Inheritance

11

CHAPTER

CHAPTER OUTLINE

11.1 INTRODUCTION

Inheritance is one of the most useful and essential characteristics of object-oriented programming. The existing classes are the main components of inheritance. The new classes are created from existing ones. The properties of the existing classes are simply extended to the new classes. The new classes created by using such a method are known as `derived classes`, and the existing classes are known as `base classes`, as shown in Figure 11.1. The relationship between the base and derived classes is known as `kind of relationship`. The programmer can define new member variables and functions in the derived class. The base class remains unchanged. The object of the derived class can access members of the base as well as derived classes. On the other hand, the object of the base class cannot access members of the derived classes. The base class does not know about their subclasses.

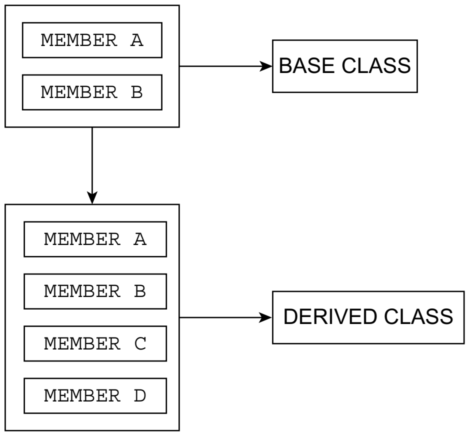


Fig. 11.1 Inheritances

11.2 REUSABILITY

`Reusability` means the reuse of properties of the base class in the derived classes. Reusability permits the reuse of members of the previous class. We can add extra features in the existing class. This is possible by creating a new class from the existing class. The new class will have the features of both the old and the derived new class. In Figure 11.1, the base class is reused. Reusability is achieved using inheritance. Inheritance and reusability are not different from each other. The outcome of inheritance is reusability.

In this chapter, we will deal with implementation of the mechanism of inheritance and understand its features by illustrating several examples. One can find the inheritance property in several languages, but inheritance property in C++ is slightly different. C++ enables an inheritance of all the properties of the previous class, and there is a flexibility to add new members in the derived class, which are not available in the parent class.

The base class is also called `superclass, parent,` or `ancestor`, and the derived class is called `subclass, child,` or `descendent`. It is also possible to derive a class from a previously derived class. A class can be derived from more than one class.

INHERITANCE

The procedure of creating a new class from one or more existing classes is termed *inheritance*.

11.3 ACCESS SPECIFIERS AND SIMPLE INHERITANCE

Before elaborating the inheritance mechanism, it is necessary to recall our memory for access rules studied in the Chapter 8 on *classes and objects*. We have studied the access specifiers private and public in detail and briefly studied the protected access specifiers.

The `public` members of a class can be accessed by an object directly outside the class. `Directly` means when objects access the data member without the member function of the class. The `private` members of the class can only be accessed by the public member function of the same class. The `protected` access specifier is the same as private. The only difference is that it allows its derived classes to access the protected members directly without the member functions. This difference is explained later with suitable examples.

A new class can be defined as per the syntax given below. The derived class is indicated by associating with the base class. A new class also has its own set of member variables and functions. The syntax given below creates the derived class.

```
class name of the derived class: access specifiers - name of the base
class
{
  _______________ _
  _____________      // member variables of new class (derived class)
  _______________ _
}
```

The names of the derived and base classes are separated by a colon (**:**). The access specifiers may be `private` or `public`. The keyword `private` or `public` is specified followed by a colon. Generally, the access specifier is to be specified. In the absence of an access specifier, the default is private. The access specifiers decide whether the characteristics of the base class are derived privately or publicly. The derived class also has its own set of member variables and functions. The following are the possible syntaxes of declaration:

(1)

```
class B: public A
{
   // Members of class B
};
```

In the above syntax, class `A` is a base class, and class `B` is a derived class. Here, the class `B` is derived publicly.

(2)

```
class B: private A // private derivation
{
   // members of class B
};
```

(3)

```
class B: A // by default private derivation
{
   // members of class B
};
```

(4)

```
class B: protected A // same as private
{
   // members of class B
};
```

(5)

```
Struct B: A // public derivation
{
   // members of class B
};
```

In the above syntaxes, the class `B` is derived `privately` from the base class `A`. If no access specifier is given, the default mode is private. The use of `protected` access specifier is the same as `private` access. If `struct` is used instead of `class`, the default derivation is public. It is important to note the following points:

(a) When a public access specifier is used (example 1), the public members of the base class are public members of the derived class. Similarly, the protected members of the base class are protected members of the derived class.

(b) When a private access specifier is used, the public and protected members of the base class are the private members of the derived class.

Public Inheritance

The class can be derived either publicly or privately. No third type exists. When a class is derived publicly, all the public members of the base class can be accessed directly in the derived class. However, in private derivation, an object of the derived class has no permission to directly access even public members of the base class. In such a case, the public members of the base class can be accessed using public member functions of the derived class.

In case the base class has private member variables and a class derived publicly, the derived class can access the member variables of the base class using only member functions of the base class. The public derivation does not allow the derived class to access the private member variable of the class directly as is possible for public member variables. The following example illustrates public inheritance where base class members are declared public and private.

11.1 Write a program to derive a class publicly from base class. Declare the base class with its member under public section.

```
#include<iostream.h>
#include<constream.h>
// PUBLIC DERIVATION //

class A            // BASE CLASS
{
   public:
```

```
   int x;
};
class B: public A   // DERIVED CLASS
{
   public:
   int y;
};
int main()
{
   clrscr();
   B b;          // DECLARATION OF OBJECT
   b.x=20;
   b.y=30;
   cout<<"\n member of A:"<<b.x;
   cout<<"\n Member of B:"<<b.y;
   return 0;
}
```

OUTPUT

```
Member of A : 20
Member of B : 30
```

Explanation: In the above program, two classes are defined, and each contains one public member variable. The `class B` is derived publicly from the class `A`. Consider the following statement:

```
class B : public A
```

The above statement is used to derive the new class. The keyword `public` is used to derive the class publicly. The access specifier is followed by the name of the base class.

In function `main()`, `b` is an object of `class B`. The object b can access the members of class `A` as well as of `class B` through the following statements:

```
b.x=20; // Access to base class members
b.y=30; // Access to derived class members
```

Thus, the derived class holds the members of the base class, and its object has the permission to access the members of the base class.

11.2 Write a program to derive a class publicly from base class. Declare the base class member under private section.

```
#include<iostream.h>
#include<constream.h>
// PUBLIC DERIVATION //

class A          // BASE CLASS
```

```
{
   private:
   int x;
   public:
   A() {x=20;}
   void showx()
{
   cout<<"\n x="<<x;
}
};
class B : public  A   // DERIVED CLASS
{
   public:
   int y;
   B() {y=30;}
void show()
{
   showx();
   cout<<"\n y="<<y;
}
};
int main()
{
   clrscr();
   B b;         // DECLARATION OF OBJECT
   b.show();
   return 0;
}
```

OUTPUT

```
x=20
y=30
```

Explanation : In the above program, the class `A` has one private member, default constructor, and member function `showx()`. The `class B` is derived from the class `A` publicly. The `class B` contains one public member variable `y`, default constructor, and member function `show()`.

In function `main()`, `b` is an object of the derived `class B`. Though the `class B` is derived publicly from the class `A,` the status of the members of the base class remains unchanged. The object `b` can access public members, but cannot access the private members directly. The private members of the base class can be accessed using the public member function of the base class.

The object `b` invokes the member function `show()` of the derived class. The function `show()` invokes the `showx()` function of the base class.

The object b can access the member function defined in both base and derived classes. The following statements are valid:

```
b.showx(); // Invokes member function of base class
b.show(); // Invokes member function of derived class
```

The constructor and member functions of the derived class cannot access the private members. Due to this, separate constructor and member functions are defined in both base and derived classes.

Private Inheritance

The objects of the privately derived class cannot access the public members of the base class directly. Hence, the member functions are used to access the members.

11.3 Write a program to derive a class privately. Declare the member of base class under public section.

```
// PRIVATE INHERITANCE //

#include<iostream.h>
#include<constream.h>

class A                // BASE CLASS
{
   public:
   int x;
};
class B : private  A   // DERIVED CLASS
{
   public:
   int y;
   B()
   {x=20;
   y=40;
   }
void show()
   {cout<<"\n x="<<x;
   cout<<"\n y="<<y;
   }
};
int main()
{
   clrscr();
   B b;            // DECLARATION OF OBJECT
   b.show();
   return 0;
}
```

OUTPUT

```
x=20
y=40
```

Explanation: In the above program, the `class B` is derived privately from the `class A`. The member variable `x` is a public member of the base class. However, the object `b` of the derived class cannot directly access the variable `x`. The following statements are invalid:

```
b.x=30; // cannot access
```

The `class B` is derived privately. Hence, its access is restricted. The member function of the derived class can access the members of the base class. The function `show()` does the same.

11.4 Write a program to derive a class privately.

```
#include<iostream.h>
#include<constream.h>

class A                 // BASE CLASS
{
   int x;
   public:
   A()
   {x=20;}
   void showx()
   {
   cout<<"\n x="<<x;
   }
};
class B : private A    // DERIVED CLASS
{
   public:
   int y;
   B()
   {  y=40;
   }
   void show()
   {
   showx();
   cout<<"\n y="<<y;
   }
};
int main()
{
   clrscr();
   B b;           // DECLARATION OF OBJECT
   b.show();
return 0;
}
```

OUTPUT

```
x=20
y=40
```

Explanation: In the above program, the member variables of the base class and derivation methods are private. The derived class and its objects cannot access the private members of the base class directly. Hence, separate constructor and member functions are defined in both base and derived classes. The object `b` invokes the member function `show()` to display the members. The constructors are used to initialize the member variables.

From the last few programs, it is clear that handling the public data of a class is not a tough task for the programmer. However, while accessing the private data, difficulties are faced by the programmer, because they cannot be accessed without the member functions. Even the derived class has no permission to access the private data directly. Hence, the programmer needs to define separate constructor and member functions to access the private data of that particular base class. Due to this, the length of the program is increased.

11.4 PROTECTED DATA WITH PRIVATE INHERITANCE

The member function of the derived class cannot access the private member variables of the base class. The private members of the base class can be accessed using the public member function of the same class. This approach makes the program lengthy. To overcome the problems associated with private data, the creator of C++ introduced another access specifier `protected`. The protected is similar to private, but it allows the derived class to access the private members directly. Consider the following example:

11.5 Write a program to declare protected data in base class. Access data of base class declared under Protected section using member functions of derived class.

```
// PROTECTED DATA //

#include<iostream.h>
#include<constream.h>

class A           // BASE CLASS
{
   protected:     // protected declaration
   int x;
};
class B : private A // DERIVED CLASS
{
   int y;
   public:
   B()
   {
   x=30;
   y=40;
   }
   void show()
   {
   cout<<"\n x="<<x;
   cout<<"\n y="<<y;
   }
};
int main()
{
   clrscr();
   B b;          // DECLARATION OF OBJECT
   b.show();
   return 0;
}
```

OUTPUT

```
x=30
y=40
```

Explanation: In this program, the data member variable of `class A` is declared under the protected section. The `class A` contains neither default constructor nor any member function. The `class b` is derived from class A. The member function of the derived `class B` can access the protected member of the base class.

