Unit

7

Inheritance

1. What is Inheritance?

Application programs are big and complex today. One of the techniques used to reduce the coding effort and increase the speed of program development is to reuse code. This is also one of the key concepts of object-oriented programming. This concept is *inheritance*, which is the subject of this chapter. Inheritance is one of the most important object oriented feature.

Definition:

The mechanism of creating new classes from existing ones is called as *inheritance.*

It allows existing classes to be reused. The old class or existing class is known as *base class or parent class* and the newly created class is called as *derived class* or *child class* or *subclass*. The derived class inherits attributes and behavior from the base class and can have additional attributes and functions. Inheritance is described in terms of "**is-a**" relationship.

Example: In the class diagram shown below, class Book inherits from class Media. Hence we can say an object of class Book "is-a" media object. Media is the base class and Book is the derived class.

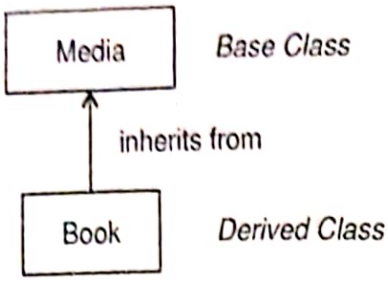


Figure 7.1: Inheritance example

> **Note** The arrow points to the base class to indicate that the derived class "inherits-from" the base class. Some references use the down arrow i.e. pointing to the derived class to indicate that the base class is "inherited-by" the derived class. However, in this book, we shall follow the first method which is the standard UML (Unified Modeling Language) notation.

▶ Advantages of Inheritance

Why should we inherit one class into another? Why not create completely new classes? The main reason to inherit is to reuse code. *For example*, if we want to add more features to the media class which has already been defined and tested, it is better to add new functionality to create a derived class rather than modifying the class or creating a completely new class. The inherited class will have all features of the base class plus its own functionality.

Inheritance provides several advantages which are listed below:

1. Inheritance allows existing classes to be reused; hence allowing reusability of the code.
2. We can provide additional functionality and attributes to the base class.
3. Inheritance avoids redundancy.
4. Provides extensibility of the application.
5. The code becomes easier to manage.
6. Inheritance allows multiple classes to be related by extracting common functionality.
7. The software model becomes closer to real life model with hierarchical relationships.
8. Modules with sufficiently similar interfaces can share a lot of code, reducing the complexity of the program.
9. Data access can be controlled by the use of private, public and protected keywords.

2. Creating a Derived Class

The *syntax* of creating a derived class is shown below.

```
class derived-class: visibility-label base-class1,
visibility-label base-class2...
{
  private:
     // members
   public:
     // members
   protected:
     // members

};
```

- The keyword class
- The name of the derived class
- A colon (:)
- Type of derivation: public, private or protected
- The name of the base classes
- The class members

Visibility mode can either be public, private or protected. If none is specified, the default mode is private.

Example 1: class B derived from A

```
class A                          //base class
{    ...
};
class B : public A               //derived class
{    ...
};
```

Example 2: class C derived from A and B

```
class A                                 //base class
{    ...
};
class C : public A , public B           //derived class
{    ...
};
```

3. Types of Inheritance

We can classify inheritance on the basis of the nature of the class hierarchy. There are five types which are shown in *figure 7.2* below.

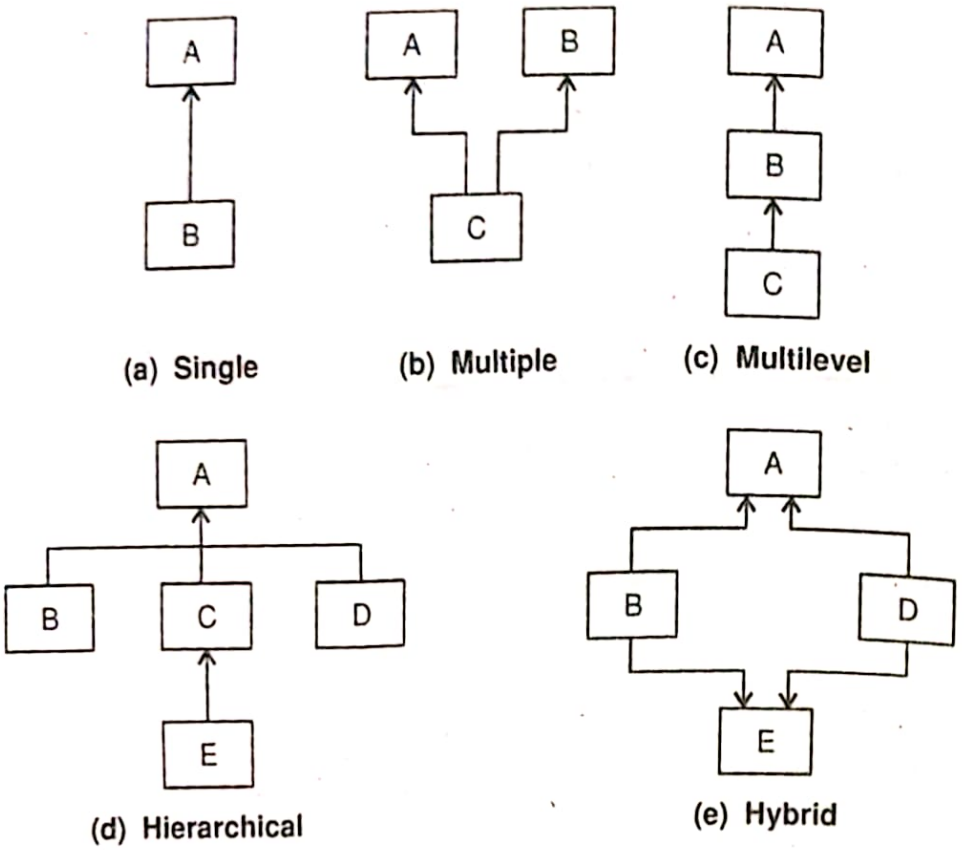


Figure 7.2: Types of inheritance

i. **Single inheritance:** In single inheritance, a class is derived from only one base class.

