0
```

In `main()`, the statement,

```
Bag bag;
```

creates the object `bag` and initializes the data member `ItemCount` to 0 through a constructor. The statement

```
bag.put( item );
```

stores the items into the `bag`. It does not check for the entry of duplicate items into a bag. Any item type can be placed any number of times into a bag and of course, without exceeding the limit or size of bag.

14.3 Derived Class Declaration

A derived class extends its features by inheriting the properties of another class, called base class and adding features of its own. The declaration of a derived class specifies its relationship with the base class in addition to its own features. The syntax of declaring a derived class is shown in Figure 14.2. Note that no memory is allocated to the declaration of a derived class, but memory is allocated when it is instantiated to create objects.

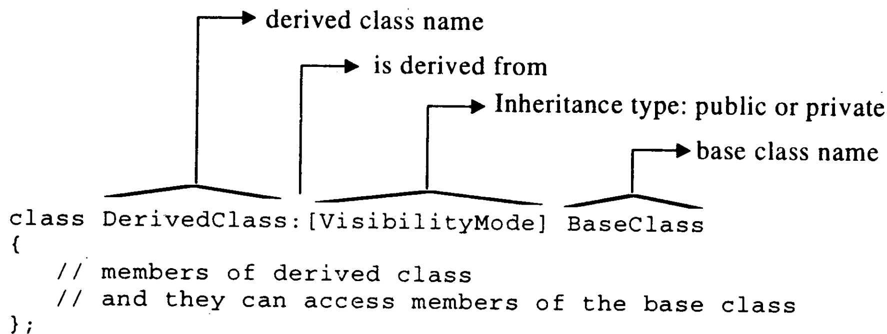


Figure 14.2: Syntax of derived class declaration

The derivation of `DerivedClass` from the `BaseClass` is indicated by the colon (:). The `VisibilityMode` enclosed within the square brackets implies that it is optional. The default visibility mode is `private`. If the visibility mode is specified, it must be either `public` or `private`. Visibility mode specifies whether the features of the base class are *publicly* or *privately inherited.*

The following are the three possible styles of derivation:

```
1.  class D: public B   // public derivation
    {
        // members of D
    };
2.  class D: private B  // private derivation
    {
        // members of D
    };
3.  class D: B          // private derivation by default
    {
        // members of D
    };
```

Inheritance of a base class with visibility mode `public`, by a derived class, causes `public` members of the base class to become `public` members of the derived class and the `protected` members of the base class become `protected` members of the derived class. Member functions and objects of the derived class can treat these derived members as though they are defined in the derived class itself. It is known that the public members of a class can be accessed by the objects of the class. Hence, the objects of a derived class can access public members of the base class that are inherited as public using the dot operator. However, `protected` members cannot be accessed with the dot operator. (See Figure 14.3.)

Inheritance of a base class with visibility mode `private` by a derived class, causes `public` members of the base class to become `private` members of the derived class and the `protected` members of the base class become `private` members of the derived class. Member functions and objects of a derived class can treat these derived members as though they are defined in the derived class with the `private` modifier. Thus objects of a derived class cannot access these members.

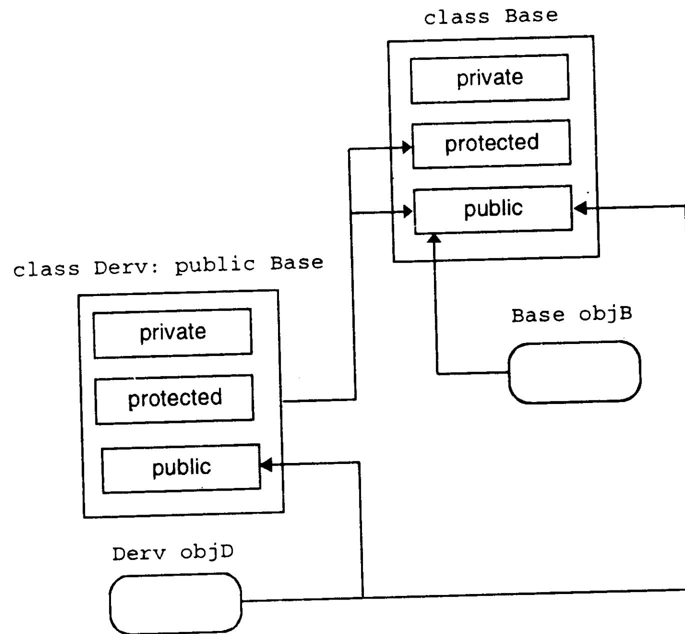


Figure 14.3: Access control of class members

Subsequent derivation of the classes from a *privately* derived class cannot access any members of the grand-parent class. The visibility of base class members undergoes modifications in a derived class as summarized in Table 14.1.

Base class visibility	Derived class visibility	
	Public derivation	Private derivation
private	Not Inherited (inherited base class members can access)	Not Inherited (inherited base class members can access)
protected	protected	private
public	public	private

Table 14.1: Visibility of class members

The private members of the base class remain private to the base class, whether the base class is inherited publicly or privately. They add to the data items of the derived class and they are not directly accessible to the member of a derived class. Derived classes can access them through the inherited member functions of the base class (see Figure 14.4).

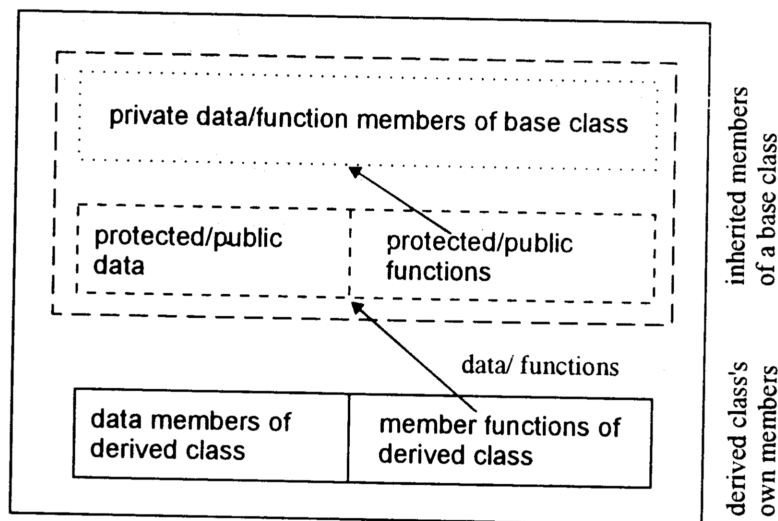


Figure 14.4: Members of derived class on inheritance

A Sample Program on Single Inheritance

A derived class may begin its existence with a copy of its base class members, including any other members inherited from more distantly related classes. *A derived class inherits data members and member functions, but not the constructor or destructor from its base class.* Recall that the program, `bag.cpp` discussed earlier has the class `Bag` and its instance, the `bag` object. A bag could be made empty or filled with items (fruits). The `Bag` class can be subjected to set operations such as union, intersection, etc.. It can be achieved by either modifying the `Bag` class or by deriving a new class called `Set` from the `Bag` class as shown in Figure 14.5.

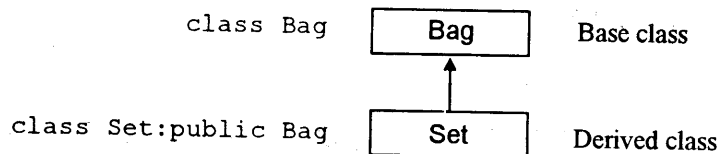


Figure 14.5: Inheritance of bag class

Considering that a large amount of time is spent in the development of the `Bag` class as well as in testing and debugging, it is not-at-all advisable to extend the `Bag` class by modifying as it will be impractical to rewrite or modify the original class especially in a large project when many programmers are involved. Also such a change would not be possible if the `Bag` class is a part of a commercial class library for which no source code is available to the user. Hence, rather than modifying `Bag`, a new class `Set` can be derived from it and the required new features can be added. It saves development cost, effort, and time.

The program `union.cpp` demonstrates the mechanism of extending the `Bag` class by using the feature of inheritance. In this case, a new class `Set` is derived from the existing class `Bag` without any modifications. A derived class `Set` inherits all the properties of the class `Bag` and extends itself by adding some of its own features to support set assignment and union operation.

```
// union.cpp: Union of sets. Set class by inheritance of Bag class
#include <iostream.h>
enum boolean { FALSE, TRUE };
const int MAX_ITEMS = 25;  // Maximum number of items that bag can hold
class Bag
{
  protected:                    // Note: not private
    int contents[MAX_ITEMS];    // bag memory area
    int ItemCount;              // number of items present in the bag
  public:
    Bag()                       // no-argument constructor
    {
      ItemCount = 0;    // When you purchase a bag, it will be empty
    }
    void put( int item )        // puts item into bag
    {
      contents[ ItemCount++ ] = item;  // item into bag,counter update
    }
    boolean IsEmpty()           // 1, if bag is empty, 0, otherwise
    {
      return ItemCount == 0 ? TRUE : FALSE;
    }
    boolean IsFull()            // 1, if bag is full, 0, otherwise
    {
      return ItemCount == MAX_ITEMS ? TRUE : FALSE;
    }
    boolean IsExist( int item );
    void show();
};
// returns 1, if item is in bag, 0, otherwise
boolean Bag::IsExist( int item )
{
  for( int i = 0; i < ItemCount; i++ )
    if( contents[i] == item )
      return TRUE;
  return FALSE;
}
// display contents of a bag
void Bag::show()
{
  for( int i = 0; i < ItemCount; i++ )
    cout << contents[i] << " ";
  cout << endl;
}
```