Thus, the protected mechanism reduces the program size. The derived class need not depend on the member function of the base class to access the data. The protected data safeguard the data from direct use and allow only the derived classes to access the data.

The protected access specifier should be used when it is known in advance that a particular class can be used as a base class. The data members of the classes that can be used as a base class should be declared as protected. In such classes, the programmer need not define functions. The member functions of the base class are rarely useful.

In the inheritance mechanism, the derived classes have more number of members as compared with the base class. The derived class contains the properties of a base class and a few of its own. The object of the derived class can access the members of both base and derived classes. However, the object of the base can access the members of only the base class and not any of the derived class, as shown in Figure 11.2. Hence, the programmer always uses objects of the derived classes and performs operations. The member function of the base class cannot access the data of the derived class. Obviously, the programmer uses the object of the derived classes and possibly avoids defining the member function in the base class.

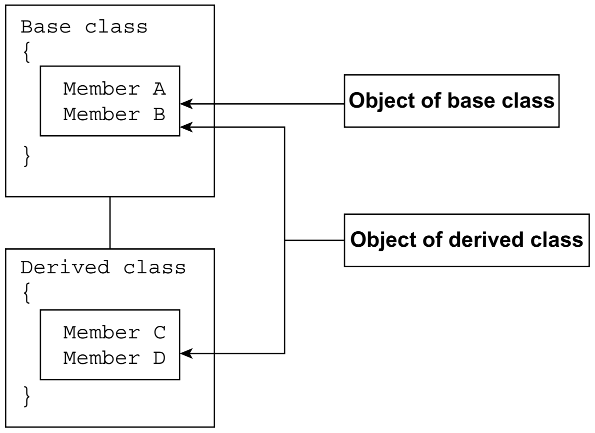


Fig. 11.2 Difference between base and derived class objects

We have learned about the access specifiers in detail. Now before moving on to a new topic, let us revise a few points related to `private`, `public`, and `protected` keywords. Table 11.1 gives a description of access specifiers followed by an explanation. Figure 11.3 gives a pictorial representation of access control.

Table 11.1 Access specifiers with their scope

Sr.No.	Base class access mode	Derived class access mode		
		`Private` derivation	`Public` derivation	`Protected` derivation
A	`public`	`private`	`public`	`protected`
B	`private`	Not inherited	Not inherited	Not inherited
C	`protected`	`private`	`protected`	`protected`

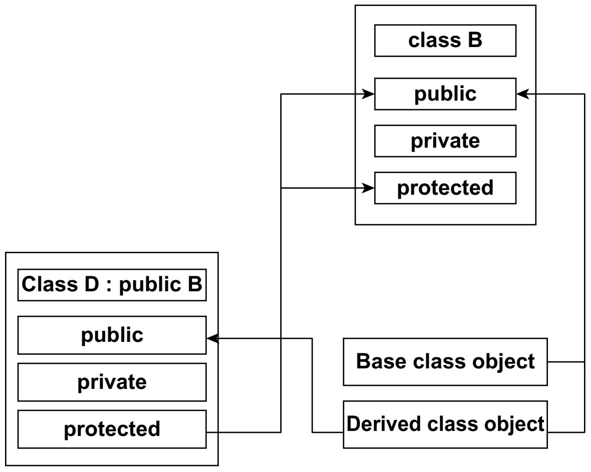


Fig. 11.3 Access scope of class members

(a) Accessible to the member function of the same class, derived class, and using objects. When a class is derived by private derivation, public members of the base class are private in the derived class.
(b) Accessible to the member function inside its own class but not in the derived class. The derived class cannot access the private members of the base class directly. The private members of the base class can be accessed only using the public member function of the same class.
(c) Accessible to the member function of the base and derived classes. When a class is derived privately, the protected members of the base class become private, and when they are derived publicly, the protected members remain protected in the derived class.

The syntax of public, private, and protected access specifiers is as follows:

```
Syntax:
Public: <declarations>
Private: <declarations>
Protected: <declarations>
```

If the member variables of a class are protected, the access specifier should be private. A protected member can be considered a hybrid of public and private members. A protected member is public for its derived class and private for other class members. Member functions and friends can use the member classes derived from the declared class. Only objects of the derived class do this task. It is possible to override the default struct access with private or protected. However, it is not possible to override the default union access. Figure 11.5 shows a pictorial representation of the access control in classes.

(1) All private members of the class are accessible to public members of the same class. They cannot be inherited.
(2) The derived class can access the private members of the base class using the member function of the base class.
(3) All the protected members of the class are available to its derived classes and can be accessed without the use of the member function of the base class. In other words, we can say that all protected members act as public for the derived class.
(4) If any class is prepared for deriving classes, it is advisable to declare all members of the base class as protected, so that derived classes can access the members directly.
(5) All the public members of the class are accessible to its derived class. There is no restriction for accessing elements.
(6) The access specifier required while deriving class is either private or public. If not specified, private is default for classes and public is default for structures.
(7) Constructors and destructors are declared in the public section of the class. If declared in the private section, the object declared will not be initialized and the compiler will flag an error.
(8) The private, public, and protected sections (visibility sections) can be defined several times in any sequence. In Figure 11.4, you can observe that the public section is declared twice, the private section is defined at last, and the protected section is declared in between two public sections. The sections can be declared in any sequence for any number of times.

```
class <class name>
{
    public:                      //  public section
           <declarations>
    protected:                   //  protected section
          <declarations>
    public:                      //  public section
           <declarations>
    private:                     //  private section.
          <declarations>
};
```

Fig. 11.4 Visibility sections

Member functions scope

The following type of functions can have access to the protected and private members of the class. Table 11.2 describes these functions. Figure 11.5 shows access scope.

Table 11.2 Access controls of functions

Sr. No.	Type of Functions	Access modes		
		Private	Protected	Public
A	Class member function	✓	✓	✓
B	Derived class member	×	✓	✓
C	Friend function	✓	✓	✓
D	Friend class member	✓	✓	✓

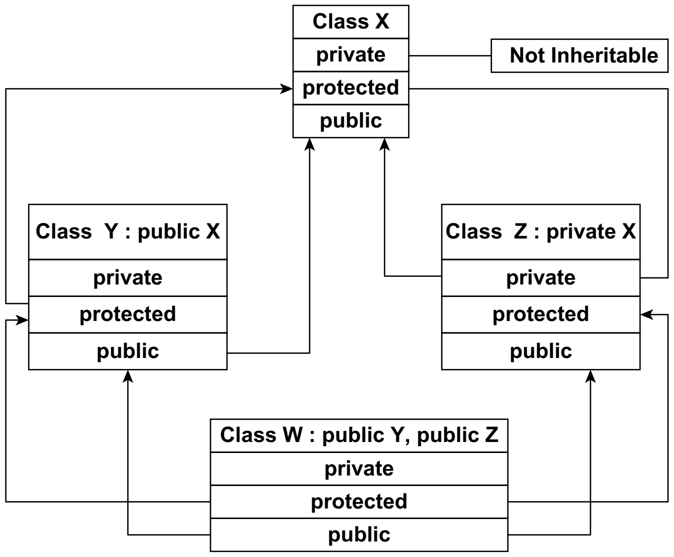


Fig. 11.5 Access scope in classes

(a) The class member function can access the private, protected, and public members.
(b) The derived class member function cannot access the private members of the base class directly. However, the private members can be accessed using the member functions of the base class.
(c) The friend function of a friend class can access the private and protected members.
(d) The member function of a friend class can access the private and protected member of the class.

11.5 TYPES OF INHERITANCE

So far, we have learned examples of simple inheritance that use one base class and one derived class. The process of inheritance can be either simple or complex.

This depends on the following points:

(1) `Number of base classes:` The program can use one or more base classes to derive a single class.
(2) `Nested derivation:` The derived class can be used as a base class, and a new class can be derived from it. This can be possible at any extent.

Based on the above points, inheritance is classified as follows:

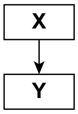
(a) Single Inheritance
(b) Multiple Inheritance
(c) Hierarchical Inheritance
(d) Multilevel Inheritance
(e) Hybrid Inheritance
(f) Multi-path Inheritance

The different types of inheritance are described in Figure 11.6. The base classes are at the top, and the derived classes are at the bottom. The arrow pointing from top to bottom indicates that the properties of the base class are inherited by the derived class and vice versa is not applicable.

(a) Single Inheritance
This occurs when only one base class is used for the derivation of a derived class. Further, derived class is not used as a base class, such a type of inheritance that has one base and derived class is known as `single inheritance`. Figure 11.6 (a) indicates single inheritance.
(b) Multiple Inheritance
When two or more base classes are used for the derivation of a class, it is called `multiple inheritance`. Figure 11.6 (b) indicates multiple inheritance.
(c) Hierarchical Inheritance
When a single base class is used for the derivation of two or more classes, it is known as `hierarchical inheritance`. Figure 11.6 (c) indicates hierarchical inheritance.
(d) Multilevel Inheritance
When a class is derived from another derived class, that is, the derived class acts as a base class, such a type of inheritance is known as `multilevel inheritance`. Figure 11.6 (d) indicates multilevel inheritance.
(e) Hybrid Inheritance
A combination of one or more types of inheritance is known as `hybrid inheritance`. Figure 11.6 (e) indicates hybrid inheritance.

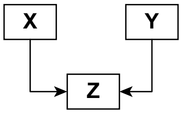
(f) Multipath Inheritance

When a class is derived from two or more classes, those are derived from the same base class. Such a type of inheritance is known as `multipath inheritance`. Figure 11.6 (f) indicates multipath inheritance.



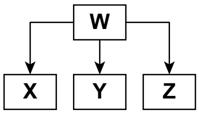
X is a base class. Y is a derived class. This type involves one base and derived class. Further, no class is derived from Y.

(a) Single Inheritance



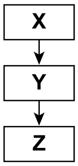
X and Y are base classes. Z is a derived class. Class Z inherits properties of both X and Y. Further, Z is not used as a base class.

(b) Multiple Inheritance



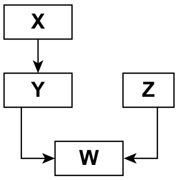
W is only one base class. X, Y and Z are derived classes. Further, X, Y and Z are not used for deriving a class.

(c) Hierarchical Inheritance



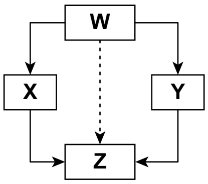
X is a base class. Y is derived from X. Further, Z is derived from Y. Here, Y is not only a derived class but also a base class. Further, Z can be used as a base class.

(d) Multilevel Inheritance



In this type, two types of inheritance is used, i.e. single and multiple inheritance. Class Y is derived from class X. It is single type of inheritance. Further, the derived class Y acts as a base class. The class W is derived from base classes Y and Z. This type of inheritance that uses more than one base class is known as multiple inheritances. Thus, combination of one or more type of inheritance is called as Hybrid inheritance.