ii. **Multiple inheritance:** In this type, a derived class inherits from more than one base class.

iii. **Multilevel inheritance:** In multilevel inheritance, a derived class is further used as a base class to create more classes. *For example*, class B inherits from A and C inherits from class B.

iv. **Hierarchical inheritance**: The classes form a hierarchy in which a single base class has many derived classes. These derived classes also act as base classes for further derivation.

v. **Hybrid or Multipath inheritance:** Combines two or more types of inheritance. *For example*, multiple and multilevel inheritance.

The visibility label used i.e. public, private or protected decides the access rights of base class members in the derived class. The private data members of the base class cannot be accessed outside

the class, not even in the derived class. The access to public and protected base class data members in the derived class depends on the label used. Correspondingly, the inheritance types are:

i. **Public inheritance:** In public inheritance, the public and protected data members of the base class remain public and protected respectively in the derived class.

ii. **Private inheritance:** In private inheritance, the public and protected data members of the base class become private in the derived class.

iii. **Protected inheritance:** In protected inheritance, the public and protected data members of the base class are accessible as protected members in the derived class.

The following table shows the access rights of base class data members in the derived class.

Table 7.1: Access rights and inheritance

Inheritance type	Base class member	Access in derived class
private	private	inaccessible
	public	private
	protected	private
public	private	inaccessible
	public	public
	protected	protected
protected	private	inaccessible
	public	protected
	protected	protected

Consider a base class which has:

i. private member x

ii. protected member y

iii. public member z and function displayB()

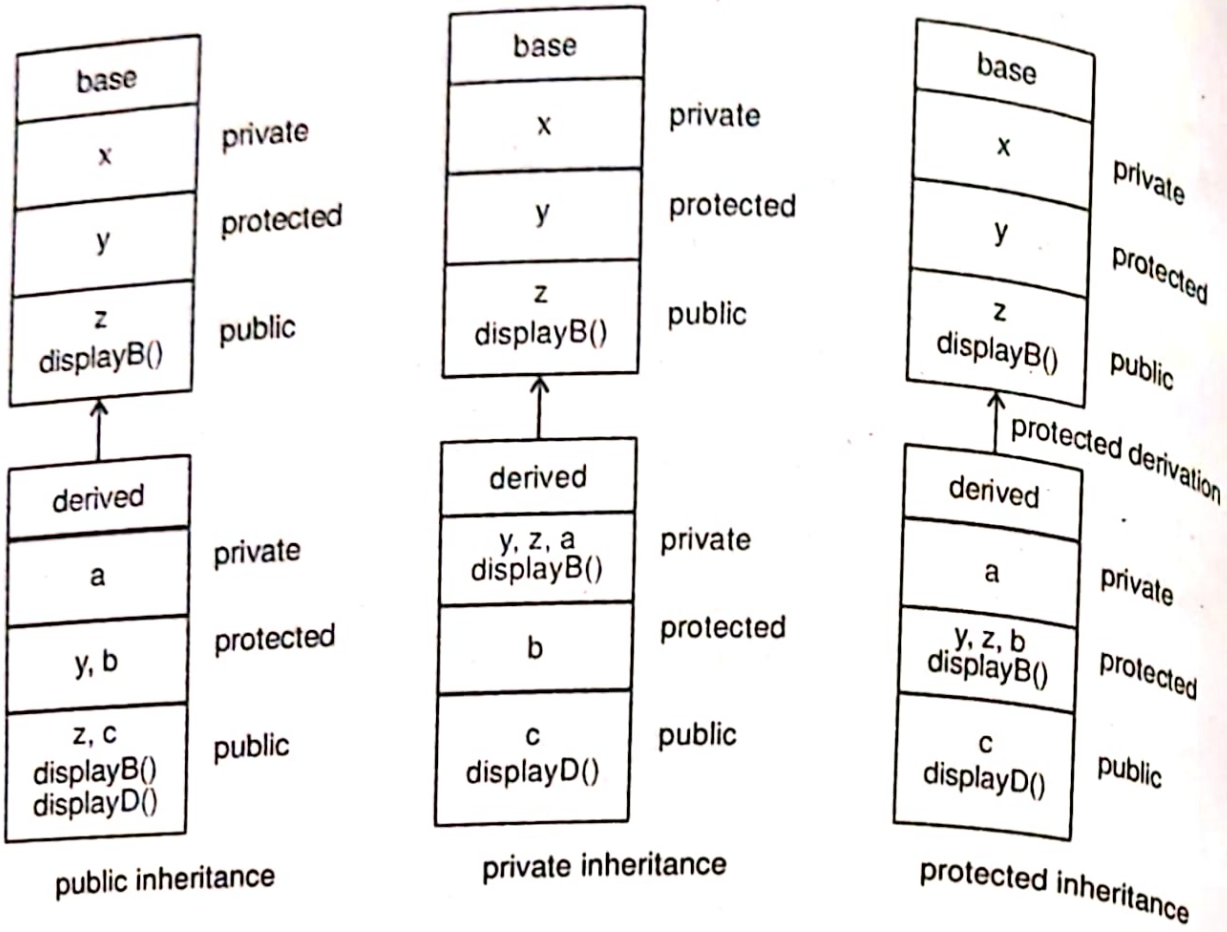
The derived class has:

i. private member a

ii. protected member b

iii. public member c and function displayB()

The following diagram shows the base and derived class member access.