```
class Set: public Bag
{
   public:
      void add( int element )
      {
         if( !IsExist( element ) && !IsFull() )
            put( element );
            // element does not exist in set and it is not full
      }
      void read();
      void operator = (Set s1);
      friend Set operator + ( Set s1, Set s2 );
};
void Set::read()
{
   int element;
   while( TRUE )
   {
      cout << "Enter Set Element <0- end>: ";
      cin >> element;
      if( element == 0 )
         break;
      add( element );
   }
}
void Set::operator = ( Set s2 )
{
   for( int i = 0; i < s2.ItemCount; i++ )
      contents[i] = s2.contents[i];          // access Bag::contents
   ItemCount = s2.ItemCount;
}
Set operator + ( Set s1, Set s2 )
{
   Set temp;
   temp = s1;    // copy all elements of set s1 to temp
   // copy those elements of set s2 into temp, those not exist in set s1
   for( int i = 0; i < s2.ItemCount; i++ )
   {
      if( !s1.IsExist( s2.contents[i] ) ) //if element of s2 is not in s1
         temp.add( s2.contents[i] );      // copy the unique element
   }
   return( temp );
}
void main()
{
   Set s1, s2, s3;   // uses no-argument constructor of Bag class
   cout << "Enter Set 1 elements .." << endl;
   s1.read();
   cout << "Enter Set 2 elements .." << endl;
   s2.read();
   s3 = s1 + s2;
```

```
  cout << endl << "Union of s1 and s2 : ";
  s3.show();      // uses Bag::show() base class
}
```

Run

```
Enter Set 1 elements ..
Enter Set Element <0- end>: 1
Enter Set Element <0- end>: 2
Enter Set Element <0- end>: 3
Enter Set Element <0- end>: 4
Enter Set Element <0- end>: 0
Enter Set 2 elements ..
Enter Set Element <0- end>: 2
Enter Set Element <0- end>: 4
Enter Set Element <0- end>: 5
Enter Set Element <0- end>: 6
Enter Set Element <0- end>: 0
Union of s1 and s2 : 1 2 3 4 5 6
```

In the above program, the `Set` class has its own features to perform set union by using the member functions of `Bag`. The statement

```
class Set: public Bag
```

derives a new class `Set` from the base class `Bag`. The base class `Bag` is *publicly inherited* by the derived class `Set`. Hence, the members of `Bag` class, that are *protected* become `protected` and *public* become `public` in the derived class `Set`. The `Set` class can treat all the members of the `Bag` class as though they are its own.

The relationship between the base class `Bag` and the derived class `Set` has been depicted in Figure 14.5. Remember, that the arrow in the diagram, means *derived from*. The arrow indicates that the derived class Set refers to the data and member functions of the base class Bag, while the base class Bag has no access to the derived class Set.

Access to Constructor

In `main()`, the statement

```
Set s1, s2, s3;        // uses no-argument constructor of Bag class
```

creates three objects `s1, s2`, and `s3` of class `Set` and initializes the `ItemCount` variable to 0 in all the three objects, even though a constructor does not exist in the derived class `Set`. Thus, if a constructor is not defined in the derived class, C++ will use an appropriate constructor from the base class. In the above example, there is no constructor defined in the class `Set` and therefore, the compiler uses the *no-argument constructor*

```
Bag()                  // no-argument constructor
{
   ItemCount = 0;  // When you purchase a bag, it will be empty
}
```

defined in the `Bag` class. The use of the base class's constructor, in the absence of a constructor in the derived class, exhibits the true nature of inheritance that happens normally in day-to-day life.

Base Class Unchanged

It may be recalled that the base class remains unchanged even if other classes have been derived from it. In `main()` of the program `union.cpp`, objects of type `Bag` could be defined as,

```
Bag bag;        // object of the base class
```

Behaviors of such objects remain the same irrespective of the existence of a derived class such as `Set`.

It should also be noted that inheritance does not work in reverse. The base class and its objects do not know about any classes derived from it. In the example `union.cpp`, the objects of the base class `Bag`, cannot use the function, `operator+()` of the derived class `Set`.

Accessing Base Class Member Functions

The object `s3` of class `Set` also uses the function `show()` from the base class `Bag`. The statement

```
s3.show();     // uses Bag::show() base class
```

in the `main()`, refers to the function `show()`, which does not exist in the derived class `Set`. It is resolved by the compiler by selecting the member function `show()` defined in the base class `Bag`.

14.4 Forms of Inheritance

The derived class inherits some or all the features of the base class depending on the visibility mode and level of inheritance. Level of inheritance refers to the length of its (derived class) path from the root (top base class). A base class itself might have been derived from other classes in the hierarchy. Inheritance is classified into the following forms based on the levels of inheritance and interrelation among the classes involved in the inheritance process:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance
- Multipath Inheritance

The different forms of inheritance relationship is depicted in Figure 14.6. The pictorial representation of inheritance showing the interrelationship among the classes involved is known as the inheritance tree or class hierarchy. Base classes are represented at higher levels (top of the hierarchy, say root) and derived classes at the bottom of the hierarchy. The arrow directed from the derived class towards the base class, indicates that the derived class accesses features of the base class without modifying it, but not vice versa (Some use convention of representing the arrow in the opposite direction to indicate *inherited from or derived from*).

Single Inheritance: Derivation of a class from only one base class is called single inheritance. The sample program, `union.cpp`, discussed above falls under this category. Figure 14.6a depicts single inheritance.

Multiple Inheritance: Derivation of a class from several (two or more) base classes is called multiple inheritance. Figure 14.6b depicts multiple inheritance.

Hierarchical Inheritance: Derivation of several classes from a single base class i.e., the traits of one class may be inherited by more than one class, is called hierarchical inheritance. Figure 14.6c depicts hierarchical inheritance.

Multilevel Inheritance: Derivation of a class from another *derived class* is called multilevel inheritance. Figure 14.6d depicts multilevel inheritance.

Hybrid Inheritance: Derivation of a class involving more than one form of inheritance is known as hybrid inheritance. Figure 14.6e depicts hybrid inheritance.

Multipath Inheritance: Derivation of a class from other *derived classes*, which are derived from the same base class is called multipath inheritance. Figure 14.6f depicts multipath inheritance.

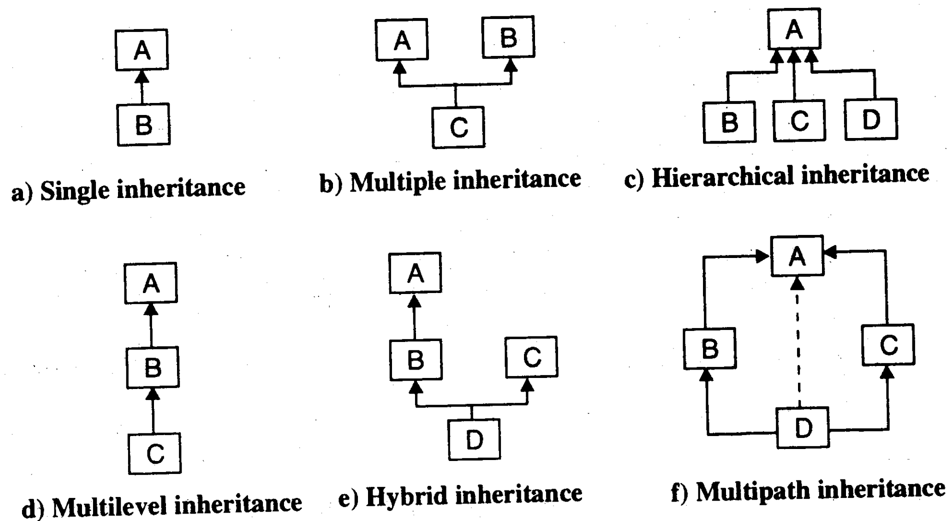


Figure 14.6: Forms of inheritance

14.5 Inheritance and Member Accessibility

The examples discussed earlier demonstrated the features of inheritance, which enhances the capabilities of the existing classes without modifying them. It is also observed that the `private` members of a base class, which cannot be inherited, are overcome by the use of access specifier `protected`. Accessibility refers to the authorization granted to access the members of a class by using an access specifier or modifier with or without inheritance. It defines the guidelines as to when a member function in the base class can be used by the objects of the derived class.

A protected member can be considered as a hybrid of a private and a public member. Like private members, protected members are accessible only to its class member functions and they are invisible outside the class. Like public members, protected members are inherited by derived classes and are also accessible to member functions of the derived class. The following rules are to be borne in mind while deciding whether to define members as private, protected, or public:

1. A private member is accessible only to members of the class in which the private member is declared. They cannot be inherited.
2. A private member of the base class can be accessed in the derived class through the member functions of the base class.

3. A protected member is accessible to members of its own class and to any of the members in a derived class.
4. If a class is expected to be used as a base class in future, then members which might be needed in the derived class should be declared protected rather than private.
5. A public member is accessible to members of its own class, members of the derived class, and outside users of the class.
6. The private, protected, and public sections may appear as many times as needed in a class and in any order. In case an inline member function refers to another member (data or function), that member must be declared before the inline member function is defined. Nevertheless, it is a normal practice to place the private section first, followed by the protected section and finally the public section.
7. The visibility mode in the derivation of a new class can be either private or public.
8. Constructors of the base class and the derived class are automatically invoked when the derived class is instantiated. If a base class has constructors with arguments, then their invocations must be explicitly specified in the derived class's initialization section. However, no-argument constructor need not be invoked explicitly. Remember that, constructors must be defined in the public section of a class (base and derived) otherwise, the compiler generates the error message: *unable to access constructor.*

Consider the following declarations of the base class to illustrate public and private inheritance:

```
class B    // base class
{
  private:
     int privateB;       // private member of base
  protected:
     int protectedB;     // protected member of base
  public:
     int publicB;        // public member of base
     int getBprivate()
     {
        return privateB;
     }
};
```

Public Inheritance

Consider the following declaration to illustrate the derivation of a new class D from the base class B publicly declared earlier:

```
class D: public B            // publicly derived class
{
  private:
     int privateD;
  protected:
     int protectedD;
  public:
     int publicD;
     void myfunc()
     {
        int a;
```

```
        a = privateB;      // Error: B::privateB is not accessible
        a = getBprivate(); // OK, inherited member accesses private data
        a = protectedB;    // OK
        a = publicB;       // OK
    }
};
```

The member function, `myfunc()` of the derived class `D` can access `protectedB` and `publicB` inherited from base class `B`. Since the class `B` is inherited as public by the derived class `D`, the status of members `protectedB`, `publicB`, `getBprivate()` remain unchanged in the derived class `D`. The statements