(e) Hybrid Inheritance



W is a base class. The classes X and Y are derived from W base class. Both X and Y inherits properties of class W. Further, class Z is derived from X and Y. X and Y have same copies of members inherited from W. Here, ambiguity is generated. Hence, virtual keyword is used to avoid ambiguity. The virtual mechanism is explained later.

(f) Multipath Inheritance

Fig. 11.6 Types of inheritance

11.6 SINGLE INHERITANCE

When only one class is derived from a single base class, such a derivation of a class is known as `single inheritance`; further, the derived class is not used as a base class. This type of inheritance uses one base and one derived class.

The new class is termed `derived class`, and the old class is called `base class`, as shown in Figure 11.7. A derived class inherits data member variables and functions of the base class. However, constructors and destructors of the base class are not inherited in the derived class.

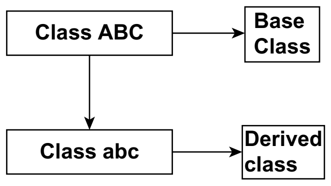


Fig. 11.7 Single inheritance

The newly created class receives entire characteristics from its base class. In single inheritance, there are only one base class and one derived class. The single inheritance is not as complicated as compared with other types of inheritance. Consider the following example:

In the above diagram, `class ABC` is a `base class`, and `class abc` is a `derived class`. The arrow shows that `class abc` is derived from `class ABC`. The program given below illustrates the single inheritance.

11.6 Write a program to show single inheritance between two classes.

```
#include<iostream.h>
#include<constream.h>

class ABC
{
   protected:
   char name[15];
   int age;
};
class abc : public ABC // public derivation
{
   float height;
   float weight;
   public:
   void getdata()
   {
   cout<<"\n Enter Name and Age:";
   cin>>name>>age;
   cout<<"\n Enter Height and Weight:";
   cin>>height >>weight;
   }
   void show()
   {
   cout<<"\n Name:"<<name <<"\n Age:"<<age<<" Years";
   cout<<"\n Height:"<<height <<" Feets"<<"\n Weight:"<<weight
               <<"Kg.";
   }
};
int main()
{
   clrscr();
   abc x;
   x.getdata(); // Reads data through keyboard.
   x.show(); // Displays data on the screen.
   return 0;
}
```

OUTPUT

```
Enter Name and Age : Santosh 24
Enter Height and Weight : 4.5 50
Name : Santosh
Age : 24 Years
Height : 4.5 Feets
Weight : 50 Kg.
```

Explanation: In the above program, the two classes `ABC` and `abc` are declared. The `class ABC` has two protected data members, `name` and `age.` The `class abc` has two `float` data members, `height` and `weight` with a two-member function `getdata()` and `show()`. The class `abc` is derived from class `ABC`. The statement `class abc: public ABC` defines the derived `class abc`. In function `main()`, `x` is an object of the derived `class abc`. The object `x` invokes the member function `getdata()` and show`()`. This function reads and displays data, respectively.

11.7 MULTILEVEL INHERITANCE

The procedure of deriving a class from a derived class is called `multilevel inheritance.`

In the above figure, `class A3` is derived from `class A2`. The `class A2` is derived from `class A1`. The `class A3` is a derived class. The `class A2` is a derived class as well as a base class for `class A3`. The `class A2` is called an `intermediate base class`. The `class A1` is a base class of `classes A2` and `A3`. The series of classes A1, A2, and A3 is called an `inheritance pathway`, as shown in Figure 11.8.

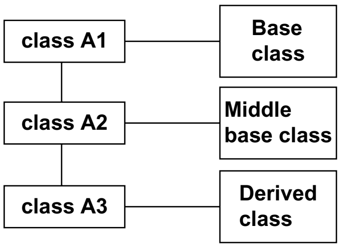


Fig. 11.8 Multilevel inheritance

11.7 Write a program to create multilevel inheritance. Create classes A1, A2, and A3.

```
// Multilevel inheritance //
#include<iostream.h>
#include<constream.h>

class A1 // Base class
{
   protected:
   char name[15];
   int age;
};
class A2 : public A1 // Derivation first level
{
   protected:
   float height;
   float weight;
};
class A3 : public A2 // Derivation second level
{
   protected:
   char sex;
   public:
   void get() // Reads data
   {
   cout<<"Name:"; cin>>name;
   cout<<"Age:"; cin>>age;
   cout<<"Sex:"; cin>>sex;
   cout<<"Height:"; cin>>height;
   cout<<"Weight:"; cin>>weight;
   }
   void show() // Displays data
   {
   cout<<"\nName:" <<name;
   cout<<"\nAge:" <<age <<" Years";
   cout<<"\nSex:" <<sex;
   cout<<"\nHeight:" <<height <<" Feets";
   cout<<"\nWeight:" <<weight <<" Kg.";
   }
}; int main()
{
   clrscr();
   A3 x; // Object Declaration
   x.get(); // Reads data
   x.show(); // Displays data
   return 0;
}
```

OUTPUT

```
Name : Balaji
Age : 26
Sex : M
Height : 4
Weight : 49.5
Name : Balaji
Age : 26 Years
Sex : M
Height : 4 Feets
Weight : 49.5 Kg.
```

Explanation: In the above program, the `classes A1, A2,` and `A3` are declared. The member variables of all these classes are protected. The class `A2` is derived from `class A1`. The `class A3` is derived from `class A2`. Thus, the `class A2` acts as a derived class as well as a base class. The function `get()` reads data through the keyboard, and the function `show()` displays data on the screen. Both the functions are invoked using object `x` of `class A3`.

11.8 MULTIPLE INHERITANCE

Multiple inheritance is a latest addition to the C++ language. When a class is derived from more than one class, this type of inheritance is called `multiple inheritance`. A class can be derived by inheriting the properties of more than one class. Properties of various pre-defined classes are transferred to a single derived class. Figure 11.9 shows multiple inheritance.

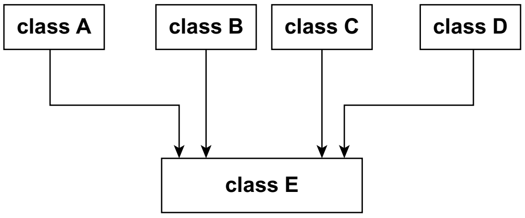


Fig. 11.9 Multiple inheritance

11.8 Write a program to derive a class from multiple base classes.

```
// Multiple Inheritance //

#include<iostream.h>
#include<constream.h>
class A {protected: int a;}; // class A declaration
class B {protected: int b;}; // class B declaration
class C {protected: int c;}; // class C declaration
class D {protected: int d;}; // class D declaration
// class E : public A, public B, public C, public D

class E : public A,B,C,D // Multiple derivation
```

```
{
   int e;
   public:
   void getdata()
   {
   cout<<"\n Enter values of a,b,c & d & e:";
   cin>>a>>b>>c>>d>>e;
   }
   void showdata()
   {
   cout<<"\n a="<<a <<" b="<<b <<" c = "<<c <<" d= "<<d <<" e="<<e;
   }
};
int main()
{
   clrscr();
   E x;
   x.getdata(); // Reads data
   x.showdata(); // Displays data
   return 0;
}
```

OUTPUT

```
Enter values of a,b,c & d & e : 1 2 4 8 16
a=1 b = 2 c = 4 d= 8 z= 16
```

Explanation: In the above program, `classes A, B, C, D,` and E are declared with a one-integer member variable each. The `class E` has a two-member function `getdata()` and `showdata()`**,** respectively. The `getdata()` is used to read integers through the keyboard, and `showdata()` is used to display the contents on the screen. The `class E` is derived from the classes `A, B, C,` and `D`. The `classes A, B, C,` and `D` act as a base class. The derivation is carried out with the statement `class E: public A, B, C, and D`. The class members of A are publicly derived, and the members of other classes are privately derived. To derive all class members publicly, they would be as `class Z: public A, public B, public C,` **and** `public D.` Remember, the meaning of both the statements is not same. Most programmers refer to the second format, as it reduces the code. The output of the program is as given above.

11.9 HIERARCHICAL INHERITANCE

We learned that in inheritance, one class could be inherited from one or more classes. In addition, new members are added to the derived class. The inheritance also supports the hierarchical arrangement of a program. Several programs require the hierarchical arrangement of classes, in which derived classes share the properties of the base class. A hierarchical unit exhibits topdown style through splitting a compound class into several simple subclasses. Figure 1.16 in Chapter 1 is a perfect example of the hierarchy of classes. The program based on it is illustrated as follows:

11.9 Write a program to show hierarchical inheritance.

```
#include<constream.h>
#include<iostream.h>

class red
{
   public:
   red() {cout<<"Red";};
};
class yellow
{
   public:
   yellow() {cout<<"Yellow";}
};
class blue
{
   public:
   blue() {cout<<"Blue";}
};
class orange : public red, public yellow
{
   public:
   orange() {cout<<"=Orange";}
};
class green : public blue, public yellow
{
   public:
   green() {cout<<"=Green";}
};
class violet : public red, public blue
{
   public:
   violet() {cout<<"=Violet";}
};
class reddishbrown : public orange, public violet
{
   public:
   reddishbrown() {cout<<"=Reddishbrown";}
};
class yellowishbrown : public green, public orange
```

```
{
   public:
   yellowishbrown() {cout<<"=Yellowishbrown";}
};
class bluishbrown : public violet, public green
{
   public:
   bluishbrown() {cout<<"=Bluishbrown";}
};
int main()
{
   clrscr();
   reddishbrown r;
   endl(cout);
   bluishbrown b;
   endl(cout);
   yellowishbrown y;
   endl(cout);
   return 0;
}
```

OUTPUT

```
Red Yellow = Orange Red Blue = Violet = Reddishbrown
Red Blue = Violet Blue Yellow = Green = Bluishbrown
Blue Yellow = Green Red Yellow = Orange = Yellowishbrown
```

Explanation: For each color, a separate class with constructors is declared. The classes red, blue, and yellow are base classes. The class orange is derived from red and yellow. The class green is derived from blue and yellow. The class violet is derived from red and blue.

The class reddishbrown is derived from orange and violet; the class yellowishbrown is derived from green and orange; and, lastly, the class bluishbrown is derived from violet and green.

In function `main()`, objects of the classes reddishbrown, yellowishbrown, and bluishbrown are declared. When the objects are declared, constructors are executed from the base to derived classes. When executed, the constructor displays the class name (color name). The output shows the names of different colors and resulting colors after their combination.

11.10 HYBRID INHERITANCE

A combination of one or more types of inheritance is known as `hybrid inheritance`. Sometimes, it is essential to derive a class using more types of inheritance. Figure 11.10 shows hybrid inheritance. In the below given diagram, the `class game` is derived from two base classes, that is, `location` and `physique`. The `class physique` is also derived from the `class player`.