▶ 'protected' Keyword

A derived class inherits properties and functions from its base class (or classes). In many cases, we need to access the base class members directly in the functions of the derived class. As seen earlier, a private base class member is inaccessible to the functions of the derived class. So, we have no choice but to make it public. However, making it public makes the members accessible to functions outside the class thereby violating the data hiding principle of OOP.

To solve this problem, C++ provides the "protected" keyword which gives special privileges to a derived class. A base class member which is declared as protected can be accessed directly by the functions of the derived class.

The following *example* illustrates the use of protected keyword.

```
class base
{
     int a;
     protected:
        int b;          //protected
};
```

```
class derived : public base

void display()
{
   {   cout<<a;      //not allowed
       cout<<b;      //allowed

   }
};
```

3.1 Single Inheritance

In this inheritance, a derived class can only inherit properties from one base class. Consider the following class hierarchy:

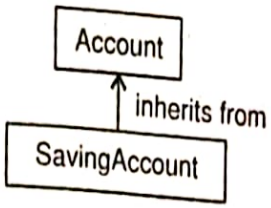


Figure 7.3: Single Inheritance

Base class: **Account** {data members - id, owner and balance, member functions - getaccount() and putaccount() }

Derived class: **SavingAccount** {data members – rate, member functions - getsaving() and putsaving() }

Program : To illustrate single inheritance

```
#include<iostream.h>
class Account              //base class
{
  int id;
  char owner[80];
  double balance;

  public:
  void getaccount();
  void putaccount();
};
class SavingAccount : public Account    //derived class
{
```

```
    double rate;
    public:
        void getsaving();
        void putsaving();
};
void Account::getaccount()
{
    cout<<"Enter id, name and opening balance :";
    cin>>id>>owner>>balance;
}
void Account::putaccount()
{
    cout<<"Id = "<<id<<endl;
    cout<<"Name = "<<owner<<endl;
    cout<<"Balance = "<<balance<<endl;
}
void SavingAccount::getsaving()
{
   getaccount(); //call base class function
   cout<<"Enter rate of interest :";
   cin>>rate;
}
void SavingAccount::putsaving()
{
    putaccount();
    cout<<"Rate = "<<rate<<endl;
}
int main()
{
    SavingAccount sa;
    sa.getaccount();
    sa.getsaving();
    sa.putaccount();
    sa.putsaving();
    return 0;
}
```

Output

```
Enter id, name and
opening balance:
672
DEF
5000
Enter rate of interest :
11.2
ID = 672
Name = DEF
Balance = 5000
Rate = 11.2
```

3.2 Overriding Base-Class Members and Functions

A derived class inherits members from the base class and defines its own members. A derived class may have new members which have the same name as the base class members. This is called **hiding** or **overriding** base class members. A base class member can be hidden by the derived class by

declaring a member in the derived class with exactly the same name. Both, data members as well as functions in the base class can be overridden in the derived class.

It is a common practice to override base class member functions in the derived class. Since a derived class object is a special type of the base class object, it will have the same functionality as the base class. Hence, it is better to keep the function name the same so that they have a common interface. *Function overriding means redefining a base class function in the inheritance hierarchy.* We use function overriding when the derived class needs the same functionality as the base class.

The following rules apply in this case:

1. Whenever a derived class instance or function uses an overridden name, the derived class member will be used by default.
2. The overridden base member function must have the same signature in the derived class.
3. A base class member hidden by a derived class member must be accessed using the syntax:

```
base-class :: hidden-member;
base-class :: hidden-function();
```

For example: In the program given below, we created two classes – Account and SavingAccount. Both classes have functions to accept and display data. We have named the functions as getaccount() and getsaving() respectively. Since, both the functions perform the same task, it would be better to name both functions as getdata(). However, this would lead to ambiguity. The ambiguity can be resolved using the scope resolution operator. In the same manner, we can define a function putdata() in the base class and redefine it in the derived class.

The following program outline shows how overriding can be achieved in the derived class.

```
class Account
{
  //data members
  public:
    void getdata();
    void putdata();

};
class SavingAccount :: public Account
{
  //data members
  public:
    void getdata();     //function overriding
    void putdata();

};
```

```
void Account::getdata()
{
   //code
}
void SavingAccount::getdata()
{
   Account::getdata();   //call to base class function
   //additional code
}
void Account::putdata()
{
   //code
}
void SavingAccount::putdata()
{
   Account::putdata();   //call to base class function
   //additional code
}
void main()
{
   Account a;
   a.getdata();  //calls base class getdata()
   a.putdata();  //calls base class putdata()
   SavingAccount sa;
   sa.getdata(); //calls derived class getdata()
   sa.putdata(); //calls derived class putdata()
}
```

3.3 Multiple Inheritance

In multiple inheritance, a derived class inherits the properties of two or more parent classes (base classes). This means a class can have many base classes.

The following program creates a derived class *stud_proj* which inherits from two classes *student* and *project*. The base class 'student' contains the data members: rollno and name. The other base class '*project*' contains the data members: projectno and title.

