The following programme demonstrates how a class *one* has granted its friendship to the class *two*. The class *two* is declared friendly in the class *one*.

```
#include<iostream.h>
#include<conio.h>.
class one
{
    friend class two;
    private:
int x;
public:
void getdata();
};
class two
{
    public:
    void disp(one);
};
    inline void one::getdata()
{
      cout <<"Enter a number" ;
      cin>>x;
}
    inline void two::disp(one obj)
    {
int x;
    cout <<"Entered number is ";
    cout << obj.x;
}
void main()
{
      one obj1;
      two obj2;
      obj1.getdata();
      obj2.disp(obj1);
getche();
}
```

You should see the output as shown below:

Enter a number 25

Entered number is 25

Though the class first has granted its friendship to the class second, it cannot access the private data of the class second through its public member function display( ) of the class first.

A non-member function can be friendly with one or more classes. When a function has declared to have friendship with more than one class, the friend classes should have forward declaration as it needs to access the private members of both classes.

The general syntax of declaring the same friend function with more than one class is :

```
class second;
class first
{
private:
    //data members;
    //member functions;
public:
    //data members;
    //member functions;
friend<return_type><fname>(class first, class second. . ..);
};
class second
{
private:
    //data members;
    //member functions;
public:
    //data members;
    //member functions;
    friend <return_type> <fname>(class first, class second. . ..);
};
```

The following programme demonstrates the use of the friend function which is friendly to two classes. The function seem to calculate the sum of two objects is declared friendly in both the classes.

```
#include<iostream.h>
#include<conio.h>
class two;
class one
{
intx;
public:
void getdata()
```

```
    {
cout <<"Enter the value for x ";
cin >>x;
}
void disp()
{
    cout<< "The value of x entered is "<< x <<"\n".
}
friend int sum(one,two);
};
class two
{
inty;
public:
void getdata()
    {
    cout <<"Enter the value for y ";
    cin>>y;
}
void disp()
    {
      cout<< "The value of y entered is"<< y<< "\n";}
      friend int sum(one, two);
};

int sum(one obj1,two obj2)
{
return(obj1.x + obj2.y);
}
void main()
{
one obj11;
two obj22;
obj11.getdata();
obj11.disp();
obj22.getdata();
obj22.disp();
```

cout < < "the sum of two private data variables x and y is";

```
int tot = sum(obj 11,obj22);
cout<<tot;
getche(); .
}
```

You should see the output as shown below.

Enter the value for x 20

The value of x entered is 20

Enter the value for y 11

The value of y entered is 11

The sum of two private data variables x & y is 31

Note the forward declaration of class second. The non-member function sum() is declared friendly to class first and class second.

6.14 DYNAMIC MEMORY ALLOCATION

Dynamic memory is allocated by the **new** keyword. Memory for one variable is allocated as below:

Ptr = new DataType (initializer);

Here,

- ptr is a valid pointer of type DataType.
- DataType is any valid c++ data type.
- Initializer (optional) if given, the newly allocated variable is initialized to that value.

Example

```
//Example Programme in C++
#include<iostream.h>
void main(void)
{
 int *ptr;
 ptr=new int(10);

 cout<<*ptr;

 delete ptr;
}
```

This is will allocate memory for an int(eger) having initial value 10, pointed by the **ptr** pointer.

Memory space for arrays is allocated as shown below:

ptr=new DataType [x];

Here,

- ptr and DataType have the same meaning as above.
- x is the number of elements and C is a constant.

Example

```
//Example Programme in C++
 #include<iostream.h>

 void main(void)
 {
  int *ptr, size;

  cin>>size;
  ptr=new int[size];

  //arrays are freed-up like this
  delete []ptr;
 }
```

Check Your Progress

Fill in the blanks:

1. A class is a way to bind the ..... and its associated functions together.
2. Objects are the basic ..... entities in an object-oriented system.
3. Static data members are ..... that are common to all objects of a class.
4. The private data values can be neither read nor written by .....

6.15 LET US SUM UP

A class represents a group of similar objects. A class in C++ binds data and associated functions together. It makes a data type using which objects of this type can be created. Classes can represent the real-world object which have characteristics and associated behaviour.

While declaring a class, four attributes are declared: data members, member functions, programme access levels (private, public, and protected) and class tag name. While defining a class its member functions can be either defined within the class definition or outside the class definition. The public member of a class can be accessed outside the class directly by using object of this class type. Private and protected members can be accessed with in the class by the member function only. The member function of a class is also called a method. The qualified name of a class member is a class name: : class member name.

The functions defined inside the class definition are automatically inline. Inline functions are not called, their code replaces their function calls in the programme. The class members are referenced

using objects of the class and the dot operator. For instance, to access member X of an object A of class ABC we will give A.x. If the class definition occurs outside the bodies of all functions, the class is a global class and its objects can be created from any of the functions in a programme. If the class definition occurs within a function, the class is a local class of the function. The objects of the class type can be created only within the function.

A global object can be declared only from a global class whereas a local object can be declared from a global as well as a local class.