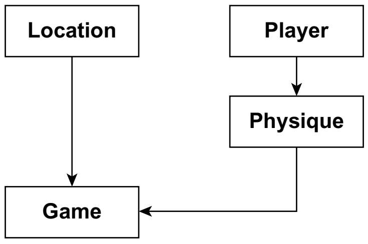


Fig. 11.10 Hybrid inheritance

11.10 Write a program to create a derived class from multiple base classes.

```
// Hybrid Inheritance //
#include<iostream.h>
#include<constream.h>

class PLAYER
{
   protected:
   char name[15];
   char gender;
   int age;
};
class PHYSIQUE : public PLAYER
{
   protected:
   float height;
   float weight;
};
class LOCATION
{
   protected:
   char city[10];
   char pin[7];
};
class GAME : public PHYSIQUE, LOCATION
{
   protected:
   char game[15];
   public:
   void getdata()
   {
   cout<<"Enter Following Information\n";
   cout<<"Name:"; cin>>name;
   cout<<"Gender:"; cin>>gender;
   cout<<"Age:"; cin>>age;
   cout<<"Height:"; cin>>height;
   cout<<"Weight:"; cin>>weight;
   cout<<"City:"; cin>>city;
   cout<<"Pincode:"; cin>>pin;
   cout<<"Game:"; cin>>game;
   }
   void show()
   {
   cout<<"\n Entered Information";
   cout<<"\nName:"; cout<<name;
   cout<<"\nGender:"; cout<<gender;
   cout<<"\nAge:"; cout<<age;
   cout<<"\nHeight:"; cout<<height;
   cout<<"\nWeight:"; cout<<weight;
   cout<<"\nCity:"; cout<<city;
   cout<<"\nPincode:"; cout<<pin;
   cout<<"\nGame:"; cout<<game;
   }
};
int main()
{
   clrscr();
   GAME G;
   G.getdata();
   G.show();
   return 0;
}
```

OUTPUT

```
Enter Following Information
Name : Mahesh
Gender : M
Age : 25
Height : 4.9
Weight : 55
City : Nanded
Pincode : 431603
Game : Cricket
Entered Information
Name : Mahesh
Gender : M
Age : 25
Height : 4.9
Weight : 55
City : Nanded
Pincode : 431603
Game : Cricket
```

Explanation: In the above program, `player`, `physique`, `location,` and `game` classes are defined. All the data members of these four classes are protected. The `class physique` is derived from the `base class player`. The `class game` is derived from `physique` and `location`; that is, the derived `class game` has two base classes `physique` and `location.` The `location` is a separate class and is not derived from any other class. The `getdata()` and show() are functions of class game and reads and display data, respectively. In main(), the object G of class game calls these functions one by one to read and write the data. The output of the program is shown at the end of the above program.

11.11 MULTIPATH INHERITANCE

When a class is derived from two or more classes, those are derived from the same base class. Such a type of inheritance is known as multipath inheritance. The multipath inheritance also consists of many types of inheritance, such as multiple, multilevel, and hierarchical, as shown in Figure 11.11.

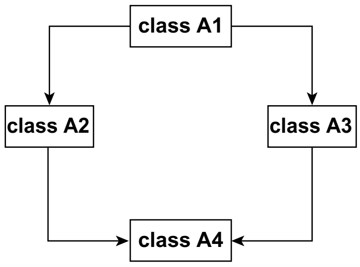


Fig. 11.11 Ambiguity in classes

```
Consider the following example:
class A1
{
   protected:
   int a1;
};
class A2 : public A1
{
   protected:
   int a2;
};
class A3: public A1
{
   protected: int a3;
};
class A4: public A2,A3
{
int a4;
};
```

In the above example, classes A2 and A3 are derived from class A1; that is, their base class is similar to class A1 (hierarchical inheritance). Both classes A2 and A3 can access the variable a1 of class A1. The class A4 is derived from classes A2 and A3 by multiple inheritance. If we try to access the variable a1 of class A1, the compiler shows error messages as given below.

```
(a) Error PRG58.CPP 30: Member is ambiguous: 'A1::a1' and 'A1::a1'
(b) Error PRG58.CPP 37: Member is ambiguous: 'A1::a1' and 'A1::a1'
```

In the above example, we can observe all types of inheritance, that is, multiple, multilevel, and hierarchical. The derived class A4 has two sets of data members of class A1 through the middle base classes A2 and A3. The class A1 is inherited twice.

11.12 VIRTUAL BASE CLASSES

To overcome the ambiguity occurring due to multipath inheritance, the C++ provides the keyword `virtual`. The keyword `virtual` declares the specified classes virtual. The example given below illustrates the virtual classes:

```
class A1
{
protected:
int a1;
};
class A2 : virtual public A1 // virtual class declaration
{
protected:
int a2;
};
class A3: virtual public A1 // virtual class declaration
{
protected:
int a3;
};
class A4: public A2,A3
{
int a4;
};
```

When classes are declared as virtual, the compiler takes essential caution to avoid the duplication of member variables. Thus, we make a class virtual if it is a base class that has been used by more than one derived class as their base class.

Consider another example that will give you a clear idea of virtual classes as well as the necessary care to be taken while declaring virtual classes.

(a) A base class cannot be specified more than once in a derived class.

```
class A
{------
------};
class A1: A, A {...}; // Illegal Declaration
```

Here, class A is specified twice. Hence, it is an illegal declaration.

(b) A base class can be indirectly passed to the derived class more than once.

```
class L: public A {...}
class K: public A {...}
class J: public L, public K {...} // Legal declaration
```

In the above case (b) , each object of class J will contain two sub-objects of class A through classes L and K.

(c) The case (b) causes problems. To avoid duplication, we can add the keyword virtual to a base class specifier.

```
For example,
class L: virtual public A {...}
class K: virtual public A {...}
class J: public K, public J {...}
```

The keyword virtual always appears before the class name. A is now a virtual base class, and class J has only one sub-object of class A.

11.11 Write a program to declare virtual base classes. Derive a class using two virtual classes.

```
// VIRTUAL BASE CLASSES //

#include<iostream.h>
#include<conio.h>

class A1
{
   protected:
   int a1;
};
class A2 : public virtual A1 // virtual declaration
{
   protected:
   int a2;
};
class A3 : public virtual A1 // virtual declaration
{
   protected:
   int a3;
};
class A4 : public A2,A3 // virtual declaration
{
   int a4;
   public:
   void get()
   {
   cout<<"Enter values for a1, a2,a3 and a4:";
   cin>>a1>>a2>>a3>>a4;
   }
   void put()
```

```
    {
    cout<<"a1="<<a1 <<"a2="<<a2 <<"a3="<<a3 <<"a4="<<a4;
    }
};
int main()
{
    clrscr();
    A4 a;
    a.get(); // Reads data
    a.put(); // Displays data
    return 0;
}
```

OUTPUT

```
Enter values for a1, a2,a3 and a4 : 5 8 7 3
a1= 5 a2 = 8 a3 = 7 a4 = 3
```

Explanation: In the above program, the `classes A1, A2, A3, and A4` are declared, and each contains one `protected member variable`. The class `A4` has two member functions `get()` and `put()`. The `get()` function reads integers through the keyboard. The `put()` function displays the contents of the member variables on the screen. The `classes A2` and `A3` are derived from `class A1`. While deriving these two classes, the `class A1` is declared as virtual as per the following statements:

```
class A2: public virtual A1 // derivation of class A2
class A3: public virtual A1 // derivation of class A3
class A4: public A2,A3 // derivation of class A4
```

The `class A4` is derived from two `classes A2 and A3` as per the statement `(c)`. In function `main()`, the object `a` of `class A4` invokes the member function `get()` and `put()`. The output of the program is as shown above.

11.13 CONSTRUCTORS, DESTRUCTORS, AND INHERITANCE

The constructors are used to initialize member variables of the object, and the destructor is used to destroy the object. The compiler automatically invokes constructors and destructors. The derived class does not require a constructor, if the base class contains a zero-argument constructor. In case the base class has a parameterized constructor, then it is essential for the derived class to have a constructor. The derived class constructor passes arguments to the base class constructor. In inheritance, normally derived classes are used to declare objects. Hence, it is necessary to define constructors in the derived class. When an object of a derived class is declared, the constructors of the base and derived classes are executed.

11.13.1 Constructors and destructors in base and derived classes

In inheritance, destructors are executed in reverse order of constructor execution. The destructors are executed when an object goes out of scope. To know the execution of constructors and destructors, let us study the following program:

11.12 Write a program to show sequence of execution of constructor and destructor in multiple inheritance.

```
#include<iostream.h>
#include<constream.h>

class A // base class
{
   public:
   A()
   {
   cout<<"\n Zero-argument constructor of base class A";}
   ~A()
   {
   cout<<"\n Destructor of the class A";}
   };
class B // base class
{
   public:
   B()
   {
   cout<<"\n Zero-argument constructor of base class B";}
   ~B() {cout<<"\n Destructor of the class B" ;}
   };
class C : public A, public B // Derivation of class
{
   public:
   C()
   {
   cout<<"\n Zero-argument constructor of derived class C";
   }
   ~C() {cout<<"\n Destructor of the class C";}
};
int main()
{
   clrscr();
   C objc; // Object declaration
   return 0;
}
```

OUTPUT

```
Zero-argument constructor of base class A
Zero-argument constructor of base class B
Zero-argument constructor of derived class C
Destructor of the class C
Destructor of the class B
Destructor of the class A
```

Explanation: In the above program, `classes A` and `b` are two classes. The `class C` is derived from `classes A` and `B`. The constructors of the base classes are executed first followed by the derived class. The destructor of the derived class is executed first followed by the base class.

11.13 Write a program to use constructor and destructor in all classes (base and derived class). Read and display the data.

{coderipe}

```
#include<iostream.h>
#include<conio.h>

class A1
{
   protected:
   char name[15];
   int age;
   A1()
   {
   cout<<"Name:"; cin>>name;
   cout<<"Age:"; cin>>age;
   }
   ~A1()
   {
   cout<<"\nName:" <<name;
   cout<<"\nAge:" <<age;
   }
};
class A2 : public A1
{
   protected:
   float height;
   float weight;
   A2()
   {
   cout<<"Height:"; cin>>height;
   cout<<"Weight:"; cin>>weight;
   }
```

```
    ~ A2()
    {
    cout<<"\nHeight:"<<height;
    cout<<"\nWeight:"<<weight;
    }
};
class A3 : public A2
{
    char sex;
    public:
    A3()
    {
    cout<<"Sex:"; cin>>sex;
    }
    ~ A3()
    {
    cout<<"\nSex:" <<sex;
    }
};
int main()
{
    clrscr();
    A3 x;
    return 0;
}
```

OUTPUT

```
Name : Ajay
Age : 20
Height : 4.5
Weight : 40
Sex : M
Sex : M
Height : 4.5
Weight : 40
Name : Ajay
Age : 20
```

Explanation: In the above program, classes `A1`, `A2`, and `A3` have their own constructors and destructors. Here, the `class A1` is the base class of `A2`, and `A2` is the base class of `A3` or, in other words, `A3` is a derived class of `A2` and `A2` is a derived class of `A1`.

`Execution of constructors:` The execution of constructors takes place from the base class to the derived class. Thus, the sequence of the execution of constructors is `A1()`, `A2()`, and `A3()`.