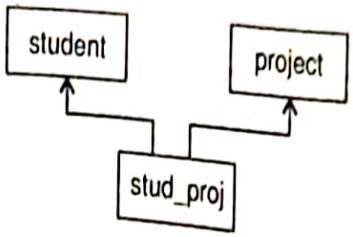


Figure 7.4: Example of Multiple Inheritance

The derived class *stud_proj* contains the data member 'duration' and inherits members from both base classes.

Program : To illustrate multiple inheritance

```
#include<iostream.h>
class student                    // base class 1
{
 int rollno;
 char name[80];
 public:
  void getdata()
  {
    cout<<"Enter student roll number and name :";
    cin>>rollno>>name;
  }
  void putdata()
  {
    cout<<"Student roll number = "<<rollno<<endl;
    cout<<"Student name = "<<name<<endl;
  }
};
class project                    //base class 2
{
 int projno;
 char title[80];
 public:
  void getdata()
  {
    cout<<"Enter project number and title :";
    cin>>projno>>title;
  }
  void putdata()
  {
    cout<<"Project number = "<<projno<<endl;
    cout<<"Project title = "<<title<<endl;
```

```
  }
};
class stud_proj : public student, public project
{
  int duration;
  public:
   void getdata()
   {
     student::getdata();
     project::getdata();
     cout<<"Enter project duration in months:";
     cin>>duration;
   }
   void putdata()
   { student::putdata();
     project::putdata();
   cout<<"Project duration ="<<duration<<"months"<<endl;
   }
};
int main()
{
   stud_proj sp;
   sp.getdata();
   sp.putdata();
   return 0;
}
```

Output

```
Enter student roll number and name:
10      ABC
Enter project number and title:
65      Inventory
Enter project duration in months: 6
Student roll number = 10
Student name = ABC
Project number = 65
Project title = Inventory
Project duration = 6 months
```

Ambiguity in Multiple Inheritance

In some cases, it may happen that the base classes have the same member names. Moreover, the derived class may also override the member. This will cause conflict in the base class members and the derived class member. To avoid ambiguity between the derived class members and base class members, we must use the scope resolution operator :: along with the data members and methods which have the same name.

For example, consider two base classes base1, base2 and a derived class. The two base classes have a data member named 'a'. The following program segment illustrates how ambiguity occurs in the base classes.

Program: Ambiguity in multiple inheritance

```
#include<iostream.h>
using namespace std;
class base1
{
  public:
    int a;
};
class base2
{
  public:
    int a;
};
class derived : public base1, public base2
{
  public:
    int b;
};
void main()
{
  derived objd;
  objd.a = 10;
  objd.b = 20;
}
```

Output

Error: Member is ambiguous base1::a and base2::a

In the above program, there is a conflict between the data member 'a' of both base classes. To avoid such ambiguities, we should use the scope resolution operator.

```
void main()
{
  derived objd;
  objd.base1::a = 40;          // accessing the base1 member
  objd.base2::a = 60;          // accessing the base2 member
  objd.b = 20;                 // accessing the derived class member
}
```

3.4 Multilevel Inheritance

In multilevel inheritance, a derived class acts as a base class for other classes. The following diagram shows class 'A' which serves as a base class for class ' B ' which in turn serves as a base class for class ' C '. Class 'B' is called as intermediate base class as it acts as a link between A and C. Thus, there are two base classes (A and B) and two derived classes (B and C). The advantage of multilevel inheritance is that a derived class can be further inherited.

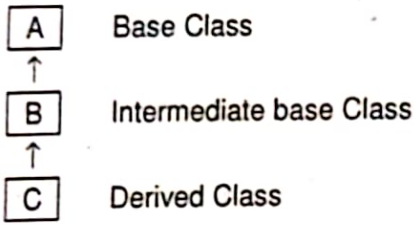


Figure 7.5: Multilevel Inheritance

Declaration of multilevel inheritance:

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

Consider the multilevel hierarchy as shown in *figure 7.6*. Class student stores rollno and name, class exam stores marks obtained in six subjects and class result contains the total marks and percentage. In the result class, we will need access to roll number, name as well as marks.

For this purpose, we can declare these members as protected so that they can be accessed in the derived classes.

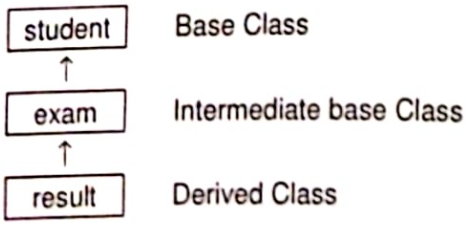


Figure 7.6: Multilevel Inheritance example

Program : To illustrate multilevel inheritance

```
#include<iostream.h>
class student
{
 protected:
   int rollno;
    char name[80];
public:
  void getdata()
  {
  cout<<"Enter student roll number and name :";
  cin>>rollno>>name;
  }
};
class exam : public student
{
 protected:
  int marks[6];
 public:
  void getdata()
  {
  cout<<"Enter marks of 6 subjects :";
  for(int i=0; i<6;i++)
     cin>>marks[i];
  }
};
class sturesult : public exam
{
  float perc;
  public:
   void getdata()
   {
     student::getdata();
     exam::getdata();
     float sum = 0;
     for(int i=0;i<6;i++)
       sum = sum + marks[i];
     perc = sum/6;
   }
   void marklist()
   { cout<<"\n\n\t\t\t MARKLIST" << endl;
     cout<<"==========================================="<<endl;
     cout<<"ROLL NUMBER = "<< rollno << endl;
     cout<<"NAME    = "<< name <<endl;
     cout<<"MARKS   = ";
     for(int i=0; i<6; i++)
       cout<<marks[i]<<"\t";
```

Output

```
Enter roll number and name : 1  ABC
Enter marks of 6 subjects : 85  70     60
    90      65      95
MARKLIST
==================================
ROLL NUMBER  = 1
NAME =  ABC
MARKS = 85 70 60     90     65     75
PERCENTAGE   =  74.166664
==================================
```

```
        cout<<"\nPERCENTAGE = "<< perc<<endl;
        cout<<"=========================================";
    }
};
int main()
{
 sturesult s;
 s.getdata();
 s.marklist();
 return 0;
 }
```

3.5 Hierarchical Inheritance

Hierarchical inheritance is the type of inheritance when derived classes in many levels share the properties of one or many base classes at one level. The class hierarchy follows a tree like structure. As an *example,* consider a hierarchical classification of accounts in a bank as shown in *figure* 7.7. All accounts possess certain common features which are inherited from a single base class Account.