```
D objd;        // objd is a object of class D
int d;         // temporary variable d
```

define the object `objd` and the integer variable `d`. Consider the following statements referring to the object `objd`. Access to the `protected` member of the base class `B` in the statement,

```
d = objd.protectedB; // Error: 'B::protectedB' is not accessible
```

is invalid; `protectedB` has *protected* visibility status in class `D`. However the public member of the class `B` in the statement

```
d = objd.publicB;      // OK
```

is valid; `publicB` has public visibility status in class `D`. The inherited member function, `getBprivate()` in the statement

```
d = objd.getBprivate(); //OK,inherited member accesses private data
```

accesses a private data member of the base class.

In a subsequent derivation such as

```
class X : public D
{
   public:
      void g();
};
```

the member function `g()` in the derived class `X` may still access members `protectedB` and `publicB` and even retains the original protected and public status. Note that, private members of the classes `B` and `D` can be accessed through inherited members of the base class.

Private Inheritance

Consider the following declaration to illustrate the derivation of the new class `D` from the existing base class `B` privately:

```
class D: private B      // privately derived class
{
   private:
      int privateD;
   protected:
      int protectedD;
   public:
      int publicD;
```

```
    void myfunc()
    {
       int a;
       a = privateB;       // Error: B::privateB is not accessible
       a = getBprivate();  // OK,inherited member accesses private data
       a = protectedB;     // OK
       a = publicB;        // OK
    }
};
```

The member function `myfunc()` of the derived class `D` may access `protectedB` and `publicB` inherited from the base class `B`. Since, the base class `B` is inherited as the private base class of the derived class `D`, the status of members `protectedB`, `publicB`, and `getBprivate()` become `private` in the derived class `D`. The statements

```
D objd;         // objd is a object of class D
int d;          // temporary variable d
```

define the object `objd` and the integer variable `d`. Consider the following statements referring to the object `objd`. Access to the *protected* member of the base class `B` in the statement

```
d = objd.protectedB; // Error: B::protectedB is not accessible
```

is invalid; `protectedB` has `private` visibility status in the class `D`. Access to the public member of class `B` in the statement

```
d = objd.publicB;    //  Error: B::publicB is not accessible
```

is also invalid; `publicB` has `private` visibility status in the class `D`. The use of inherited member function, `getBprivate()` in the statement

```
d = objd.getBprivate();  // Error: getBprivate() is not accessible
```

is invalid; it has become a private member of the derived class `D`, however, a member function of the derived class can access—`myfunc()` accesses `getBprivate()` function.

In a subsequent derivation such as

```
class X : public D      // X is derived with D as base class
{
   public:
      void g();
};
```

the member function `g()` in `X` cannot access members `protectedB` and `publicB` since these members have gained `private` visibility status in class `D`. However, they (including private members of the classes B and D) can be accessed through inherited members of the base class.

Member Functions Accessibility

The various categories of functions which have access to the private and protected members could be any of the following:

- a member function of a class
- a member function of a derived class
- a friend function of a class
- a member function of a friend class

Function Type	Access directly to		
	Private	Protected	Public
Class Member	Yes	Yes	Yes
Derived class member	No	Yes	Yes
Friend	Yes	Yes	Yes
Friend class member	Yes	Yes	Yes

Table 14.2: Access control to class members

The friend functions and member functions of a friend class have direct access to both the private and protected members of a class. A member function of a class has access to all the members of its own class, be it private, protected, or public. The member functions of a derived class can directly access only the `protected` or `public` members; however they can access the `private` members through the member functions of the base class. Table 14.2 and Figure 14.7 summarizes the scope of access in various situations.

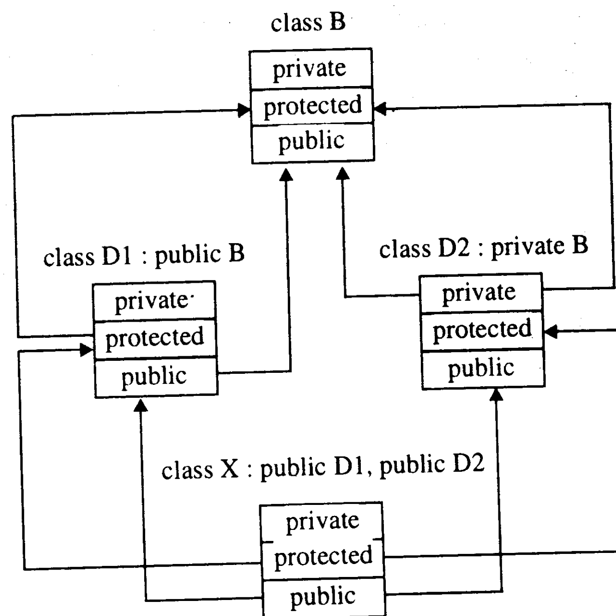


Figure 14.7: Access mechanism in classes

14.6 Constructors in Derived Classes

The constructors play an important role in initializing an object's data members and allocating required resources such as memory. The derived class need not have a constructor as long as the base class has a no-argument constructor. However, if the base class has constructors with arguments (one or more), then it is *mandatory* for the derived class to have a constructor and pass the arguments to the base class constructor. In the application of inheritance, objects of the derived class are usually created instead of the base class. Hence, it makes sense for the derived class to have a constructor and pass arguments to the constructor of the base class. When an object of a derived class is created, the constructor of the base class is executed first and later the constructor of the derived class.

The following examples illustrate the order of invocation of constructors in the base class and the derived class.

1. No-constructors in the base class and derived class

When there are no constructors either in the base or derived classes, the compiler automatically creates objects of classes without any error when the class is instantiated.

```
// cons1.cpp: No-constructors in base class and derived class
#include <iostream.h>
class B      // base class
{
   // body of base class, without constructors
};
class D: public B    // publicly derived class
{
   // body of derived base class, without constructors
   public:
   void msg()
   {
      cout << "No constructors exists in base and derived class" << endl;
   }
};
void main()
{
   D objd; // base constructor
   objd.msg();
}
```

Run

```
No constructors exists in base and derived class
```

2. Constructor only in the base class

```
// cons2.cpp: constructor in base class only
#include <iostream.h>
class B      // base class
{
   public:
```

```
  B()
  {
    cout << "No-argument constructor of the base class B is executed";
  }
};
class D: public B    // publicly derived class
{
  public:
};
void main()
{
  D obj1; // accesses base constructor
}
```

Run

```
No-argument constructor of the base class B is executed
```

3. Constructor only in the derived class

```
// cons3.cpp: constructors in derived class only
#include <iostream.h>
class B      // base class
{
  // body of base class, without constructors
};
class D: public B    // publicly derived class
{
  // body of derived base class, without constructors
  public:
  D()
  {
    cout << "Constructos exists in only in derived class" << endl;
  }
};
void main()
{
  D objd;          // accesses derived constructor
}
```

Run

```
Constructos exists in only in derived class
```

4. Constructor in both base and derived classes

```
// cons4.cpp: constructor in base and derived classes
#include <iostream.h>
class B      // base class
{
  public:
```

```
    B()
    {
    cout<<"No-argument constructor of the base class B executed first\n";
    }
};
class D: public B    // publicly derived class
{
  public:
    D()
    {
    cout<<"No-argument constructor of the derived class D executed next";
    }
};
void main()
{
   D objd;  // access both base constructor
}
```

Run

```
No-argument constructor of the base class B executed first
No-argument constructor of the derived class D executed next
```

5. Multiple constructors in base class and a single constructor in derived class

```
// cons5.cpp: multiple constructors in base and single in derived classes
#include <iostream.h>
class B     // base class
{
   public:
      B() { cout << "No-argument constructor of the base class B"; }
      B(int a) { cout <<"One-argument constructor of the base class B"; }
};
class D: public B    // publicly derived class
{
   public:
      D( int a )
      { cout << "\nOne-argument constructor of the derived class D"; }
};
void main()
{
   D objd( 3 );
}
```

Run

```
No-argument constructor of the base class B
One-argument constructor of the derived class D
```

6. Constructor in base and derived classes without default constructor

The compiler looks for the no-argument constructor by default in the base class. If there is a constructor in the base class, the following conditions must be met:

- The base class must have a no-argument constructor

- If the base class does not have a default constructor and has an argument constructor, they must be explicitly invoked, otherwise the compiler generates an error.

```
// cons6.cpp: constructor in base and derived class
#include <iostream.h>
class B     // base class
{
  public:
    B(int a) { cout << "One-argument constructor of the base class B";  }
};
class D: public B    // publicly derived class
{
  public:
    D( int a )
    { cout << "\nOne-argument constructor of the derived class D"; }
};
void main()
{
  D objd( 3 );
}
```

The compilation of the above program generates the following error:

```
Cannot find 'default' constructor to initialize base class 'B'
```

This error can be overcome by explicit invocation of a constructor of the base class as illustrated in the program `cons7.cpp`.

7. Explicit invocation in the absence of default constructor

```
// cons7.cpp: constructor in base and derived classes
#include <iostream.h>
class B     // base class
{
  public:
    B(int a)
    { cout << "One-argument constructor of the base class B"; }
};
class D: public B    // publicly derived class
{
  public:
    D( int a ) : B(a)
    { cout << "\nOne-argument constructor of the derived class D"; }
};
void main()
{
  D objd( 3 );
}
```

Run

```
One-argument constructor of the base class B
One-argument constructor of the derived class D
```

In the derived class D, the statement

```
D( int a ):B(a)
```

defines the derived class constructor D(int a) and calls the constructor of the base class using the special form :B(a). Here, the constructor of B is first invoked with an argument a specified in the constructor function D and then the constructor of D is invoked.

8. Constructor in a multiple inherited class with default invocation

```
// cons8.cpp: constructor in base and derived class, order of invocation
#include <iostream.h>
class B1    // base class
{
   public:
      B1() { cout << "\nNo-argument constructor of the base class B1"; }
};
class B2    // base class
{
   public:
      B2() { cout << "\nNo-argument constructor of the base class B2"; }
};
class D: public B2, public B1    // publicly derived class
{
   public:
      D()
      { cout << "\nNo-argument constructor of the derived class D"; }
};
void main()
{
   D objd;
}
```

Run