`Execution of destructors:` The execution of destructors is in opposite order as compared with constructors, that is, from the derived class to the base class. Thus, the sequence of the execution of destructors is `A3()`, `A2()`, and `A1()`. We learnt in Chapter 9 that the object created last destroyed first. The same rule is applicable in the execution of constructors and destructors in inheritance.

11.14 Write a program to create derived class from base classes. Use constructor and destructors.

```
#include<iostream.h>
#include<conio.h>

class in_t
{
   protected:
   int i;
   public:
   in_t()
   {
   cout<<"\n Constructor in_t()";
   i=1;
   }
   ~in_t()
   {
   cout<<"\n Destructor in_t()";
   }
};
class floa_t
{
   protected:
   float f;
   public:
   floa_t()
   {
   cout<<"\n Constructor floa_t()";
   f=1.5;
   }
   ~floa_t()
   {
   cout<<"\n Destructor floa_t()";
   }
};
class cha_r : public in_t, floa_t
{
   protected:
   char c;
   public:
   void show()
   {
   cout<<"\n\ti="<<i;
   cout<<"\n\tf="<<f;
   cout<<"\n\tc="<<c;
   }
   cha_r()
   {
   cout<<"\n Constructor cha_r()";
   c='A';
   }
   ~cha_r()
   {
   cout<<"\n Destructor cha_r()";
   }
};
main()
{
   clrscr();
   cha_r a;
   a.show();
   return 0;
}
```

OUTPUT

```
Constructor in_t()
Constructor floa_t()
Constrcutor cha_r()
i= 1
f= 1.5
c= A
Destructor cha_r()
Destructor floa_t()
Destructor in_t()
```

Explanation: In the above program, three `classes` `in_t`, `floa_t`, and `cha_r` are defined, and each contains one protected variable of type `int`, `float`, and `char`, respectively. Each class contains a constructor and a destructor. In function `main()`, `a` is an object of class derived `class chr_r`. The member function `show()` is invoked by the object `a` to display the contents of member variables of all the classes. The constructors are executed from the base to the derived class, and destructors are loaded from the derived to the base class. The following program explains the execution of constructors and destructors in classes in greater detail:

11.13.2 Base and derived classes without constructors

In the absence of constructors in base and derived classes, the implicit constructors and destructors are executed when objects are declared. Consider the following program:

11.15 Write a program to derive a class. The base and derive classes should be without constructors.

```
#include<iostream.h>
#include<conio.h>

class I
{
   public:
   int x;
};
class II : public I
{
   int y;
   public:
   void set (int j, int k)
   {
   x=j;
   y=k;
   }
   void show()
   {
   cout<<"x="<<x <<"y="<<y;
   }
};
int main()
{
   clrscr();
   II i;
   i.set(4,5);
   i.show();
   return 0;
}
```

OUTPUT

```
X = 4 y = 5
```

Explanation: In the above program, `class I` is a base class, and `class II` is derived from `I`. Both classes are without constructors. The `class II` has used member functions `set()` and `show()` to assign and display data.

11.13.3 Base class with constructors and derived class without constructors

11.16 Write a program to declare a base class with constructor and a derived class without constructor.

```
#include<iostream.h>
#include<constream.h>

class I
{
   public:
   int x;
   I()
   {
   cout<<"In base class constructor";
   x=10;
   }
};
class II : public I
{
   int y;
   public:
};
int main()
{
   clrscr();
   II i;
   return 0;
}
```

OUTPUT

```
In base class constructor
```

Explanation: The base `class I` has a constructor, and the derived `class II` does not have a constructor. The `i` is an object of the derived `class II`. When `i` is declared, the constructor of the base class is executed. The derived class is without a constructor; hence, only the base class constructor is executed.

11.13.4 Base class without constructors and derived class with constructors

11.17 Write a program to declare base class without constructor and derived class without constructor.

```
#include<iostream.h>
#include<constream.h>
```

```
class I
{
};
class II : public I
{
   public:
   II()
   {
   cout<<"\nIn derived class constructor\n";
   }
};
int main()
{
   clrscr();
   II i; // Object declaration
   return 0;

}
```

OUTPUT

In derived class constructor

Explanation: In the above program the base `class I` has no constructor. The derived `class II` has a constructor. The constructor is executed when the object of `class II` is declared.

11.13.5 Base and derived classes with constructors

11.18 Write a program to declare both base and derived classes with constructor.

```
#include<iostream.h>
#include<conio.h>

class I
{
   public:
   I()
   {
   cout<<"\n In base class constructor";}
   };
   class II : public I
   {
   public:
   II() {cout<<"\n In derived class constructor\n";}
};
```

```
int main()
{
   clrscr();
   II i;
   return 0;
}
```

OUTPUT

```
In base class constructor
In derived class constructor
```

Explanation: In the above program, both the base and derived classes contain constructors. When the object of the derived class type is declared and constructors of both the classes are executed, the constructor of the base class is executed first followed by the constructor of the derived class.

11.13.6 Base class with various constructors and derived class with one constructor

11.19 Write a program to declare multiple constructors in base class and single constructor in derived class.

```
#include<iostream.h>
#include<conio.h>

class I
{
   public:
   int x;
   I() {cout<<"\nZero argument base class constructor";}
   I ( int k)
   {
   cout<<"\nOne argument base class constructor";
   }
};
class II : public I
{
   int y;
   public:
   II (int j)
   {
   cout<<"\nOne argument derived class constructor";
   y=j;
   }
};
```

```
int main()
{
   clrscr();
   II i(2);
   return 0;
}
```

OUTPUT

Zero argument base class constructor
One argument derived class constructor

Explanation: In the above program, the `class I` has a zero- and a one-argument constructor. The `class II` has only a one-argument constructor. The object `i` is declared with an integer. The zero-argument constructor of the base class and the one-argument constructor of the derived class are executed.

11.13.7 Base and derived classes without default constructors

11.20 Write a program to declare base and derived class without default constructor.

```
#include<iostream.h>
#include<conio.h>

class I
{
   public:
   int x;
   I ( int k)
   {
   x=k;
   cout<<"\nOne argument base class constructor";}
};
class II : public I
{
   int y;
   public:
   II (int j):I(j)
   {
   cout<<"\nOne argument derived class constructor";
   y=j;
   }
};
int main()
{
   clrscr();
   II i(2);
   return 0;
}
```

OUTPUT

One argument base class constructor
One argument derived class constructor

Explanation: In the above program, no default constructor is declared. In the `class II` one argument, a constructor is declared, and the base class constructor is explicitly invoked. When object `i` is declared, a one-argument constructor of both the classes is executed. In the absence of an explicit call of the base class constructor, the compiler will display the error message "`Cannot find default constructor to initialize base class 'I.'`"

11.13.8 Constructors and multiple inheritance

11.21 Write a program to derive a class using multiple base classes. Observe the execution of constructor when the object of derived class is declared.

```
#include<iostream.h>
#include<conio.h>
class I
{
   public:
   I()
{
   cout<<"\n Zero argument constructor of base class I";
}
};
class II
{
   public:
   II() {cout<<"\n Zero argument constructor of base class II";}
};
class III : public II,I
{
   public:
   III()
{
   cout<<"\n Zero argument constructor of base class III";}
};
int main()
{
   clrscr();
   III i;
   return 0;
}
```

OUTPUT

```
Zero argument constructor of base class II
Zero argument constructor of base class I
Zero argument constructor of base class III
```

Explanation: The `classes I` and `II` are the base classes of the derived `class III`. In function `main()`, `i` is an object of the derived `class III`. The execution of constructors depends on the sequence given while deriving a class as per the following statement:

```
   class III : public II,I
```

Here, the `class II` is the first base class, and `class I` is the second base class; the execution sequence of the constructors will be as shown in the output.

11.13.9 Constructors in multiple inheritance with explicit calls

11.22 Write a program to derive a class using multiple base classes. Invoke the constructors of base classes explicitly.

```
#include<iostream.h>
#include<conio.h>

class I
{
   public:
   I() {cout<<"\n Zero argument constructor of base class I";}
};
class II
{
   public:
   II() {cout<<"\n Zero argument constructor of base class II";}
};
class III : public II,I
{
   public:
   III():II(), I()
   {
   cout<<"\n Zero argument constructor of base class III";
   }
};
```

```
int main()
{
   clrscr();
   III i;
   return 0;
}
```

OUTPUT

```
Zero argument constructor of base class II
Zero argument constructor of base class I
Zero argument constructor of base class III
```

Explanation: In this program in the `class III`, explicitly the constructor of both the base `classes II` and `I` are invoked. The execution sequence can be observed in the output as shown above. The execution sequence of the constructors depends on the sequence of the base class and not on the explicit calls.

11.13.10 Multiple inheritance and virtual class

11.23 Write a program to derive a class using multiple base classes. Invoke the constructors of base classes explicitly. Declare any one base class as virtual.

```
#include<iostream.h>
#include<conio.h>

class I
{
   public:
   I()
{
   cout<<"\n Zero argument constructor of base class I";
}
};
class II
{
   public:
   II()
{  cout<<"\n Zero argument constructor of base class II";}
};
class III : public II, virtual I
{
   public:
   III() : II(), I()
   {
```

```
   cout<<"\n Zero argument constructor of base class III";
   }
};
int main()
{
   clrscr();
   III i;
   return 0;
}
```

OUTPUT

```
Zero argument constructor of base class I
Zero argument constructor of base class II
Zero argument constructor of base class III
```

Explanation: In this program, the base `class I` is declared a virtual class while deriving the class. The constructor of the virtual class is executed first. The execution of the constructor here is not according to the sequence of the base class.

11.13.11 Execution of constructors in multilevel inheritance

11.24 Write a program to derive a classes using multilevel inheritance and observe the execution sequence of constructors.

```
#include<iostream.h>
#include<conio.h>

class I
{
   public:
   I()
   {
   cout<<"\n Zero argument constructor of base class I";
   }
};
class II : public I
{
   public:
   II()
   {
   cout<<"\n Zero argument constructor of base class II";
   }
};
class III : public II
{
   public:
   III() {cout<<"\n Zero argument constructor of base class III";}
};
int main()
{
   clrscr();
   III ii;
   return 0;
}
```

OUTPUT

```
Zero argument constructor of base class I
Zero argument constructor of base class II
Zero argument constructor of base class III
```

Explanation: In this program, the `class II` is derived from the `class I`. The `class III` is derived from the `class II`. The `class II` is a base as well as a derived class. In function `main()`, `ii` is an object of the `class III`. The constructors are executed from base to derived classes as shown in the output. Table 11.3 shows the order of execution of the constructors.

Table 11.3 Execution sequences of constructors

Statements	Sequence of execution	Remarks
Class II : public I	I() – Base class constructor II() – Derived class constructor	Single inheritance
Class III: public I, II	I() – Base class constructor II() – Base class constructor III() – Derived class constructor	Multiple inheritance
Class III: public I, virtual II	II() – Virtual class constructor I() – Base class constructor III() – Derived class constructor	Multiple inheritance

class II : public I {};
class III: public II {};
	I() – First base class constructor
	II() – Second base class constructor
	III() – Derived class constructor
		Multilevel inheritance

11.14 OBJECT AS A CLASS MEMBER

Properties of one class can be used in another class using inheritance or using the object of a class as a member in another class. Declaring the object as a class data member in another class is also known as `delegation.` When a class has an object of another class as its member, such a class is known as a `container class`.