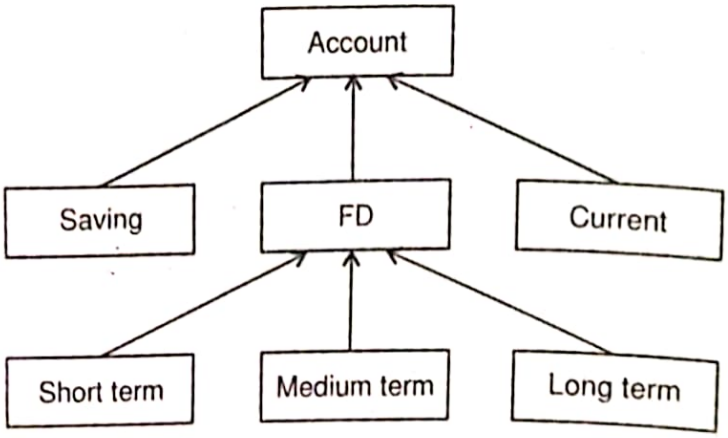


Figure 7.7: Hierarchical classification of bank accounts

As an *example*, a part of the inheritance hierarchy is shown in the class outline below:

```
class account          // Base Class - level 1
{
  . . .
} ;
class savings : public account      // level 2
{
  . . .
} ;
```

```
class FD : public account          // level 2
{
  . . .
} ;
class shortterm : public FD        //level 3
{
  ...
};
class longterm : public FD         //level 3
{
  . . .
} ;
```

The base class data members of level 1 and level 2 which need to be accessed by level 3 derived class functions should be declared **protected**.

3.6 Hybrid or Multipath Inheritance

Hybrid inheritance allows two or more inheritance types to be combined in the same inheritance hierarchy. However, this may lead to complex hierarchies in which a derived class inherits multiple times from the same indirect base class. This is called **Multipath inheritance**.

In the following figure, 'ABC' has two direct base classes 'B' and 'C' which have a common base class 'A'. The 'ABC' class inherits the properties of base class A via two separate paths; one via B and the other via C. The class A is referred as indirect base class and B and C are direct base classes for class ABC.

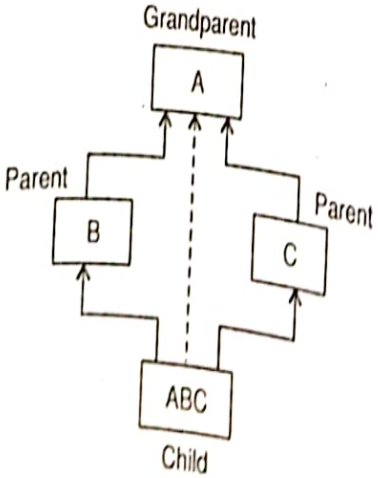


Figure 7.8: Multipath Inheritance

Considering the above class hierarchy, the class outline will be as follows:

```
class A
{
  protected:
    int x;
  . . .
};
class B : public A
{
  . . .
};
class C : public A
{
  . . .
};
class ABC : public B, public C
{
  . . .
};
```

In the above segment, the data member x is inherited twice in the derived class ABC, once through the derived class B and again through C. Hence, class ABC has two copies of the data member 'x'. Since there are two copies of 'x', we will not know which 'x' to use. This will lead to complications and wastage of memory. The following program shows how class ABC inherits two copies of the data member 'x' from class A.

Program : To illustrate multipath inheritance

```
#include<iostream.h>
class A
{
  int x;
};
class B: public A
{
  int y;
};
class C: public A
{
  int z;
};
class ABC: public B, public C
{
  int p;
};
```

```
int main()
{
  cout<<"Size of class A = "<<sizeof(A)<<endl;
  cout<<"Size of class B = "<<sizeof(B)<<endl;
  cout<<"Size of class C = " <<sizeof(C)<<endl;
  cout<<"Size of class ABC = " <<sizeof(ABC)<<endl;
  return 0;
}
```

Output

```
Size of class A = 2
Size of class B = 4
Size of class C = 4
Size of class ABC = 10
```

As seen from the program, the size of class ABC is 10 (x = 2 bytes, x=2 bytes, y=2 bytes, z=2 bytes, p=2 bytes). Note that these outputs will vary on 32 bit machines.

3.7 Virtual Base Class

To prevent a derived class from inheriting multiple data members of the indirect base class, the intermediate base classes should be declared as **virtual base classes**. Any base class which is declared using the keyword *virtual* is called as virtual base class. A virtual base class is an indirect base class whose data members are used by derived classes without duplication.

Virtual base class is a useful method to avoid unnecessary repetition of the same data member in the multiple inheritance hierarchies.

Syntax to create a virtual base class:

```
class class-name : visibility-label virtual base-class
{
      // members

};
```

Note The label and the keyword *virtual* may be used in any order.