```
No-argument constructor of the base class B2
No-argument constructor of the base class B1
No-argument constructor of the derived class D
```

The statement

```
class D: public B2, public B1    // publicly derived class
```

specifies that the class D is derived from the base classes B1 and B2 in order. Hence, constructors are invoked in the order B2(), B1(), and D(); the constructors can be defined with or without arguments.

9. Constructor in a multiple inherited class with explicit invocation

```
// cons9.cpp: constructors with explicit invocation
#include <iostream.h>
class B1    // base class
{
   public:
      B1() {  cout << "\nNo-argument constructor of the base class B1"; }
};
```

```
class B2    // base class
{
   public:
     B2() {  cout << "\nNo-argument constructor of the base class B2"; }
};
class D: public B1, public B2
{
   public:
     D(): B2(), B1()    // explicit call to constructors
     {  cout << "\nNo-argument constructor of the derived class D"; }
};
void main()
{
   D objd;
}
```

Run

```
No-argument constructor of the base class B1
No-argument constructor of the base class B2
No-argument constructor of the derived class D
```

In the above program, the statement

```
class D: public B1, public B2    // publicly derived class
```

specifies that, the class D is derived from the base classes B1 and B2 in order. The statement

```
D(): B2(), B1()
```

in the derived class D, specifies that, the base class constructors must be called. However, the constructors are invoked in the order B1(), B2, and D, the order in which the base classes appear in the declaration of the derived class.

10. Constructor in base and derived classes in multiple inheritance

```
// cons10.cpp: constructor in base and derived classes, order of invocation
#include <iostream.h>
class B1    // base class
{
   public:
     B1() { cout << "\nNo-argument constructor of the base class B1"; }
};
class B2    // base class
{
   public:
     B2() { cout << "\nNo-argument constructor of a base class B2"; }
};
class D: public B1, virtual B2     // public B1, private virtual B2
{
   public:
     D(): B1(), B2()
     { cout << "\nNo-argument constructor of the derived class D"; }
};
```

```
void main()
{
   D objd;  // base constructor
}
```

Run

```
No-argument constructor of a base class B2
No-argument constructor of the base class B1
No-argument constructor of the derived class D
```

The statement

```
class D: public B1, virtual B2      // public B1, private virtual B2
```

specifies that the class `D` is derived from the base classes `B1` and `B2`. The statement

```
D():B1(), B2()
```

in the derived class `D`, specifies that, the base class constructors must be called. However, the constructors are invoked in the order `B2(),B1`, and `D()`, instead of the order in which base classes appear in the declaration of the derived class, since, the virtual base class constructors are invoked first followed by an orderly invocation of constructors of other classes.

11. Constructor in multilevel inheritance

```
// cons11.cpp: constructor in base and derived classes, order of invocation
#include <iostream.h>
class B    // base class
{
   public:
      B() { cout << "\nNo-argument constructor of a base class B"; }
};
class D1: public B    // derived class
{
   public:
      D1() { cout << "\nNo-argument constructor of a base class D1"; }
};
class D2: public D1   // publicly derived class
{
   public:
      D2()
      { cout << "\nNo-argument constructor of a derived class D2"; }
};
void main()
{
   D2 objd;  // base constructor
};
```

Run

```
No-argument constructor of a base class B
No-argument constructor of a base class D1
No-argument constructor of a derived class D2
```

The statement

```
class D2: public D1    // publicly derived class
```

specifies that the class `D2` is derived from the derived class `D1` of `B`. The constructors are invoked in the order `B()`, `D1()`, and `D2()` corresponding to the order of inheritance.

In the derived class, first the constructors of virtual base classes are invoked, second any non-virtual classes, and finally the derived class constructor. Table 14.3 shows the order of invocation of constructors in a derived class.

Method of Inheritance	Order of Execution
`class D: public B { ... };`	B( ): base constructor D( ): derived constructor
`class D: public B1, public B2 { ... };`	B1( ): base constructor B2( ): base constructor D( ): derived constructor
`class D: public B1, virtual B2 { .. };`	B2( ): virtual base constructor B1( ): base constructor D( ): derived constructor
`class D1: public B { ... }; class D2: public D1 { .. };`	B( ): super base constructor D1( ): base constructor D2( ): derived constructor

Table 14.3: Order of invocation of constructors

14.7 Destructors in Derived Classes

Unlike constructors, destructors in the class hierarchy (parent and child class) are invoked in the reverse order of the constructor invocation. The destructor of that class whose constructor was executed last, while building object of the derived class, will be executed first whenever the object goes out of scope. If destructors are missing in any class in the hierarchy of classes, that class's destructor is not invoked. The program `cons12.cpp` illustrates the order of invocation of constructors and destructors in handling instances of a derived class.

```
// cons12.cpp: order of invocation of constructors and destructors
#include <iostream.h>
class B1    // base class
{
   public:
      B1() { cout << "\nNo-argument constructor of the base class B1"; }
      ~B1()
      {
         cout << "\nDestructor in the base class B1";
      }
};
class B2    // base class
{
   public:
      B2() { cout << "\nNo-argument constructor of the base class B2"; }
      ~B2()
      {
         cout << "\nDestructor in the base class B2";
      }
};
class D: public B1, public B2    // publicly derived class
{
   public:
      D()
      { cout << "\nNo-argument constructor of the derived class D"; }
      ~D()
      {
         cout << "\nDestructor in the base class D";
      }
};
void main()
{
   D objd;
}
```

Run

```
No-argument constructor of the base class B1
No-argument constructor of the base class B2
No-argument constructor of the derived class D
Destructor in the base class D
Destructor in the base class B2
Destructor in the base class B1
```

Note that, in this program the constructors are invoked in the order of `B1()`, `B2()`, `D()` whereas, the destructors are invoked in the order of `D()`, `B2()`, `B1()`, which is in reverse order.

In case of dynamically created objects using the `new` operator, they must be destroyed explicitly by invoking the `delete` operator. More specialized class's (which are at the bottom of the hierarchy) destructors are called before a more general one (which are at the top of the hierarchy). As usual, no arguments can be passed to destructors, nor can any return type be declared.

14.8 Constructors Invocation and Data Members Initialization

In multiple inheritance, the constructors of base classes are invoked first, *in the order in which they appear in the declaration of the derived class*, whereas in the case of multilevel inheritance, they are executed *in the order of inheritance*. It is the responsibility of the derived class to supply initial values to the base class constructor, when the derived class objects are created. Initial values can be supplied either by the object of a derived class or a constant value can be mentioned in the definition of the constructor. The syntax for defining a constructor in a derived class is shown in Figure 14.8.

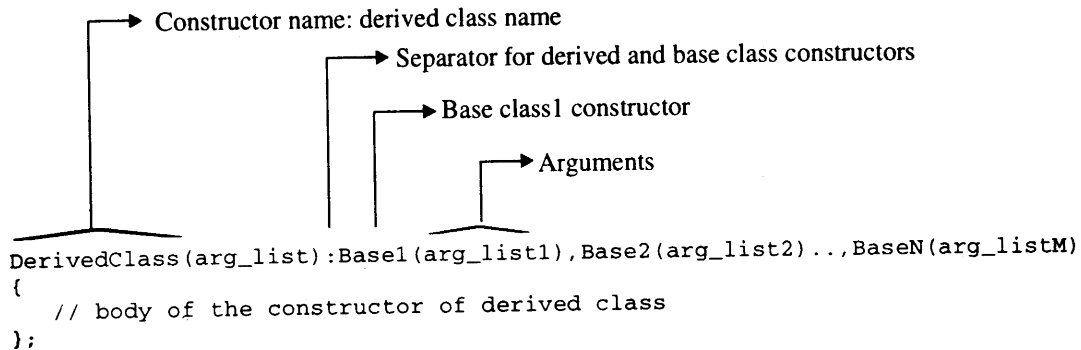


Figure 14.8: Syntax of derived class constructor

The parameters arg_list1, arg_list2, .., arg_listM are the list of arguments passed to the constructor or they can be any constant value those match with the arguments of the *constructor list* of base classes.

C++ supports another method of initializing the objects of classes through the use of the *initialization list* in the constructor function. It facilitates the initialization of data members by specifying them in the header section of the constructor. The general form of this method is shown in Figure 14.9.

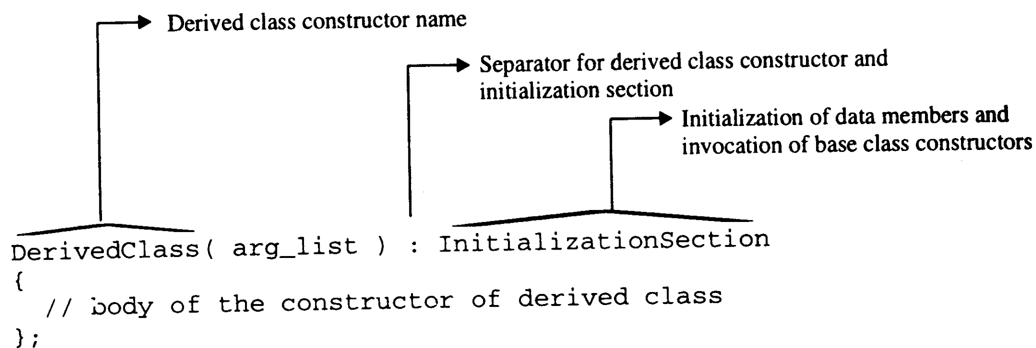


Figure 14.9: Syntax of initialization at derived class constructor

Data member initialization is represented by

```
DataMemberName( value )
```

The data members (DataMemberName) to be initialized are followed by the initialization value enclosed

in parentheses (resembles a function call). The `value` can be arguments of a constructor, expression or other data members. In the initialization section, any parameter of the argument-list can be used as an initialization value. The data member to be initialized must be a member of its own class. The program `cons14.cpp` illustrates the use of initialization section of the constructor. The following rules must be noted about the initialization and order of invocation of constructors:

- The initialization statements (in the initialization section) are executed in the order of definition of data members in the class.
- Constructors are invoked in the order of inheritance. However, the following rules apply when class is instantiated: first, the constructors of virtual base classes are invoked, second, any non-virtual classes, and finally, the derived class constructor.