In inheritance, the derived class can use members of the base class. Here, the derived class is a kind of base class. The programmer can also add new members to the derived class.

In delegation, the class consists of objects from other classes. The composed class uses the properties of other classes through their objects. This kind of a relationship is known as `has-a-relationship or containership.`

```
class I
{
*****  // members
*****
}
class II
{
I j;  // object of class I as member
******
}
```

11.25 Write a program use object of one class in another class as a member.

```
#include<iostream.h>
#include<conio.h>

class I
{
   public:
   int x;
   I()
   {
   cout<<"\n Constructor of class I";
   x=20;
   }
};
class II
{
   public:
   int k;
   I y;
   II()
   {
   k=30;
   cout<<"\n Constructor of class II";
   }
   void show()
   {
   cout<<"\n x="<<y.x <<" k="<<k;
   }
};
int main()
{
   clrscr();
   ii.show();
   return 0;
}
```

OUTPUT

```
Constructor of class I
Constructor of class II
x= 20 k = 30
```

Explanation: In the above program, the `class II` contains integer `k` and object `y` of class `I`. In function `main()`, object `ii` is an object of `class II`. The constructor of `class I` is executed first, because when the compiler reaches the `class II`, it finds an object of `class I`. We know that the object declaration always executes the constructor of that class. Thus, the object of `class I` is declared, and the constructor of `class I` is executed. The constructor of `class II` is executed, and the variable of `class II` is initialized. The member function `show()` displays the contents of `x` and `k`. The contents of `class I` member is obtained from object `y`. The dot (**.**) operator is used to access the elements of `class I`.

11.26 Write a program to access member variable of base class using object, scope access operator and direct.

```
#include<iostream.h>
#include<conio.h>

class A1
{
   public:
   char name[15];
   int age;
};
class A2: public A1
{
   private:
   A1 a;
   float height;
   float weight;
   public:
   A2()
   {
   clrscr();
   cout<<"Access Using Scope Access operator\n";
   cout<<"Name:"; cin>>A1::name;
   cout<<"Age:"; cin>>A1::age;
   cout<<"Access Using object of the class\n";
   cout<<"Name:"; cin>>a.name;
   cout<<"Age:"; cin>>a.age;
   cout<<"Access Using direct member variables\n";
   cout<<"Name:"; cin>>name;
   cout<<"Age:"; cin>>age;
   cout<<"Height:"; cin>>height;
   cout<<"Weight:"; cin>>weight;
   }
   ~A2()
   {
   cout<<"\nDisplay using Scope Access operator\n";
   cout<<"\nName:" <<A1::name;
   cout<<"\nAge:" <<A1::age;
   cout<<"\nDisplay Using object of the class\n";
   cout<<"\nName:" <<a.name;
   cout<<"\nAge:" <<a.age;
   cout<<"\nAccess Using direct member variables\n";
   cout<<"\nName:" <<name;
   cout<<"\nAge:" <<age;
   cout<<"\nHeight:" <<height;
   cout<<"\nWeight:" <<weight;
   }
};
int main()
{
   A2 x;
   return 0;
}
```

OUTPUT

```
Name : Ajay
Age : 21
Access Using object of the class
Name : Amit
Age : 20
Access Using direct member variables
Name : Arun
Age : 19
Height : 5.5
Weight: 31
Display using Scope Access operator
Name : Arun
Age : 19
Display Using object of the class
```

```
Name : Amit
Age : 20
Access Using direct member variables
Name : Arun
Age : 19
Height: 5.5
Weight: 31
```

Explanation: In the above program, `A1` and `A2` are two classes. The `class A2` is derived from the `class A1`. The `class A1` has two public member variables. The `class A2` has three private members; one of the members is an object of `class A1`, that is, `object a`. The `object a` holds its separate set of member variables of `class A1.` There are three ways of accessing the member variables of base `class A1`. Table 11.4 describes the status of access of the member variables of `class A1` and the derived `class A2` when we use access specifiers such as `private`, `public`, and `protected`.

Table 11.4 Access Specifiers

Access (using)	`Public`	`Private`	`Protected`
Scope Access :: operator	Possible	Not Possible	Possible
Object	Possible	Not Possible	Not Possible
Direct	Possible	Not Possible	Possible

(1) The first way is to use scope access operators as per the following statements:

```
cin>>A1::name;
cin>>A1::age;
```

In the above statements, `A1` is a class name, and `name` and `age` are member variables of `class A1`. When an access specifier is public or protected, the above statements are valid.

(2) The second method is to use an object of the base class, as per the following statements:

```
cin>>a.name;
cin>>a.age;
```

In the above statement, the member variables of `class A1` are accessed using an object of the same class, who is the member of the derived class. This is possible only when the member variables of `class A1` are public and is not possible if the member variables of `class A1` are protected or private.

(3) The third method directly uses the member variables. Direct access is possible when the access specifier is public or protected, as per the following statements:

cin>>name;
cin>>age;

11.27 Write a program to derive a class from two base classes. Use objects of both the classes as member variables for derived class. Initialize and display the contents of classes using constructor and destructor.

```
#include<iostream.h>
#include<conio.h>

class A
{
   public :
   int a1;
};
class B
{
   public:
   int b1;
};
class AB
{
   public:
   A a;
   B b;
   public :
   AB()
   {
   a.a1=65;
   b.b1=66;
   cout<<"a1="<<a.a1 <<"b1="<<b.b1;
   }
   ~AB(){};
};
int main()
{
   clrscr();
   AB ab;
   return 0;
}
```

OUTPUT

```
a1=65 b1=66
```

Explanation: In the above program, `classes A` and `B` are declared with one integer each. The `class AB` is declared, which contains objects of `classes A` and `B` as member variables. The constructor of `class AB` initializes the member variables of `classes A` and **B.** The constructor also displays contents on the screen. Finally, the destructor destroys the object.

11.15 ABSTRACT CLASSES

When a class is not used for creating objects, it is called an `abstract class`. The abstract class can act as a base class only. It is a layout abstraction in a program, and it allows a base on which several levels of inheritance can be created. The base classes act as a foundation of the hierarchical class. An abstract class is developed only to act as a base class and to inherit, and no objects of this class are declared. An abstract class gives a skeleton or a structure; using this, other classes are shaped. The abstract class is central and generally present at the starting of the hierarchy. The `hierarchy of classes` means a chain or group of classes that are involved with one another. In the last program, `class A` is an abstract class, because no instance (object) of class A is declared.

11.16 QUALIFIER CLASSES AND INHERITANCE

The following program explains the behavior of qualifier classes and the classes declared within them with inheritance.

11.28 Write a program to create derived class from the qualifier class.

```
#include<iostream.h>
#include<conio.h>

class A
{
   public:
   int x;
   A() {}
   class B
   {
   public:
   int y;
   B() {}
   };
}; class C : public A,A::B
{
   public:
   int z;
   void show()
   {
   cout<<endl<<"x="<<x <<"y="<<y<<"z="<<z;
   }
   C (int j,int k, int l)
   {
   x=j;
   y=k;
   z=l;
   }
};
```

```
int main()
{
   clrscr();
   C c(4,7,1);
   c.show();
   return 0;
}
```

OUTPUT

```
x = 4 y =7 z = 1
```

Explanation: In the above program, the `class B` is defined inside the `class A`. The `class A` is a qualifier class of `class B`. The `class C` is inherited from the `classes A` and `B`. In the statement `class C: public A, A::B`, `class C` is inherited from `A` and `B`. To access the `class B`, it is preceded by the qualifier `class A` and the scope access operator. If we mention only `class A`, the `class B` will not be considered for inheritance. Similarly, if we mention only `class B`, the qualifier `class A` will not be considered for inheritance.

11.17 CONSTRUCTORS IN DERIVED CLASS

When a class is declared, a constructor is also declared inside the class in order to initialize data members. It is not possible to use a single constructor for more classes. Every class has its own constructor and destructor with a similar name as the class. When a class is derived from another class, it is possible to define a constructor in the derived class, and the data members of both base and derive classes can be initialized. It is not essential to declare a constructor in a base class. Thus, the constructor of the derived class works for its base class; such constructors are called `constructors` in the derived class or `common constructors`.

11.29 Write a program to initialize member variables of both base and derived class using a constructor of derived class.

```
#include<iostream.h>
#include<conio.h>

class A
{
   protected:
   int x;
   int y;
};
class B : private A
{
   public:
   int z;
   B() {x=1,y=2,z=3;
   cout<<"x="<<x <<"y="<<y <<"z="<<z;}
};
int main()
{
   clrscr();
   B b;
   return 0;
}
```

OUTPUT

```
x= 1 y =2 z=3
```

Explanation: In the above program, the `classes A B` are declared. The `class B` is derived from the `class A`. The constructor of the `class B` initializes member variables of both classes. Hence, it acts as a common constructor of both base and derived classes.

11.18 POINTERS AND INHERITANCE

The private and public member variables of a class are stored in successive memory locations. A pointer to a public member variable gives us access to private member variables.

The same is true for the derived class. The member variables of the base and derived classes are also stored in successive memory locations. The following program explains the mechanics of accessing private data members of the base class using the address of public member variables of the derived class using pointers. Here, no member functions are used.

11.30 Write a program to access private member variables of base class using pointers

```
#include<iostream.h>
#include<conio.h>

class A
{
   private:
   int x;
   int y;
   public:
   A() {
   x=1;
   y=2;
}
};
class B : private A
{
   public:
```

```
   int z;
   B() {z=3;}
};
int main()
{
   clrscr();
   B b; // object declaration
   int *p; // pointer declaration
   p=&b.z; // address of piblic member variabe is stored in
            pointer
   cout<<endl<<"Address of z:"<<(unsigned)p <<""<<"Value of
             z:"<<*p;
   p--; // points to previous location
   cout<<endl<<"Address of y:"<<(unsigned)p <<""<<"Value of
             y:"<<*p;
   p--;
   cout<<endl<<"Address of x:"<<(unsigned)p <<""<<"Value of
             x:"<<*p;
   return 0;
}
```

OUTPUT

```
Address of z : 65524 Value of z :3
Address of y : 65522 Value of y :2
Address of x : 65520 Value of x :1
```

Explanation: In the above program, `class A` contains two private member variables `x` and `y`. The constructor initializes the member variables. The `class B` is derived from the `class A`. The `class B` has one public member variable. In function `main()`, `b` is an object of class `B`. The pointer `p` is an integer pointer. The address of the member variable `z` of the derived `class B` is assigned to the pointer `p`. By applying a decrease operation, we get the previous memory locations where the member variables of the base class are stored. The values of all member variables are displayed along with their addresses.

11.19 OVERLOADING MEMBER FUNCTION

The derived class can have a similar function name as the base class member function. An object of the derived class invokes the member function of the derived class even if a similar function is preset in the base class.