Examples:

```
class A
{
  //members
};
class B : public virtual A
{   //members

};
```

```
class C : virtual public A
{
    //members
};
class D : public B, public C
{
    //members;
};
```

In the following class diagram, classes Theory and Practical inherit from the base class – Student. The Result class further inherits from the Theory and Practical classes. Theory and Practical should be declared as virtual base classes.

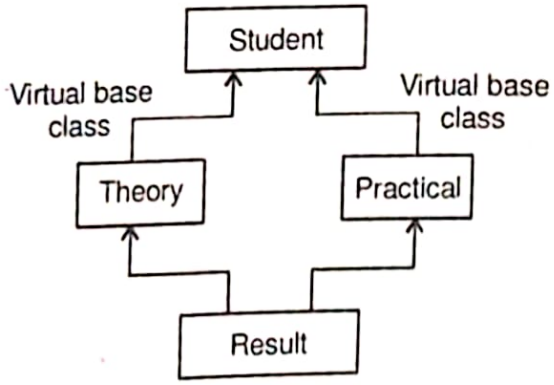


Figure 7.9: Example of Virtual base class

The following program implements the class hierarchy shown in the above diagram. The data members of the base classes are declared protected so that they can be accessed in the derived class Result.

Program : To illustrate virtual base class

```
#include<iostream.h>
class student
{
  protected:
     int rollno;
     char name[80];
  public:
  void getdata()
  {
     cout<<"Enter student roll number and name :";
     cin>>rollno>>name;
  }
};
```

```
class theory: public virtual student
{
protected:
  int marks[6];
public:
void getdata()
 {
   cout<<"Enter marks of 6 subjects :";
   for(int i=0; i<6;i++)
   cin>>marks[i];

 }
};
class practical : public virtual student
{
 protected:
   int marks[3];
 public:
  void getdata()
  {
    cout<<"Enter marks of 3 practicals :";
    for(int i=0; i<3;i++)
    cin>>marks[i];

  }
};
class result: public theory, public practical
{ float perc;
  public:
   void getdata()
   { student::getdata();
     theory::getdata();
     practical::getdata();
     float sum = 0;
    for(int i=0;i<6;i++)
       sum = sum + theory::marks[i];
    for(i=0;i<3;i++)
       sum = sum + practical::marks[i];
   perc = sum/9;
  }
  void marklist()
  {
    cout<<"\n==========  RESULT  ==========="<<endl;
    cout<<"ROLL NUMBER = "<< rollno << endl;
    cout<<"NAME    = "<< name <<endl;
```

```
        cout<<"THEORY MARKS = ";
        for(int i=0; i<6; i++)
            cout<<theory::marks[i]<<"\t";
        cout<<"\nPRACTICAL MARKS = ";
        for(i=0; i<3; i++)
            cout<<practical::marks[i]<<"\t";
        cout<<"\nPERCENTAGE = "<< perc<<endl;
        cout<<"=========================================";
    }
};
int main()
{
    result r;
    r.getdata();
    r.marklist();
    return 0;
}
```

Output

```
Enter roll number and name : 1  ABC
Enter marks of 6 subjects : 85      75      65      55      65      75
Enter marks of 3 practicals : 80 75   93
============== RESULT ==================
ROLL NUMBER  = 1
NAME         =  ABC
THEORY MARKS   =  85     75     65     55     65     75
PRACTICAL MARKS = 80   75     93
PERCENTAGE   =  74.22221
=========================================
```

4. Constructors in Derived Class

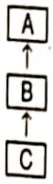
A constructor is used to initialize the objects. In inheritance, when we create a derived class object, the base class constructors are executed first and then the derived class constructor is executed.

The following points should be noted regarding constructors in base and derived classes:

1. If the base class constructor does not take arguments, the derived class need not have a constructor.

2. If the base class has a parameterized constructor, the derived class must define a constructor and pass arguments to the base class constructor.
3. The base class constructors are called and executed before executing the derived constructor.
4. In case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class.
5. In multilevel inheritance, the constructors will be executed in the order of inheritance; from the topmost base class in the hierarchy to the lowest in the hierarchy.

Consider the following *example* of multilevel inheritance:



Here, the constructors will be executed in the order A, B and then C.

Program : To illustrate constructor execution

```
#include<iostream.h>
class A
{
  public:
    A()
    {    cout<<"In Constructor of class A" <<endl;

    }
};
class B : public A
{
  public:
    B()
    {    cout<<"In Constructor of class B" <<endl;

    }
};
class C : public B
{
  public:
    C()
    { cout<<"In Constructor of class C" <<endl;
    }
};
int main()
{  C obj;
   return 0;
}
```

Output

```
In Constructor of class A
In Constructor of class B
In Constructor of class C
```

▶ Passing Parameters to Base Class Constructors

If the base class constructor does not take any arguments, the derived class need not have a constructor. However, if the base class contains a parameterized constructor, then it is compulsory for the derived class to define a constructor and pass the arguments to the base class constructors.

For passing the values to the base class constructors, C++ supports a special argument passing mechanism as shown in the syntax below. The derived class constructor should pass values to the base class constructor(s).