The object are created separately to store their data members. They can be passed to as well as returned from functions. The ordinary members functions can access both static as well as ordinary member of a class.

6.16 KEYWORDS

Class: Represents the real-world objects which have characteristics and associated behaviour.

Public Members: Class members (data members and member functions) that can be used by any function.

Private Members: Class members that are hidden from the outside world.

Global Class: A class whose definition occurs outside the bodies of all functions in a programme. Objects of this class type can be declared from anywhere in the programme.

Local Class: A class whose definition occurs inside a function body. Objects of this class type can be declared only within the function that defines this class type.

Friend Function: A function which is not a member of a class but which is given special permission to access private and protected members of the class.

Static Member Functions: Functions that can access only the static members.

Inline Function: A function definition such that each call to the function is, in effect, replaced by the statements that define the function.

Constructor: A member function having the same name as its class and that initializes class objects with legal initial values.

Copy Constructor: A constructor that initializes an object with the data values of another object.

Default Constructor: A constructor that takes no arguments.

Destructor: A member function having the same name as its class but preceded by ~ sign and that deinitializes an object before it goes out of scope.

Temporary Object: An anonymous short lived object.

6.17 QUESTIONS FOR DISCUSSION

1. Define class. What are the differences between structures and classes in C++?
2. Write short notes on:
   i. Member functions,
   ii. Inline functions

3. What is an array? How will you create arrays of objects?
4. Write a note on constructors.
5. What is parameterized constructors?
6. What are the various characteristics of destructor method?
7. Why cannot we pass an object by value to a copy constructor?
8. Write a programme to print the score board of a cricket match in real time. The display should contain the batsman's name, runs scored, indication if out, mode by which out, bowler's score (overs played, maiden overs, runs given, wickets taken). As and when a ball is thrown, the score should be updated.

   (Print: Use separate arrays to store batsmen's and bowler's information)

Check Your Progress: Model Answer

1. Data
2. run-time
3. data objects
4. non-member functions

6.18 SUGGESTED READINGS

Robert Lafore, *Object-oriented Programming in Turbo C++*, Galgotia Publications.

E Balagurusamy, *Object-Oriented Programming with C++*, Tata Mc Graw-Hill

Herbert Schildt, *The Complete Reference C++*, Tata Mc Graw Hill

UNIT IV

LESSON

7

INHERITANCE

CONTENTS

7.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the concept of inheritance
- Discuss the single inheritance
- Describe the types of base classes
- Identify and explain the types of derivations
- Discuss the ambiguity in single inheritance
- Explain the concept of multiple inheritance
- Identify container classes

7.1 INTRODUCTION

Reaccessability is yet another feature of OOP's. C++ strongly supports the concept of reusability. The C++ classes can be used again in several ways. Once a class has been written and tested, it can be adopted by another programmers. This is basically created by defining the new classes, reusing the properties of existing ones. The mechanism of deriving a new class from an old one is called 'INHERITENCE'. This is often referred to as 'IS-A' relationship because very object of the class being defined "is" also an object of inherited class. The old class is called 'BASE' class and the new one is called 'DERIEVED' class.

7.2 SINGLE INHERITANCE

C++ strongly supports the concept of reusability. The C++ classes can be revised in several ways. Once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically clone by creating new classes, revising the properties of the existing ones. The mechanism of deriving a new class from an old one is called inheritance (or derivation). The old class is referred to as the base class and new one is called the derived class. The derived class inherits some or all of the traits from the base class. A class can also inherits properties from more than one class or from more than one level. A derived class with only one base class is called single inheritance.

7.3 TYPES OF BASE CLASSES

We have just discussed a situation which would require the use of both multiple and multi level inheritence. Consider a situation, where all the three kinds of inheritence, namely multi-level, multiple and hierarchical are involved.

Let us say the 'child' has two direct base classes 'parent1' and 'parent2' which themselves has a common base class 'grandparent'. The child inherits the traits of 'grandparent' via two separate paths. It can also be inherit directly as shown by the broken line. The grandparent is sometimes referred to as 'INDIRECT BASE CLASS'. Now, the inheritence by the child might cause some problems. All the public and protected members of 'grandparent' are inherited into 'child' twice, first via 'parent1' and again via 'parent2'. So, there occurs a duplicacy which should be avoided.