11.31 Write a program to overload member function in base and derived class.

```
#include<iostream.h>
#include<constream.h>
```

```
class B
{
   public:
   void show()
   {
   cout<<"\n In base class function";
   }
};
class D: public B
{
   public:
   void show()
   {
   cout<<"\n In derived class function";
   }
};
int main()
{
   clrscr();
   B b; // b is object of base class
   D d; // d is object of derived class
   b.show(); // Invokes Base class function
   d.show(); // Invokes Derived class function
   d.B::show(); // Invokes Base class function
   return 0;
}
```

OUTPUT

```
In base class function
In derived class function
In base class function
```

Explanation: In this program, the `class D` is derived from the `class B`. Both the classes have a similar function `show()` as a member function. In function `main()`, the objects of both `classes` B and D are declared. The object `b` invokes the member function `show()`. The object `b` is the object of the base class; hence, it invokes the member function of `show()` of the base class. In addition, the object of the base class cannot invoke the member function of the derived class, because the base class does not have information on the classes derived under it.

The object `d` of the derived class invokes the function `show()`. When functions with a similar name and argument list are present in both the base and derived classes, the object of the derived class gives first priority to the function of its own class. Thus, the statement `d.show()` invokes the function of the derived class. To invoke the function of the base class with the object of the derived class, the class name and scope access operator precede the matching function name. From the above program, it is also clear that we can declare the objects of both base and derived classes. The objects of both base and derived classes are independent of one another.

11.20 ADVANTAGES OF INHERITANCE

(1) The most frequent use of inheritance is for deriving classes using existing classes, which provides reusability. The existing classes remain unchanged. By reusability, the development time of software is reduced.
(2) The derived classes extend the properties of base classes to generate more dominant objects.
(3) The same base classes can be used by a number of derived classes in class hierarchy.
(4) When a class is derived from more than one class, all the derived classes have similar properties to those of base classes.

11.21 DISADVANTAGES OF INHERITANCE

(1) Though object-oriented programming is frequently propagandized as an answer for complicated projects, inappropriate use of inheritance makes programs more complicated.
(2) Invoking member functions using objects creates more compiler overheads.
(3) In class hierarchy, various data elements remain unused, and the memory allocated to them is not utilized.

11.22 MORE PROGRAMS

11.32 Write a program to explain execution of constructor and destructor in single inheritance.

```
#include<iostream.h>
#include<conio.h>
// Constructor, Destructor and Single Inheritance //

class Father
{
   protected:
   char name[20];
   public:
   Father()
   {cout<<"\n Base Class Constructor.";
   cout<<"\n Enter Father Name:";
   cin>>name;
   }
   ~Father()
   {
```

```
   cout<<"\n Base Class Destructor.\n";
   }
};
class Child: private Father
{
   char cname[15];
   public:
   Child()
   {
   cout<<"\n Derived Class Constructor.";
   cout<<"\n Enter Child Name:";
   cin>>cname;
   }
   ~ Child()
   {
   cout<<"\n Derived Class Destructor\n";
   cout<<""<<cname <<""<<name;
   }
};
int main()
{
   clrscr();
   Child C;
   return 0;
}
```

OUTPUT

```
Base Class Constructor.
Enter Father Name : Manoj
Derived Class Constructor.
Enter Child Name : Sanjay
Derived Class Destructor
Sanjay Manoj
Base Class Destructor.
```

Explanation: In the above program, two classes, `class father` and `class child`**,** are declared with one character data member each. Both the classes contain a constructor and a destructor. The `class child` is derived from the `class father`. Hence, the `class father` is the base class, and the `class child` is the derived class.

In function `main()`, `C` is an object of the derived `class child`. The constructor of the base class is executed first followed by the constructor of the derived class. However, the destructor of the derived class is executed first. After this, the destructor of the base class is executed.

Here, the destructor of the derived class prints the contents of both the data members. The base class has no knowledge of the derived class. Hence, it cannot access the data members of the derived class.

11.33 Write a program to explain execution of constructor and destructor in multilevel inheritance.

```
#include<iostream.h>
#include<conio.h>
// Constructor, Destructor and Multilevel Inheritance //

class Grandfather
{
   protected:
   char gname[20];
   public:
   Grandfather()
   {
   cout<<"\n Constructor of class Grandfather\n";
   cout<<"\n Enter Grandfather Name:";
   cin>>gname;
   }
   ~Grandfather()
   {
   cout<<"\n Destructor of class Grandfather\n";
   }
};
class Father : public Grandfather
{
   protected:
   char name[20];
   public:
   Father()
   {cout<<"\n Constructor of class Father\n";
   cout<<"\n Enter Father Name:";
   cin>>name;
   }
   ~Father()
   {
   cout<<"\n Destructor of class Father\n";
   }
};
class Child: private Father
{
   char cname[15];
   public:
```

```
   Child()
   {
   cout<<"\n Constructor of class Child\n";
   cout<<"\n Enter Child Name:";
   cin>>cname;
   }
   ~ Child()
   {
   cout<<"\n Destructor of class Child\n";
   cout<<"\n Grand Father:"<<gname;
   cout<<"\n Father:"<<name;
   cout<<"\n Child:"<<cname;
   }
};
int main()
{
   clrscr();
   Child C;
   return 0;
}
```

OUTPUT

```
Constructor of class Grandfather
Enter Grandfather Name : x
constructor of class Father
Enter Father Name : y
Constructor of class Child
Enter Child Name : z
Destructor of class Child
Grand Father : x
Father : y
Child : z
Destructor of class Father
Destructor of class Grandfather
```

Explanation: In the above program, three classes are declared. They are `grandfather`, `father`, and `child`. Each one of them has a character data-type member. Each class contains a pair of a constructor and a destructor. The class `father` is derived from the `class grandfather`. Similarly, the `class child` is derived from the `class father`. The `class grandfather` is a base class of the `class father`. The `class father` is a base class of the `class child`. The `class child` is derived from the `class father`. The `class father` is an intermediate class that acts as a base class as well as a derived class. Constructors are executed from the base class to the derived class, and destructors are executed from the derived class to the base class.

11.34 Write a program to explain multilevel inheritance with member functions.

```
#include<iostream.h>
#include<conio.h>
// Multilevel Inheritance and member functions//

class Grandfather
{
   private:
   char gname[20];
   public:
   void getg()
   {
   cout<<"\n Enter Grandfather Name:";
   cin>>gname;
   }
   void showg()
   {cout<<"\n Grandfather Name:"<<gname;}
};
class Father : public Grandfather
{
   private:
   char name[20];
   public:
   void getf()
   {
   cout<<"\n Enter Father Name:";
   cin>>name;
   }
   void showf()
   {cout<<"\n Father Name:" <<name;}
};
class Child : public Father
{
   private:
   char cname[15];
   public:
   void getc()
   {
   getg();
   getf();
   cout<<"\n Enter Child Name:";
   cin>>cname;
   }
   void showc()
   {
   showg();
   showf();
   cout<<"\n Child:"<<cname;
   }
};
int main()
{clrscr();
   Child C;
   C.getc();
   C.showc();
   return 0;
}
```

OUTPUT

```
Enter Grandfather Name : XXX
Enter Father Name : YYY
Enter Child Name : ZZZ
Grandfather Name : XXX
Father Name : YYY
Child : ZZZ
```

Explanation: The above program is similar to the previous one. Instead of constructor and destructor members, functions are used.

11.35 Write a program to explain how constructor and destructor are executed in multiple inheritance.

```
#include<iostream.h>
#include<conio.h>

class A
{
   public:
   A() {cout<<"\n Constructor of class A";}
   ~A() {cout<<"\n Destructor of class A";}
};
class B
{
   public:
   B() {cout<<"\n Constructor of class B";}
   ~B() {cout<<"\n Destructor of class B";}
};
```

```
class C
{
   public:
   C() {cout<<"\n Constructor of class C";}
   ~C() {cout<<"\n Destructor of class C";}
};
class D : public A,B,C
{
   public:
   D() {cout<<"\n Constructor of class D";}
   ~D() {cout<<"\n\n Destructor of class D";}
};
int main()
{
   clrscr();
   D x;
   return 0;
}
```

OUTPUT

```
Constructor of class A
Constructor of class B
Constructor of class C
Constructor of class D
Destructor of class D
Destructor of class C
Destructor of class B
Destructor of class A
```

Explanation: In the above program the `class D` is a derived class from base `classes A, B,` and `C`. Each class contains a constructor and a destructor. This type of inheritance is called `multiple inheritance`. The `classes A, B,` and `C` are base classes. The execution of construction depends on the order of the base class given while deriving the class. Consider the statement `class D: public A, B, C`. Here, the execution of constructors starts from base `class A` to base `class C` followed by derived `class D`, If we change the order similar to the statement `class D: public C, B, A,` the execution of construction starts from `class C` to `class A` followed by derived `class D`.

11.36 Write a program to explain execution of constructor and destructor in hybrid inheritance.

```
#include<iostream.h>
#include<conio.h>
// Constructors and Hybrid Inheritance //
```

```
class A
{
   public:
   A() {cout<<"\n Constructor of class A";}
   ~A() {cout<<"\n Destructor of class A";}
};
class B : public A
{
   public:
   B() {cout<<"\n Constructor of class B";}
   ~B() {cout<<"\n Destructor of class B";}
};
class C
{
   public:
   C() {cout<<"\n Constructor of class C";}
   ~C() {cout<<"\n Destructor of class C";}
};
class D : public B,C
{
   public:
   D() {cout<<"\n Constructor of class D";}
   ~D() {cout<<"\n\n Destructor of class D";}
};
int main()
{
   clrscr();
   D x;
   return 0;
}
```

OUTPUT

```
Constructor of class C
Constructor of class A
Constructor of class B
Constructor of class D
Destructor of class D
Destructor of class B
Destructor of class A
Destructor of class C
```

Explanation: The above program is similar to the previous one. The `class B` is derived from `class A`. The `class D` is derived from `classes B` and `C`. This type of inheritance is called `hybrid inheritance`. The execution of constructors is similar to multiple inheritance.

11.37 Write a program to demonstrate single inheritance.

```
#include<iostream.h>
#include<conio.h>

class A
{
   protected:
   int c;
   public:
   A() {c=0;}
   A (int j) {c=j;}
   void show() {cout<<endl<<"c="<<c;}
   void operator ++() {c++;}
};
class B : public A
{
   public:
   void operator --() {c--;}
};
int main()
{
   clrscr();
   B a;
   ++a;
   a.show();
   --a;
   a.show();
   return 0;
}
```

OUTPUT

```
c = 1
c = 0
```

Explanation: In the above program, the `class B` is inherited from `class A`, and hence, the object of the derived `class B` can access the member functions of the base class. When the object is created, the constructor of the base class is executed. The derived class has no constructor. The member of the derived class can access the members of the base class if the access specifier is public or protected. A protected member can be accessed by the member function of a similar class or any class derived from it.

11.38 Write a program to access function derived from private inheritance.

```
#include<iostream.h>
#include<conio.h>

class B
```

```
{
   public:
   void one() {cout<<endl<<"one";}
   void two() {cout<<endl <<"two";}
};
class D : private B
{
   public:
   B::one;
};
int main()
{
   clrscr();
   D d;
   d.one();
   //d.two(); // not accessible
   return 0;
}
```

Explanation: In the above program, the `class B` has two member functions, `one()` and `two()`. The `class D` is derived using private inheritance. The public section of the derived class contains declaration B**::**one that allows the object of the derived class to access the function. The function `one()` is accessible, and the function `two()` is not accessible.