The general form of defining a derived class constructor is:

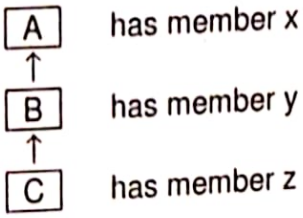
```
derived-constructor(arglist1, arglist2,...,arglistN, arglistD):
base1(arglist1), base2(arglist2),...,baseN(arglistN)
{
  // body of derived constructor
  // using arguments arglistD
}
```

Where,

- arglist1, arglist2,...,arglistN = parameters for base classes base1, base2...baseN
- arglistD = parameters for the derived class
- base1(arglist1), base2(arglist2), ...baseN(arglistN) : are function calls to the base class constructors

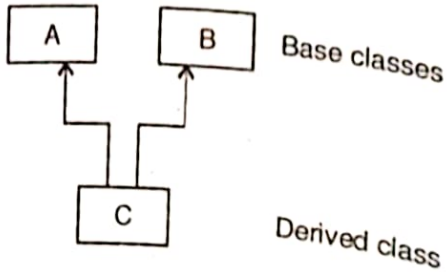
Examples of Constructors in derived classes:

1.



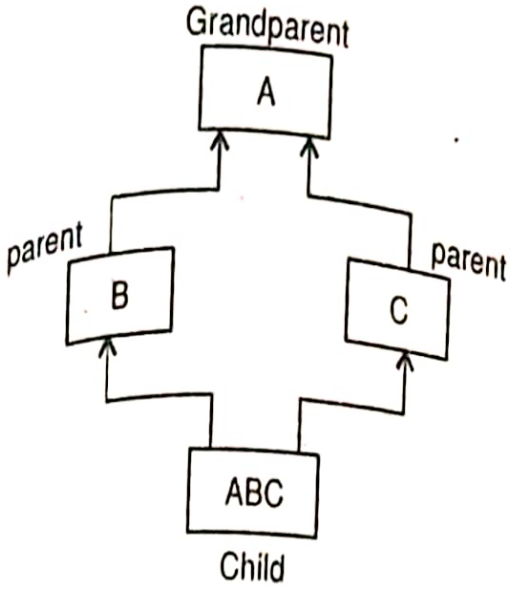

```
A(int x1)
{   x = x1; }
B(int x1, int y1) : A(x1)
{   y = y1 ; }
C(int   x1,   int   y1,   int   z1) :
B(x1,y1)
{   z = z1 ; }
```

2.




```
A(int x1)
{   x = x1; }
B(int y1)
{   y = y1; }
C(int  x1,  int  y1,  int  z1:  A(x1),
B(y1)
{   z = z1 ; }
```

3.




```
Data members: ABC has member p
B(int x1, int y1) : A(x1)
{ y = y1 ; }
C(int x1, int z1) : A(x1)
{ z = z1 ; }
ABC(int x1, int y1, int z1, int p1) : A(x1),
B(x1,y1), C(x1,z1)
{ p = p1 ; }
```

The following program illustrates how parameters are passed to the base class constructor from the derived class constructor. Consider two classes - Employee(id, name, salary) and Manager(department, bonus) which is derived from Employee class.

Program : To illustrate constructors in base classes

```
#include<iostream.h>
#include<string.h>
class Employee

protected:
  int id;
  char name[80];
  float salary;
public:
 Employee()
  {
    id=0; strcpy(name,""); salary=0;
  }
Employee(int id, char *name, float salary)
{
```

```
        this->id=id;
        strcpy(this->name, name);
        this->salary=salary;
    }
    void display()
    {
        cout<<"ID = "<<id<<endl;
        cout<<"NAME = "<<name<<endl;
        cout<<"SALARY = "<<salary<<endl;
    }
};
class Manager : public Employee
{
    char dept[20];
    float bonus;
  public:
    Manager()
    {
      bonus=0; strcpy(dept,"");
    }
    Manager(int id, char *name, float salary, char *dept, float
  bonus):Employee(id, name, salary)
  {
    strcpy(this->dept, dept);
    this->bonus=bonus;
  }
  void display()   //override base class display
  {
    cout<<"ID = "<<id<<endl;
    cout<<"NAME = "<<name<<endl;
    cout<<"DEPT = "<<dept<<endl;
    cout<<"TOTAL SALARY = "<<salary+bonus<<endl;
  }
};
int main()
{
  Manager m(10, "ABC", 20000, " Sales", 5000);
  m.display();
  return 0;
}
```

Output

ID = 10
NAME = ABC
DEPT = Sales
TOTAL SALARY = 25000

5. Destructor in Derived Class

A destructor of a class is a special member function whose purpose is to perform clean-up operations like releasing memory, closing connections etc. It is invoked automatically whenever an object goes out of scope. The constructors are invoked in order from the base class to the derived class.

With destructors, it is the opposite. Destructors in hierarchy are invoked from derived class to the base class i.e. they are executed in the reverse order of the constructors.

The following program illustrates how constructors and destructors are invoked in inheritance.

Program : To illustrate constructor and destructor calls

```
#include<iostream.h>
class A
{
  public:
    A()
    {  cout<<"class A constructor invoked"<<endl;}
    ~A()
    {   cout<<"class A destructor invoked"<<endl; }
     };
class B : public A
{
  public:
    B()   cout<<"class B constructor invoked"<<endl;
    {
    }
    ~B()
    {  cout<<"class B destructor invoked"<<endl; }
};
class C :  public B
{
  public:
    C()
    {  cout<<"class C constructor invoked"<<endl;
     }
    ~C()
    {  cout<<"class C destructor invoked"<<endl; }
};
void main()
{  C obj;
}
```

Output

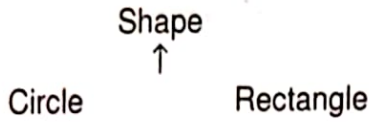
```
class A constructor invoked
class B constructor invoked
class C constructor invoked
class C  destructor invoked
class B destructor invoked
class A destructor invoked
```

6. Abstract Classes

In some cases, we may not want the user to create objects of a base class but only create derived class objects.