```
// cons13.cpp: data members initialization through initialization-section
#include <iostream.h>
class B     // base class
{
   protected:
      int x, y;
   public:
      B(int a, int b): x(a), y(b) {} // x = a, y = b
};
class D: public B    // derived class
{
   private:
      int a, b;
   public:
      D(int p, int q, int r): a(p), B( p, q ), b(r) {}
   void output()
   {
      cout << "x = " << x << endl;
      cout << "y = " << y << endl;
      cout << "a = " << a << endl;
      cout << "b = " << b << endl;
   }
};
void main()
{
   D objb(5, 10, 15);
   objb.output();
}
```

Run

```
x = 5
y = 10
a = 5
b = 15
```

The constructor statement in the class `B`

```
B(int a, int b): x(a), y(b) {} // x = a, y = b
```

initializes the data members x and y to a and b respectively. The constructor statement in class D

```
D(int p, int q, int r): a(p), B( p, q ), b(r) {}
```

initializes the data members a and b to p and r respectively. It invokes the constructor B(int, int) of the base class B.

Consider the following declaration of class to illustrate the order of initialization:

```
class B     // base class
{
  private:
     int x, y;
  public:
     B(int a, int b): x(a), y(b) {} // x = a, y = b
};
```

Assume, the constructor of the class B is rewritten for illustration and object objb is defined as

```
B objb( 5, 10 );
```

The following examples illustrates the initialization of data members with different formats:

1. B(int a, int b): x(a), y(a+b)

The data member x is assigned the value a and y is assigned the value of the expression (a+b), i.e., x = 5 and y = (5+10) = 15.

2. B(int a, int b): x(a), y(x+b)

The data member x is assigned the value of a and y is assigned the value of the expression (x+b), i.e., x = 5 and y = (5+10) = 15. Note that the newly initialized data member can also be used in further initializations.

3. B(int a, int b): y(a), x(y+b)

It produces a wrong result, because, the statement which initializes the data member x is the first one to be executed (x is defined first data member in the class B). Hence the computation x(y+b) (i.e x = y+b) produces a wrong result because the data member y is not yet initialized. The program runtime.cpp illustrates this case. Thus, the order of data members in the initialization list is important.

```
// runtime.cpp: initialization through constructor header
#include <iostream.h>
class B
{
  private:
     int  x, y;
  public:
     B( int a, int b ): y(a), x(y+b) {} // No compilation, but  run-time
     void print()
     {
        cout << x << endl;
        cout << y << endl;
     }
}
```

```
void main()
{
   B b( 2, 3 );
   b.print();
}
```

Run

```
4211
2
```

The compiler converts the constructor of the class B into the following form:

```
B( int a, int b )
{
   x = (y+b);
   y = a;
}
```

In the above converted constructor, it should be noted that the statement

```
x = (y+b);
```

refers to the data member y which is still not initialized. Hence, the program produces the wrong result.

14.9 Overloaded Member Functions

The members of a derived class can have the same name as those defined in the base class. An object of a derived class refers to its own functions even if they are defined in both the base class and the derived class. The program cons14.cpp illustrates the overloaded data and member functions in the base and derived classes.

```
// cons14.cpp: overloaded members in base and derived classes
#include <iostream.h>
class B     // base class
{
  protected:
     int x;
     int y;
  public:
     B() {}
     void read()
     {
        cout << "X in class B ? ";
        cin >> x;
        cout << "Y in class B ? ";
        cin >> y;
     };
     void show()
     {
        cout << "X in class B = " << x << endl;
        cout << "Y in class B = " << y << endl;
     }
};
```

```
class D: public B    // publicly derived class
{
  protected:
     int y;
     int z;
  public:
  void read()
     {
       B::read();  // read base class data first
       cout << "Y in class D ? ";
       cin >> y;
       cout << "Z in class D ? ";
       cin >> z;
     };
     void show()
     {
       B::show();  // display base class data first
       cout << "Y in class D = " << y << endl;
       cout << "Z in class D = " << z << endl;
       cout<<"Y of B, show from D = "<< B::y;  //refers to y of class B
     };
};
void main()
{
   D objd;
   cout << "Enter data for object of class D .."  << endl;
   objd.read();
   cout << "Contents of object of class D .." << endl;
   objd.show();
}
```

Run

```
Enter data for object of class D ..
X in class B ? 1
Y in class B ? 2
Y in class D ? 3
Z in class D ? 4
Contents of object of class D ..
X in class B = 1
Y in class B = 2
Y in class D = 3
Z in class D = 4
Y of B, show from D = 2
```

In the derived class, there can also be functions with the same name as those in base class. It results in ambiguity. The compiler resolves the conflict by using the following rule:

If the same member (data/function) exists in both the base class and the derived class, the member in the derived class will be executed.

The above rule is true for derived classes. Objects of the base class do not know anything about the

derived class and will always use the base class members. Consider the statements

```
objd.read();
objd.show()
```

in function main(). In the first statement, objd, the object of a class D, invokes the read() function defined in the class D, instead of the read() function of the class B. Similarly, the function show() referenced by the objd uses the function defined in the class D.

Scope Resolution with Overriding Functions

The statement in class D

```
B::read();  // read base class data first
```

refers to the function read() defined in the base class B due to the use of scope resolution operation. Similarly, the statement

```
B::show();  // display base class data first
```

in the function show() of derived class D refers to the show() function of the base class B.

The statement

```
cout << "Y of B, show from D = " << B::y; // refers to y of class B
```

in the function show() has B::y, which refers to the data member defined in the base class B and not the one defined in the derived class D. These features of C++ demonstrates the creation of powerful functions using primitive functions. The general format of scope resolution for class members is shown in Figure 14.10.

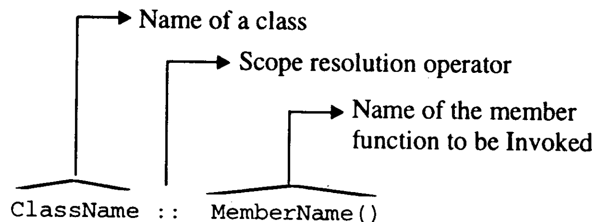


Figure 14.10: Syntax of member function access through scope resolution operator

For instance, as in the following statements

B::read() refers to the member function read() defined in the class B
B::y refers to the data member y defined in the class B

prefixing the class name to the member separated by scope resolution operator :: informs the compiler to call the member function specified in the class B.

Inheritance in the Stack Class

The various programs discussed so far, belong to the category of single Inheritance. Another practical example of inheritance is the stack, which is the most popularly used data-structure in building compilers, execution of recursive programs, allocating storage for local variables, and so on. The stack operates on the principle of Last-In-First-Out, popularly called LIFO policy. The last item entered into the stack is the first one to come out as shown in Figure 14.11.

The program `stack.cpp` has two classes, `Stack` as the base class and `MyStack` as the derived class of `Stack`. The base class `Stack` models a stack as a simple data storage device. It allows to push integers onto the stack and pop them off. However, it has a potential flaw. It does not check for the underflow or overflow that occurs in the manipulation of a stack. The program might not work since data would be placed in memory beyond the end of the `stack[]` array. Trying to pop too many items from the stack results in popping out meaningless data since, it would be reading data from memory locations outside the array.

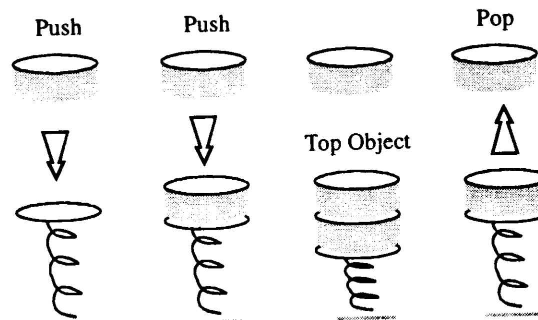


Figure 14.11: Stack operations

The potential flaw, in the class `Stack` can be overcome by developing a new class `MyStack`, a derived class inherited from the old stack class `Stack`. Objects of `MyStack` operate exactly the same way as those of Stack, except that it will issue a warning if an attempt is made to push an item onto a stack which is already full, or try to pop items out of an empty stack.

```
// stack.cpp: Overloading of functions in base and derived classes
#include <iostream.h>
const int MAX_ELEMENTS = 5; // maximum size of stack, you can change this
class Stack                 // base class
{
  protected:                // Note: cannot be private
    int stack[ MAX_ELEMENTS + 1 ]; // for stack[1]..stack[MAX_ELEMENTS]
    int  StackTop;          // It points to current stack top  element
  public:
    Stack()
    {
      StackTop = 0;         // Initially no elements in stack, stack empty
    }
    void push( int element )
    {
      ++StackTop;          // Update StackTop for new entry
      stack[StackTop] = element;    // put element into the stack
    }
    void pop( int &element )
    {
      element = stack[ StackTop ];
      --StackTop;          // Update StackTop to point to next element
    }
};
```

```
// derivation of a new class from the class Stack
class MyStack : public Stack
{
   public:
      int push( int element ) // return 1, if success, 0 otherwise
      {
         if( StackTop < MAX_ELEMENTS )    // if stack is not full
         {
            Stack::push( element );       // calls base class push
            return 1;     // push successful
         }
         cout << "Stack Overflow" << endl;
         return 0;       // stack overflow
      }
      int pop( int & element )   // return 1, if success, 0 otherwise
      {
         if( StackTop > 0 )      // if stack is not full
         {
            Stack::pop( element );        // calls base class pop
            return 1;    // pop successful
         }
         cout << "Stack Underflow" << endl;
         return 0;      // stack underflow
      }
};
void main()
{
   MyStack stack;
   int element;
   // push elements into Stack until it overflows
   cout << "Enter Integer data to put into the stack ..." << endl;
   do
   {
      cout << "Element to Push ? ";
      cin >> element;
   }
   while( stack.push( element ) ); // push and check for overflow
   // pop all elements from stack
   cout << "The Stack Contains ..." << endl;
   while( stack.pop( element ) )
      cout << "pop: " << element << endl;
}
```

Run