11.39 Write a program to derive classes using multilevel inheritance. Create a common constructor in the lowermost derived class.

```
#include<iostream.h>
#include<conio.h>

class A
{
   public:
   int x;
   A() {}
};
class B : public A
{
   public:
   int y;
   B() {}
};
class C : public B
{
   public:
   int z;
   void show()
   {
   cout<<endl<<"x="<<x <<"y="<<y<<"z="<<z;
   }
   C (int j,int k, int l)
   {
   x=j;
   y=k;
   z=l;
   }
};
int main()
{
   clrscr();
   C c(4,7,1);
   c.show();
   return 0;
}
```

OUTPUT

```
x = 4 y =7 z = 1
```

Explanation: In the above program, `classes A, B,` and `C` are declared. The `class B` is inherited from the `class A`**,** and the `class C` is inherited from the `class B`. The `classes A` and `B` have a zero-argument constructor. The `class C` has three argument constructors and an initialized element of base `classes A` and `B`. Thus, the constructor defined in `class C` acts as a common constructor between these three classes.

In function `main()`, the variable **c** is an object of `class C`**,** and three integers are passed to it. The function `show()` is used to display the contents of the variables.

11.40 Write a program to explicitly call constructor of base classes in multiple inheritances.

```
#include<constream.h>
#include<iostream.h>

struct X
{
   X (int a) // constructor
   {
   cout<<a;
   }
};
struct Y // constructor
```

```
{
   Y (int b) {cout<<b;}
};
struct Z : public X, public Y
{
   Z (int p, int q,int r): X(p), Y(q) // constructor
   {
   cout<<r;
   }
};
int main()
{
   clrscr();
   Z z(1,2,3); // Object of derived class
   return 0;
}
```

OUTPUT

```
123
```

Explanation: The constructors of the base class are executed first followed by the derived class constructor. When a class is derived using multiple inheritance, the execution of base class constructors depends on the sequence of the base class given in the derivation statement. The statement is as follows:

```
struct Z : public X, public Y
```

Here, both X and Y are base classes, and the constructor of class X is executed first followed by the constructor of class Y.

Both base and derived classes contain a one-argument constructor. When an object is declared, three values are passed to it. In the derived class constructor, constructors of both the classes are explicitly invoked with one argument. Thus, arguments of base as well as derived classes are initialized using constructors.

SUMMARY

(1) Inheritance is one of the most useful and essential characteristics of an object-oriented programming language. Inheritance allows the programmer to utilize the previously defined classes with newer ones. The new class is assembled using the properties of the existing classes.

(2) The procedure of developing a new class from an old class is termed `inheritance`.

(3) The new class is termed a `derived class`, and the old class is called a `base class`.

(4) `Single inheritance:` When a new class is derived from only one base

class, such a type of inheritance is called a `single inheritance`.

(5) `Multilevel inheritance:` The procedure of deriving a class from a derived class is called `multilevel inheritance.`

(6) `Multiple inheritance` or `hierarchical inheritance:` When a class is derived from more than one class, this type of inheritance is called `multiple inheritance` or `hierarchical inheritance`.

(7) `Hybrid inheritance:` When a class is derived from another one or more base classes, this process is termed `hybrid inheritance`.

(8) When classes are declared virtual, the compiler takes essential caution to avoid duplication of member variables. Thus, we make a class virtual if it is a base class that has been used by more than one derived class as their base class.

(9) `Execution of constructors in inheritance:` The execution of constructors takes place from base class to derived class.

(10) `Execution of destructors in inheritance:` The execution of destructors is in opposite order as compared with constructors, that is, from derived class to base class.

(11) `Public:` If member variables of a class are public, any function can access them. In C++, the members of a struct and union are by default public.

(12) `Private:` If member variables of a class are private, member functions and friends can only access them, declared in a similar class. The members of a class are by default private.

(13) `Protected:` If member variables of a class are protected, its scope is similar to that for private functions. In addition, member functions and friends can use the member classes derived from the declared class, but only in objects of the derived type.

(14) When a class is not used for creating objects, it is called an `abstract class`.

(15) The constructor of the derived class works for its base class; such constructors are called `common constructors`.

(16) The derived class can have a similar function name as the base class member function. An object of the derived class invokes a member function of the derived class even if a similar function is preset in the base class.

(17) Properties of one class can be used in another class using inheritance or using the object of a class as a member in another class. Declaring the object as a class data member in another class is also known as `delegation.` When a class has an object of another class as its member, such a class is known as a `container class`.

EXERCISES

(A) Answer the following questions

(1) What do you mean by inheritance?

(2) What do you mean by base class and derived class?

(3) Describe the various types of inheritance with examples.

(4) What is the difference between single and multilevel inheritance?

(5) What is the difference between multilevel and hybrid inheritance?

(6) How are constructors and destructors executed in multilevel inheritance?

(7) What is the use of the keyword `virtual`?

(8) What do you mean by `virtual classes`?

(9) What are abstract classes?

(10) Describe the use of public, private, and protected access specifiers.

(11) What are nested classes?

(12) Explain the mechanism for accessing private members of the base class using pointers.

(13) What is the difference between private and protected access specifiers?

(14) Do you think that accessing private data using pointers is a limitation of encapsulation?

(15) How do structures and classes provide inheritance differently?

(16) What is the difference between private and protected inheritance?

(17) What are the advantages and disadvantages of inheritance?

(18) What do you mean by object delegation?

(19) What is a common constructor?

(20) Explain hierarchical inheritance.

(B) Answer the following by selecting the appropriate option

(1) What will be the output of the following program?

```
class A
{public:
int a;
A() {a=10;}
};
class B : public A
{public:
int b;
B() {b=20;}
~ B() {cout<<"\n a="<<a <<"
b="<<b;}
};
void main() {B();}
```

(a) a = 10 b = 20
(b) a = 20 b = 10
(c) a = 30 b = 30
(d) none of the above

(2) Identify the access specifier
(a) public
(b) virtual
(c) void
(d) _class

(3) An object a cannot access the variable

```
class A
{public:
int a;
private:
int b;
public:
A() {a=10,b=20;}
};
void main() {A a;}
```

(a) b
(b) a
(c) both (a) and (b)
(d) both a and b are accessible

(4) Private data members of a class can be accessed by
(a) public member function of a similar class
(b) directly by the object
(c) private member function of a similar class
(d) none of the above

(5) In multilevel inheritance, the middle class acts as
(a) base class as well as derived class
(b) only base class
(c) only derived class
(d) none of the above

(6) In single inheritance, constructors are executed from
(a) base class to derived class
(b) derived class to base class
(c) both (a) and (b)
(d) none of the above

(7) In the following program, the object of which class can access all member variables?

```
struct A {int x;};
struct B: A {int y;};
struct C: B {int z;};
struct D: C {int k;};
```

(a) Object of class D
(b) Object of class B
(c) Object of class C
(d) Object of class A

(8) The protected keyword allows
(a) derived class to access base class members directly

(b) prevents direct access of public members
(c) allows objects to access private members
(d) all of the above

(9) The class is declared virtual when
(a) two or more classes involved in inheritance have a common base class
(b) more than one class is derived
(c) we want to prevent a base class from inheritance
(d) one of the above

(10) The ambiguity of members normally occurs in
(a) single inheritance
(b) multilevel inheritance
(c) multiple inheritance
(d) none of the above

(11) In the following program, class A is an

```
class A {int x;};
class B: A {int y;};
void main() {B b;}
```

(a) abstract class
(b) virtual class
(c) derived class
(d) none of the above

(12) Class A is a base class of class B. The relationship between them
(a) is a kind of a relationship
(b) has a relationship
(c) is a relationship
(d) none of the above

(C) Attempt the following programs

(1) Write a program to define three classes A, B, and C. Each class contains private data members. Derive class C from classes A and B by applying multiple inheritance. Read and display the data using a constructor and a destructor.

(2) Write a program to declare classes X, Y, and Z. Each class contains one character array as a data member. Apply multiple inheritance. Concatenate a string of classes X and Y and store it the class Z. Show all the three strings. Use a constructor and a destructor.

(3) Write a program to calculate the salary of a medical representative based on the sales. Bonus and incentive to be offered to him will be based on total sales. If the sale exceeds Rs.150000/- follow the particulars of Column (1); otherwise, follow those given in Column (2). Apply a conditional constructor and destructor.

Column 1	Column 2

Basic = Rs. 3000.
Hra = 20% of basic.
Da = 110% of basic.
Conveyance = Rs.500.
Incentive = 10% of sales.
Bonus = Rs. 1500.
	Basic = Rs. 3000.
	Hra = 20% of basic.
	Da = 110% of basic.
	Conveyance = Rs.500.
	Incentive = 5% of sales.
	Bonus = Rs. 1000.

(4) Write a program to calculate the energy bill. Read the starting and ending meter reading. The charges are as follows:

No. of Units Consumed	Rates in (Rs.)
200 – 500	4.50
100 – 200	3.50
Less than 100	2.50

(5) The price of a trophy depends on the type of material from which it is created. The following table gives the list of prizes. Write a program to input the serial number of the material. In addition, display the material and its prize.

Serial no.	Material	Prize of trophy
01	Gold	Rs. 20000/-
02	Silver	Rs. 9500/-

03	Steel	Rs. 5000/-
04	Bronze	Rs. 500/-
05	Pewter	Rs. 750/-

(6) A newspaper agent pays two variant rates for the delivery of newspapers. A delivery boy can earn Rs.2/- on a morning delivery but only Rs. 1.50/- on an evening delivery. Boys are salaried either for a morning delivery or for an evening paper round but not both morning and evening paper rounds. Write a program to read (input) the code for rounds (0 for morning and 1 for evening round) and the number of total paper rounds in one week done by a boy; compute his earnings and display it.

(7) The postage for ordinary post is Rs. 2/- for the first 15 grams and Rs. 1 for each additional 10 grams. Write a program to calculate the charge of the postage for a post weighing N grams. Read the weights of N packets and display the total amount of postage.

(8) Declare a class of vehicle. Derived classes are two-wheelers, three-wheelers, and four-wheelers. Display the properties of each type of vehicle using the member function of classes.

(9) Create classes' country, state, city, and village. Arrange these classes in a hierarchical manner.

(D) Find bugs in the following programs

(1)

```
struct A {int x;};
struct B {int y;};
class C: public A, B {};
void main()
{
C c;
c.x=20;
c.y=30;
}
```

(2)

```
struct A {int x;};
struct B: A {int y;};
struct C: A {int z;};
struct D: B,C {int k;};
void main()
{
 D d;
d.x=20;
}
```

(3)

```
struct A
{int x;
struct B {int y;};};
struct C : A::B,A {};
void main()
{C c;
c.x=40;
c.y=20;
}
```

(4)

```
struct A {int x;};
class B : A
{A a;};
void main()
{
B b; b.x=20;
b.a.x=30;
}
```