A class which is not used to create objects, but is designed only to act as a base class, is called an **abstract class**. It is a class which cannot be instantiated. An abstract class is useful only as a base class to create one or more derived classes. Classes derived from the abstract class must define the pure virtual function.

Consider the following class hierarchy:



In this case, we will be interested in creating objects of the Circle and Rectangle class. We do not want to create objects of the Shape class. The Shape class only serves as a base class and provides data members and functions which are common to both the derived classes. Creating an object of the Shape class will give an error.

To create an abstract base class, the class must have atleast one pure virtual function. Pure virtual functions will be covered in the next chapter.

EXERCISES

A. Multiple Choice Questions

1. The important features of inheritance in C++ is
   a. to extend the capabilities of a class
   b. to hide the details of base class
   c. to facilitate the reusability of code
   d. to provide encapsulation

2. In public derivation, accessibility of members of base class undergo the following modifications in the derived class:
   a. public becomes protected, protected becomes protected and private is not inherited
   b. public becomes public, protected becomes protected and private is not inherited
   c. public becomes protected, protected becomes private, private becomes private
   d. None of the above.

3. If in a derived class constructor you wanted to pass specific parameters to a base class constructor, which one of the following do you use?
   a. Late binding
   b. Virtual functions
   c. Initialization list
   d. Virtual base class

4. An abstract class is a class:
   a. Which cannot be inherited
   b. Which cannot have a virtual function.
   c. Having no data members and functions.
   d. Which cannot be instantiated

5. What is "multiple inheritance"?
   a. When a parent class has two or more child classes
   b. When a base class has two or more derived classes
   c. When a child class has two or more parent classes
   d. Where two classes inherit from each other

6. What C++ syntax is used to declare that a class B is derived from class A?
   a. class A derives B { ... };
   b. class B from A { ... };
   c. class B : public A { ... };
   d. class B subclass of A { ... };

7. What is "inheritance"?
   a. A relationship in which the structure and functionality of a child class is defined in terms of the structure and functionality of the "parent" class.
   b. A relationship in which the structure and functionality of a class (the "parent") is defined in terms of the structure and functionality of another (the "child").
   c. A relationship in which the structure of a class includes one or more instances of another.
   d. A relationship in which the functionality of a class makes calls to the functionality of another.

8. What is the relationship called between a class and its public parent class?
   a. "...is a..." b. "...has a..."
   c. "...uses a..." d. "...becomes a..."

9. If a derived-class object is created and later destroyed what is the order of the constructor and destructor calls on the object.
   a. Base(), Derived(), ... , ~Base(), ~Derived()
   b. Derived(), Base(), ... , ~Derived(), ~Base()
   c. Base(), Derived(), ... , ~Derived(), ~Base()
   d. Derived(), Base(), ... , ~Base(), ~Derived()

B. State True or False

1. Inheritance is used to improve data hiding and encapsulation.
2. When deriving a class from a base class with protected inheritance, public members of the base class became protected members of the derived class.
3. When deriving a class from a base class with public inheritance, protected members of the base class became public members of the derived class.
4. A protected member of a base class cannot be accessed from a member function of the derived class.
5. In case constructors are not specified in a derived class, the derived class will use the constructors of the base class for constructing its objects.
6. A base class cannot be declared an abstract class.
7. A derived class can never be made an abstract class.

C. Review Questions

1. What does inheritance mean in C++? Explain its advantages.
2. What are the different forms of inheritance? Given an *example* for each.
3. How is direct base class different from the indirect base class declaration in C++?
4. Define Multiple Inheritance.
5. What are the syntactic rules to be followed to avoid the ambiguity in single and multiple inheritance?
6. Explain multipath inheritance with an example.

7. Explain the syntax of constructors in derived class.

8. What is an abstract base class?

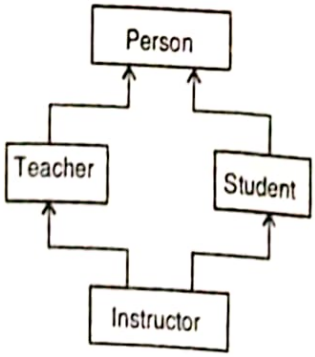
D. Programming Exercises

1. Create a class vehicle having two private members(vehicle number, owner name). Derive a class two-wheeler having one private member (type i.e. scooter/bike etc). Define functions accept and display in both the classes. Write a main function to accept details of n two-wheeler objects (vehicle number, owner name, type).

2. Write class declarations and member function definitions for a C++ base class to represent an employee (code, name, designation).Design derived classes as emp_account(account number, joining date) from employee and emp_sal (basic+pay, earnings ,deduction) from emp_account. Write a menu driven program to:

   i. Build a master table ii. Display all entries iii. Sort list.

3. Design the classes using following hierarchical inheritance.



Each class has member functions accept and display. Write a program to accept details of n instructors and display the details.

4. Implement the following class hierarchy.

Student: id, name,

StudentExam (derived from Student): Marks of n subjects (n can be variable)

StudentResult (derived from StudentExam) : percentage, grade

Define a parameterized constructor for each class and appropriate functions to accept and display data. Create an object of the StudentResult class and display the result in an appropriate format.

Answers

A.

1.	a,c	2.	b	3.	c	4.	d
5.	c	6.	c	7.	a.	8.	a.
9.	c						

B.

1.	False	2.	True	3.	False	4.	False
5.	True	6.	False	7.	True		