```
Enter Integer data to put into the stack ...
Element to Push ? 1
Element to Push ? 2
Element to Push ? 3
Element to Push ? 4
Element to Push ? 5
Element to Push ? 6
```

```
Stack Overflow
The Stack Contains ...
pop: 5
pop: 4
pop: 3
pop: 2
pop: 1
Stack Underflow
```

14.10 Abstract Classes

In order to exploit the potential benefits of inheritance, the base classes are improved or enhanced without modifications, which results in a derived class or inherited class. The objects created often are the instances of a derived class but not of the base class. The base class becomes just the foundation for building new classes and hence such classes are called *abstract base classes* or *abstract classes*. An abstract class is one that has no instances and is not designed to create objects. An abstract class is only designed to be inherited. It specifies an interface at a certain level of inheritance and provides a framework, upon which other classes can be built.

In the previous example (`stack.cpp`), the class `Stack` serves as a framework for building the derived classes and it is treated as a member of the derived class `MyStack`. The abstract class is the most important class and normally exists at the root of the hierarchy; it is a pathway to extending the system. Hence, the class `Stack` is sometimes loosely called as *abstract class* or *abstract base class*, meaning that no actual instances (objects) of these classes are created. However, abstract classes, in addition to inheritance, have more significance in connection with virtual functions, which will be discussed later in the chapter on *Virtual Functions*.

Abstract classes have other benefits. It provides a framework upon which other classes can be built and need not follow the trick of C (language, C++'s base class) programming. Most of the C programmers follow tricks of creating skeleton code and then copying and modifying the skeleton to create new functionality. One problem with skeleton code is if any modification is done to skeleton code, the changes must be propagated manually throughout the system -- an error prone process at best. In addition, it is difficult to find out whether bugs are in original skeleton or in modified system versions. By using abstract classes, interface can be changed which immediately propagate changes throughout the system with no errors. All changes made by the programmer in the derived classes are shown explicitly in the code, any bugs that show up are almost isolated in the new code.

14.11 Multilevel Inheritance

Derivation of a class from another *derived class* is called multilevel inheritance. It is very common in inheritance that a class is derived from a derived class as shown in Figure 14.12. The class `B` is the base class for the derived class `D1`, which in turn serves as a base class for the derived class `D2`. The class `D1` provides a link for the inheritance between `B` and `D2`, and is known as *intermediate* base class. The chain `B`, `D1`, `D2` is known as the *inheritance path*.

A derived class with multilevel inheritance is declared as follows:

```
class B { ... };            // Base class
class D1: public B() // D1 derived from B
```

```
class D2: public D1()       // D2 derived from D1
```

The multilevel inheritance mechanism can be extended to any number of levels.

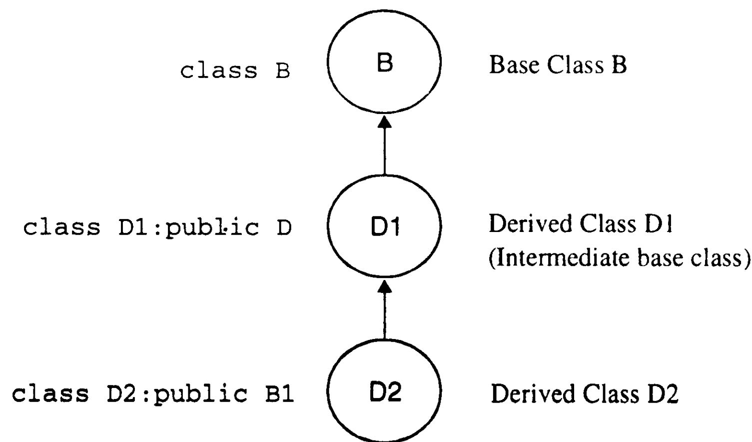


Figure 14.12: Multilevel Inheritance

The inheritance relation shown in Figure 14.13 is modeled in the program `exam.cpp`. It consists of three classes namely, person, student, and exam. Here, the class person is the base class, student is the intermediate base class, and exam is the derived class. The `student` class inherits the properties of person class whereas, the `exam` class inherits the properties of the `student` class (directly) and properties of the `person` class (indirectly).

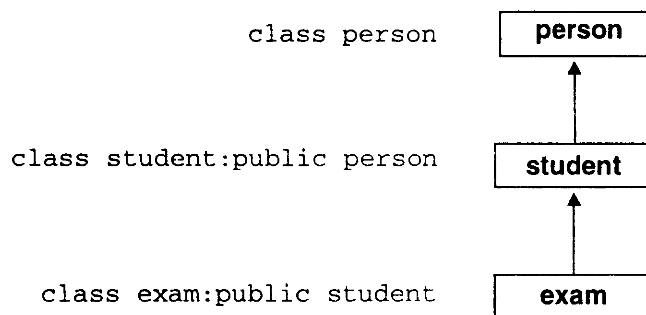


Figure 14.13: Multilevel inheritance

```
// exam.cpp: Models Examination database using Inheritance
#include <iostream.h>
const int MAX_LEN = 25;     // maximum length of name
class person
{
  private:                  // Note: cannot be referred by derived class
    char name[ MAX_LEN ];      // person name
    char sex;                  // person sex, M - male, F - female
    int age;                   // person age
```

```
  public:
     void ReadData()
     {
        cout << "Name ? ";
        cin >> name;
        cout << "Sex ? ";
        cin >> sex;
        cout << "Age ? ";
        cin >> age;
     }
     void DisplayData()
     {
        cout << "Name: " << name << endl;
        cout << "Sex : " << sex << endl;
        cout << "Age : " << age << endl;
     }
};
class student : public person  // publicly derived intermediate-base class
{
  private:
     int RollNo;       // student roll number in a class
     char branch[20];  // branch or subject student is studying
  public:
     void ReadData()
     {
        person::ReadData();  // uses ReadData of person class
        cout << "Roll Number ? ";
        cin >> RollNo;
        cout << "Branch Studying ? ";
        cin >> branch;
     }
     void DisplayData()
     {
        person::DisplayData();  // uses DisplayData of person class
        cout << "Roll Number: " << RollNo << endl;
        cout << "Branch: " << branch << endl;
     }
};
class exam: public student    // derived class
{
  protected:
     int Sub1Marks;
     int Sub2Marks;
  public:
     void ReadData()
     {
        student::ReadData();  // uses ReadData of student class
        cout << "Marks Scored in Subject 1 < Max:100> ? ";
        cin >> Sub1Marks;
        cout << "Marks Scored in Subject 2 < Max:100> ? ";
```

```
        cin >> Sub2Marks;
      }
      void DisplayData()
      {
        student::DisplayData(); // uses DisplayData of student class
        cout << "Marks Scored in Subject 1: " << Sub1Marks << endl;
        cout << "Marks Scored in Subject 2: " << Sub2Marks << endl;
        cout << "Total Marks Scored: " << TotalMarks();
      }
      int TotalMarks()
      {
        return Sub1Marks + Sub2Marks;
      }
};
void main()
{
   exam annual;
   cout << "Enter data for Student ..." << endl;
   annual.ReadData();   // uses exam::ReadData
   cout << "Student details ..." << endl;
   annual.DisplayData();   // exam::DisplayData
}
```

Run

```
Name ? Rajkumar
Sex ? M
Age ? 24
Roll Number ? 9
Branch Studying ? Computer-Technology
Marks Scored in Subject 1 < Max:100> ? 92
Marks Scored in Subject 2 < Max:100> ? 88
Student details ...
Name: Rajkumar
Sex :  M
Age : 24
Roll Number: 9
Branch: Computer-Technology
Marks Scored in Subject 1: 92
Marks Scored in Subject 2: 88
Total Marks Scored: 180
```

In `main()`, the statements

```
annual.ReadData();     // uses exam::ReadData
annual.DisplayData(); // exam::DisplayData
```

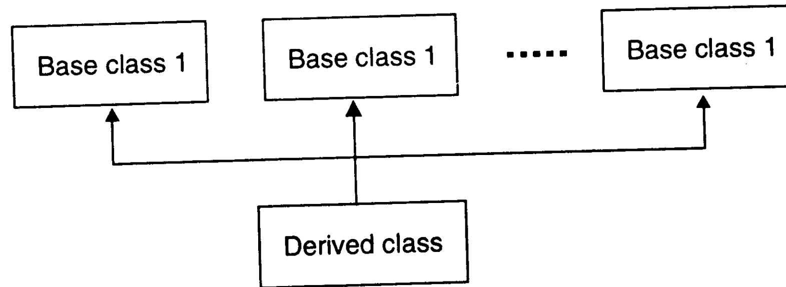
refer to the member functions of the class exam, since `annual` is its object. The statements in `ReadData()` function of the class exam

```
student::ReadData();     // uses ReadData of student class
student::DisplayData(); // uses DisplayData of student class
```

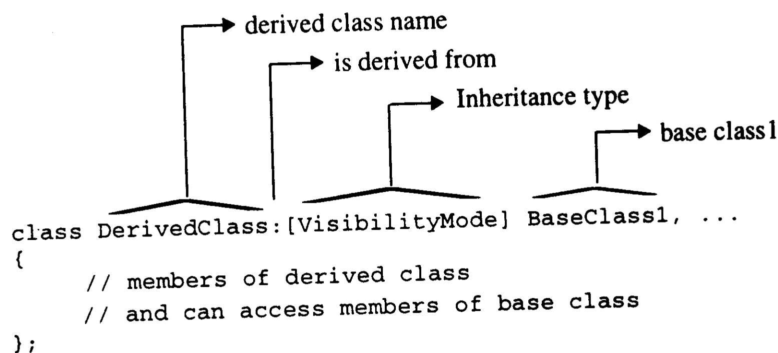
refers to the functions defined in the `student` class.

14.12 Multiple Inheritance

A class can be derived by inheriting the traits of two or more base classes. Multiple inheritance refers to the derivation of a class from several (two or more) base classes. It allows the combination of the features of several existing, tested, and well proven classes as a starting point for defining new classes. Multiple inheritance model is shown in Figure 14.14a and its syntax is shown in Figure 14.14b.



(a) Mutiple inheritance model




```
class DerivedClass:[VisibilityMode] BaseClass1, ...
{
      // members of derived class
      // and can access members of base class
};
```

(b) Syntax of mutiple inheritance

Figure 14.14: Mutiple inheritance

The default visibility mode is `private`. If visibility mode is specified, it must be either `public` or `private`. In multiple inheritance also, the inheritance of base classes with visibility mode `public`, implies that the `public` members of the base class become `public` members of the derived class and `protected` members of the base class become `protected` members of the derived class. Inheritance of base classes with visibility mode `private` causes both the `public` and `protected` members of the base class to become `private` members of the derived class. However, in both the cases `private` members of the base class are not inherited and they can be accessed through member functions of the base class.

The following declaration illustrates the concept of multiple inheritance:

```
class D: public B1, public B2  // multiple inheritance
{
  private:
```

```
      int privateD;
      void func1() {}
   protected:
      int protectedD; // D's own features
      void func2()
      { /* Null body function */ }
   public:
      int publicD;   // D's own features
      void func3();
};
```

The base classes B1 and B2 from which D is derived are listed following the colon in D's specification; they are separated by commas.

Constructors and Destructors

The constructors in base classes can be no-argument constructors or multiple argument constructors as discussed in the following sections.

No-Argument Constructor

Consider an example with the base classes A and B having constructors and the derived class C which has a no-argument constructor as in the program mul_inh1.cpp.

```
// mul_inh1.cpp: no-argument constructors in base and derived classes
#include <iostream.h>
class A  // base class1
{
   public:
      A()
      { cout << "a"; }
};
class B  // base class2
{
   public:
      B()
      { cout << "b"; }
};
class C: public A, public B   // derived class
{
   public:
      C()
      { cout << "c"; }
};
void main()
{
   C objc;
}
```

Run

abc

The base class constructors are always executed first, working from the first base class to the last

and finally through the derived class constructor. Since the derived class is declared as

```
class C: public A, public B
```

The constructor of the base class A is executed first, followed by the constructor of the class B and finally the constructor of the derived class C. Hence, the above program would print abc on the screen. If classes involved in multiple inheritance have destructors, they are invoked in the reverse order of the constructors invocation.

Passing Parameters to Multiple Constructors

Some or all parameters that are supplied to a derived class constructor may be passed to the base class(es) constructor. Therefore, if any base class constructor has one or more parameters, all classes derived from it must also have constructors with or without parameters.The program mul_inh2.cpp illustrates the base classes A and B having constructors with arguments; their derived class C must also have constructors.

```
// mul_inh2.cpp: constructors with arguments, must be called explicitly
#include <iostream.h>
class A  // base class1
{
   public:
      A( char c )
      { cout << c; }
};
class B  // base class2
{
   public:
      B( char b )
      { cout << b; }
};
class C: public A, public B   // derived class
{
   public:
      C( char c1, char c2, char c3): A( c1 ), B( c2 )
      { cout << c3; }
};
main()
{
   C objc( 'a', 'b', 'c');
}
```

Run

abc

In this case, the parameters c2 and c3 are passed to the constructors of the base classes A and B respectively. The arguments a, b and c are actually passed to the constructors of A, B, and C respectively even though they are parameters to the constructor of the class C. The constructors are executed in the order A, B, and C, hence, the above program would print abc on the screen. In general, parameters can be passed to the constructors of the base class as shown in the following syntax:

```
derived(parameter list):base1(parameter list1), base2(parameter list2), ...
```

The parameter lists of the base classes' constructors may contain any expression that has global scope (e.g., global constants, global variables, dynamically initialized global variables), as well as parameters that were passed to the derived class's constructor. The program `mul_inh3.cpp` illustrates the handling of constructors with arguments in the base class and the derived class.

```
// mul_inh3.cpp: constructors with arguments, if not called explicitly
#include <iostream.h>
class A  // base class1
{
   public:
      A( char c )
      { cout << c; }
};
class B  // base class2
{
   public:
      B( char b )
      { cout << b; }
};
class C: public A, public B
{
   public:
      C( char c1, char c2, char c3): B( c2 )
      { cout << c3; }
};
main()
{
   C objc( 'a', 'b', 'c');
}
```

The above program cannot be executed, since the following error is generated during compilation:

```
Error: Cannot find 'A::A()' to initialize base class in function C::C(char,
char, char)
```

If there are constructors in the base class and all of them are of type *constructors with arguments*, they must be explicitly specified in the derived class constructor. Otherwise, the compiler generates a compilation error. However, if a no-argument constructor also exists along with other constructors in base class, the compiler invokes the no-argument constructor as a default. Note that the base classes used in inheritance must preferably have a no-argument constructor.

Ambiguity in Member Access

Ambiguity is a problem that surfaces in certain situations involving multiple inheritance. Consider the following cases:

- Base classes having functions with the same name
- The class derived from these base classes is not having a function with the name as those of its base classes
- Members of a derived class or its objects referring to a member, whose name is the same as those in base classes

These situations create ambiguity in deciding which of the base class's function has to be referred. This problem is resolved using the scope resolution operator as shown in Figure 14.15. The program `mul_inh4.cpp` illustrates the same.

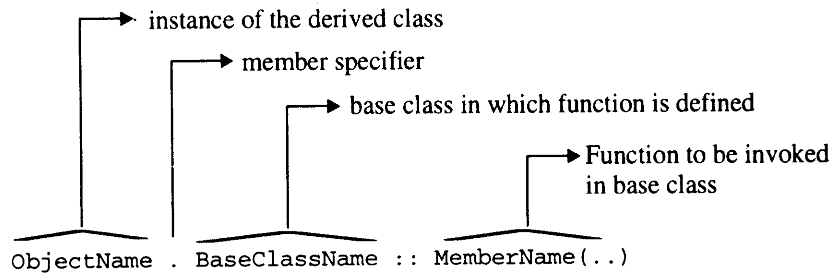


Figure 14.15: Syntax of handling ambiguity in multiple inheritance

```
// mul_inh4.cpp: overloaded functions in base classes
#include <iostream.h>
class A  // base class1
{
   char ch;     // private data, default
   public:
      A( char c )
      { ch = c; }
      void show()
      {
         cout << ch;
      }
};
class B  // base class2
{
   char ch;     // private data, default
   public:
      B( char b )
      { ch = b; }
      void show()
      {
         cout << ch;
      }
};
class C: public A, public B
{
   char ch;     // private data, default
   public:
      C( char c1, char c2, char c3): A( c1 ), B( c2 )
      {
         ch = c3;
      }
};
:
```

```
main()
{
   C objc( 'a', 'b', 'c');
   // objc.show();           // Error: Field 'show' is ambiguous in C
   cout << endl << "objc.A::show() = ";
   objc.A::show();
   cout << endl << "objc.B::show() = ";
   objc.B::show();
}
```

Run

```
objc.A::show() = a
objc.B::show() = b
```

In `main()`, the statement

```
objc.show(); // Error: Field 'show' is ambiguous in C
```

is ambiguous (whether to choose `A::show()` or `B::show()` ?) to the compiler resulting in a compilation error. It is resolved using the scope resolution operator as follows.

```
objc.A::show();
```

refers to the version of `show()` in the class `A`, while,

```
objc.B::show();
```

refers to the function in the class `B`. Thus, the scope resolution operator circumvents the ambiguity.

The program `mul_inh5.cpp` illustrates the base and derived classes, which have members with the same name.

```
// mul_inh5.cpp: overloaded functions in base and derived classes
#include <iostream.h>
class A  // base class1
{
   char ch;    // private data, default
   public:
      A( char c )
      {  ch = c;  }
      void show()
      {
         cout << ch;
      }
};
class B  // base class2
{
   char ch;    // private data, default
   public:
      B( char b )
      { ch = b; }
      void show()
      {
         cout << ch;
      }
};
```

```
class C: public A, public B
{
   char ch;     // private data, default
   public:
      C( char c1, char c2, char c3): A( c1 ), B( c2 )
      { ch = c3; }
      void show()
      {
         // show(); invokes C::show(), leading to infinite recursion
         A::show();
         B::show();
         cout << ch;
      }
};
main()
{
   C objc( 'a', 'b', 'c');
   cout << "objc.show() = ";
   objc.show();        // refers to show() defined in the derived class C
   cout << endl << "objc.C::show() = ";
   objc.C::show();
   cout << endl << "objc.A::show() = ";
   objc.A::show();
   cout << endl << "objc.B::show() = ";
   objc.B::show();
}
```

Run

```
objc.show() = abc
objc.C::show() = abc
objc.A::show() = a
objc.B::show() = b
```

In `main()`, the statements

```
objc.show();
objc.C::show();
```

refer to the same version of `show()` defined in the class `C`, while

```
objc.A::show();
```

refers to the version of `show()` defined in the class `A`, and

```
objc.B::show();
```

refers to the function defined in the class `B`. In the derived class `C`, statements in `show()`

```
A::show();
B::show();
```

refer to the functions defined in the classes `A` and `B` respectively.

Example on Multiple Inheritance

Consider a publishing company that publishes and markets books, whose activities are shown in Figure 14.16. Create a class `publication` that stores the `title` (string) and `price` (float) of a publication. Create another class `sales` that holds an array of three *float*'s so that it can record the sales of a

particular publication for the last three months. From these two classes, derive a new class called book that hold pages of integer type. Each of these classes should have the member functions getdata() and display().

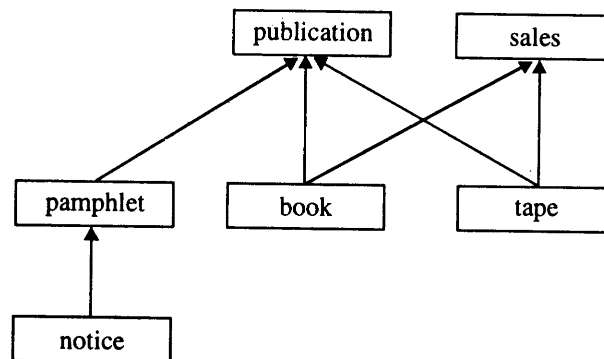


Figure 14.16: Multiple products company

From the publication and sales classes, derive the tape class, which adds playing time in minutes (type float). Create another class pamphlet from publication, which has no features of its own. Derive a class notice from pamphlet class having data members char whom[20] and member functions getdata() and putdata().

The program publish1.cpp models the class hierarchy shown in Figure 14.16. Note that, inheritance of the class publication by the classes, pamphlet, book, and tape illustrates the reuse of the code.

```
// publish1.cpp: Multiple products company modeling with multiple inheritance
#include <iostream.h>
class publication   // base class, appears as abstract class
{
   private:
      char title[40];   // name of the publication work
      float price;      // price of a publication
   public:
      void getdata()
      {
         cout << "\tEnter Title: ";
         cin >> title;
         cout << "\tEnter Price: ";
         cin >> price;
      }
      void display()
      {
         cout << "\tTitle = " << title << endl;
         cout << "\tPrice = ' << price << endl;
      }
};
class sales   // base class
{
   private:
```

```
    float PublishSales[3];//sales of a publication for the last 3 months
  public:
    void getdata();
    void display();
};
void sales::getdata()
{
  int i;

  for( i = 0; i < 3; i++ )
  {
    cout << "\tEnter Sales of " << i+1 << " Month: ";
    cin >> PublishSales[i];
  }
}
void sales::display()
{
  int i;
  int TotalSales = 0;

  for( i = 0; i < 3; i++ )
  {
    cout<<"\tSales of "<<i+1 << " Month = " << PublishSales[i] << endl;
    TotalSales += PublishSales[i];
  }
  cout << "\tTotal Sales = " << TotalSales << endl;
}
class book : public publication, public sales   // derived class
{
  private:
    int pages;  // number of pages in a book
  public:
    void getdata() // overloaded function
    {
      publication::getdata();
      cout << "\tEnter Number of Pages: ";
      cin >> pages;
      sales::getdata();
    }
    void display()
    {
      publication::display();
      cout << "\tNumber of Pages = " << pages << endl;
      sales::display();
    }
};
class tape : public publication, public sales   // derived class
{
  private:
    float PlayTime;   // playing time in minutes
```

```
   public:
      void getdata()
      {
         publication::getdata();
         cout << "\tEnter Playing Time in Minute: ";
         cin >> PlayTime;
         sales::getdata();
      }
      void display()
      {
         publication::display();
         cout << "\tPlaying Time in Minute = " << PlayTime << endl;
         sales::display();
      }
};
//for pamphlet class, sales class is not inherited, because, pamphlets
// cannot be sold, they are published for advertisement purpose
class pamphlet : public publication // derived class
{
};
class notice: public pamphlet   // derived, can access publics of pamphlet
{
   private:
      char whom[20];    // notice to all distributors
   public:
      void getdata()
      {
          pamphlet::getdata(); // intern calls getdata of publication
          cout << "\tEnter Type of Distributor: ";
          cin >> whom;
      }
      void display()
      {
         pamphlet::display(); // intern calls display of publication
         cout << "\tType of Distributor = " << whom << endl;
      }
};
void main()
{
   book book1;
   tape tape1;
   pamphlet pamp1;
   notice notice1;
   cout << "Enter Book Publication Data ..." << endl;
   book1.getdata();
   cout << "Enter Tape Publication Data ..." << endl;
   tape1.getdata();
   cout << "Enter Pamphlet Publication Data ..." << endl;
   pamp1.getdata();
   cout << "Enter Notice Publication Data ..." << endl;
   notice1.getdata();
```

```
    cout << "Book Publication Data ..." << endl;
    book1.display();
    cout << "Tape Publication Data ..." << endl;
    tape1.display();
    cout << "Pamphlet Publication Data ..." << endl;
    pamp1.display();
    cout << "Notice Publication Data ..." << endl;
    notice1.display();
}
```

Run

```
Enter Book Publication Data ...
    Enter Title: Microprocessor-x86-Programming
    Enter Price: 180
    Enter Number of Pages: 750
    Enter Sales of 1 Month: 1000
    Enter Sales of 2 Month: 500
    Enter Sales of 3 Month: 800
    Enter Tape Publication Data ...
    Enter Title: Love-1947
    Enter Price: 100
    Enter Playing Time in Minute: 10
    Enter Sales of 1 Month: 200
    Enter Sales of 2 Month: 500
    Enter Sales of 3 Month: 400
Enter Pamphlet Publication Data ...
    Enter Title: Advanced-Computing-95-Conference
    Enter Price: 10
Enter Notice Publication Data ...
    Enter Title: General-Meeting
    Enter Price: 100
    Enter Type of Distributor: Retail
Book Publication Data ...
    Title = Microprocessor-x86-Programming
    Price = 180
    Number of Pages = 705
    Sales of 1 Month = 1000
    Sales of 2 Month = 500
    Sales of 3 Month = 800
    Total Sales = 2300
Tape Publication Data ...
    Title = Love-1947
    Price = 100
    Playing Time in Minute = 10
    Sales of 1 Month = 200
    Sales of 2 Month = 500
    Sales of 3 Month = 400
    Total Sales = 1100
Pamphlet Publication Data ...
    Title = Advanced-Computing-95-Conference
```

```
   Price = 10
Notice Publication Data ...
   Title = General-Meeting
   Price = 100
   Type of Distributor = Retail
```

14.13 Hierarchical Inheritance

A well established method of program design is the hierarchical model, which can be modeled better using the concepts of inheritance. Many programming problems fall into this category. Hierarchical model follows a top down approach by breaking up a complex class into simpler constituent classes. In other words, in the hierarchical model, a complex class is conceptualized as being made up of simpler classes. Figure 14.17 illustrates the hierarchical classification of vehicles in a vehicle license department respectively. Hierarchical inheritance resembles the multilevel inheritance, in which only one derived class path is taken into consideration.

In C++, hierarchical programs can be easily converted into class hierarchies. The superclass (base class) includes the features that are common to all the subclasses (derived classes). A subclass is created by inheriting the properties of the base class and adding some of its own features. The subclass can serve as a superclass for the lower level classes again and so on. The program `vehicle.cpp` models the class hierarchy in C++ for the problem shown in Figure 14.17.

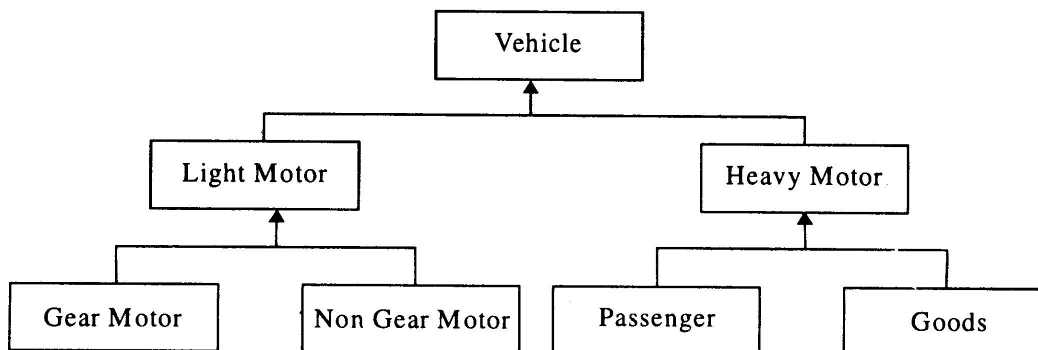


Figure 14.17: Classification of vehicles

```
// vehicle.cpp: Vehicle Database Hierarchical Model
#include <iostream.h>
const MAX_LEN= 25;   // length of string
class Vehicle
{
  protected:
     char name[ MAX_LEN];     // name of the vehicle
     int WheelsCount;         // number of wheels to vehicle
  public:
     void GetData()
     {
        cout << "Name of the Vehicle ? ";
```

```
            cin >> name;
            cout << "Wheels ? ";
            cin >> WheelsCount;
        }
        void DisplayData()
        {
            cout << "Name of the Vehicle : " << name << endl;
            cout << "Wheels : " << WheelsCount << endl;
        }
};
class LightMotor: public Vehicle
{
    protected:
        int SpeedLimit;
    public:
        void GetData()
        {
            Vehicle::GetData();
            cout << "Speed Limit ? ";
            cin >> SpeedLimit;
        }
        void DisplayData()
        {
            Vehicle::DisplayData();
            cout << "Speed Limit : " << SpeedLimit << endl;
        }
};
class HeavyMotor: public Vehicle
{
    protected:
        int LoadCapacity;    // load carrying capacity
        char permit[MAX_LEN]; // permits: state, country, international
    public:
        void GetData()
        {
            Vehicle::GetData();
            cout << "Load Carrying Capacity ? ";
            cin >> LoadCapacity;
            cout << "Permit Type ? ";
            cin >> permit;
        }
        void DisplayData()
        {
            Vehicle::DisplayData();
            cout << "Load Carrying Capacity : " << LoadCapacity << endl;
            cout << "Permit: " << permit << endl;
        }
};
```

```
class GearMotor: public LightMotor
{
   protected:
      int GearCount;
   public:
      void GetData()
      {
         LightMotor::GetData();
         cout << "No. of Gears ? ";
         cin >> GearCount;
      }
      void DisplayData()
      {
         LightMotor::DisplayData();
         cout << "Gears: " << GearCount << endl;
      }
};
class NonGearMotor: public LightMotor
{
   public:
      void GetData()
      {
         LightMotor::GetData();
      }
      void DisplayData()
      {
         LightMotor::DisplayData();
      }
};
class Passenger: public HeavyMotor
{
   protected:
      int sitting;
      int standing;
   public:
      void GetData()
      {
         HeavyMotor::GetData();
         cout << "Maximum Seats ? ";
         cin >> sitting;
         cout << "Maximum Standing ? ";
         cin >> standing;
      }
      void DisplayData()
      {
         HeavyMotor::DisplayData();
         cout << "Maximum Seats: " << sitting << endl;
         cout << "Maximum Standing: " << standing << endl;
      }
};
```