The duplication of the inherited members can be avoided by making common base class as the virtual base class: for e.g.

```
class g_parent
{
      //Body
};
class parent1: virtual public g_parent
{
      // Body
};
class parent2: public virtual g_parent
{
```

```
        // Body
};
class child : public parent1, public parent2
{
        // Body
};
```

When a class is virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritence paths exists between virtual base class and derived class. Note that keywords 'virtual' and 'public' can be used in either order.

```
//Program to show the virtual base class
    #include<iostream.h>
    #include<conio.h>
    class student              // Base class declaration
    {
    protected:
      int r_no;
      public:
      void get_n (int a)
      { r_no = a;}
      void put_n (void)
      { cout << "Roll No.: "<< r_no<<"In";}
    };
    class test : virtual public student //Virtually declared common
    {                          //base class 1
    protected:
      int para1;
      int para2;
    public:
      void get_m (int x, int y)
      {     part1= x; part2=y;}
            void put.m (void)
      {
            cout <<"marks obtained: "<< "In";
            cout << "part1 = " << part1 <<"In";
            cout << "part2 = "<< part2 << "In";
      }
    };
```

```
class sports: public virtual student   //virtually declared common
{                                       //base class 2
protected:
  int score;
public:
  void get_s (int a) {
        score = a;
  }
  void put_s (void)
  { cout << "sports wt.: "<<score<< "\n";}
};
class result: public test, public sports     //derived class
{
private : int total;
public:
  void show (void);
};
void result : : show (void)
{ total = part1 + part2 + score;
  put_n ( );
  put_m ( );
  put_s ( );  cout <<"\n total score= "<<total<<"\n";
}
main ( )
{
clrscr ( );
  result S1;
  S1.get_n (345)
  S1.get_m (30, 35);
  S1.get-S (7);
  S1.show ( );
//Program to show hybrid inheritence using virtual base classes
  #include<iostream.h>
  #include<conio.h>
    Class A
    {
```

```
protected:
      int x;
public:
      void get (int);
      void show (void);
};
void A :: get (int a)
      { x = a; }
void A :: show (void)
      { cout << X;}
Class A1 : Virtual Public A
{
protected:
      int y;
public:
      void get (int);
      void show (void);
};
void A1 :: get (int a)
      { y = a;]
void A1 :: show (void)
{
cout <<y;
{
class A2 : Virtual public A
{
protected:
      int z;
public:
      void get (int a)
      { z = a;}
      void show (void)
      { cout << z;]
};
class A12 : public A1, public A2
{
```

```
    int r, t;
    public:
          void get (int a)
          { r = a;}
          void show (void)
          { t = x + y + z + r;
                cout << "result ="<< t;
          }
    };
    main ( )
    {
    clrscr ( );
          A12 r;
          r.A : : get (3);
          r.A1: : get (4);
          r.A2 : : get (5);
          r.get (6);
          r.show ( );
    }
```

7.4 TYPES OF DERIVATIONS

When a class inherits another, the members of the base class become members of the derived class.

Class inheritance uses this general form

```
Class derived-class-name: access base-class-name
{
        // body of class
};
```

The access status of the base-class members inside the derived class is determined by access. The base-class access specifier must be either public, private or protected. If no access specifier is present, the access specifier is private by default if the derived class is class.

7.4.1 Public Inheritance

When the access specifier for a base class is public, all public members of the base become public members of the derived class.

Following program shows the single inheritance using public derivation.

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

```
    {
    int age;
    char name [10];
    public:
    void get( );
    };
    void worker :: get( )
    {
      cout << "your name please";
      cin >> name;
      cout << "your age please";
      cin >> age;
    }
    void worker :: show( )
    {
    cout << "In My name is :" <<name<< "In My age is :" <<age;
    }
    class manager : public worker          //derived class (publicly)
    {
    int now;
    public:
      void get( );
      void show( );
      };
      void manager : : get( )
      {
worker : : get ( );      //the calling of base class input fn.
cout << "number of workers under you";     (could also write:
cin >> now;                              cin>>name>>age)
}                                (if they were public)
void manager :: show( )
{
    worker :: show( );          //calling of base class o/p fn.
    cout >> "in No. of workers under me are: "<< now;
    }                     (could also write: cout<<name<<age)
    main( )
    {
    clrscr( );
```

```
    worker W1;
    manager M1;
    M1.get( );
    M1.show( );
}
```

If you input the following to this program:

Your name please

Ravinder

Your age please

27

number of workers under you

30

Then the output will be as follows:

My name is : Ravinder

My age is : 27

No. of workers under me are : 30

7.4.2 Private Inheritance

When the base class is inherited by using the private access specifier, all public and protected members of base class become private members of the derived class.

The following program shows the single inheritance by private derivation.

```
#include<iostream.h>
#include<conio.h>
class worker                    //Base class declaration
{
   int age;
   char name [10];
   public:
   void get( );
   void show( );
};
   void worker :: get( )
   {
     cout << "your name please";
     cin >> name;
     cout << "your age please";
```

```
    cin >> age;
}
void worker : show( )
{
cout<<"in my name is:"<<name<<"in"<<"my age is :"<<age;
}
class manager : worker //Derived class (privately by default)
{
int now;
public:
   void get( );
   void show( );
   };
   void manager : : get( )
   {
   worker: :get( );  //calling the get function of base
   cout<<"number of worker under you"; class which is
   cin>>now;                           inputting its data
   }                                   members i.e. name
   void manager :: show( )             and age
   {
   worker :: show( );
   cout << "in no. of worker under me are : "<<now;
   }
   main( )
   {
   clrscr( );
         worker w1;
         manager m1;
         m1.get ( );
         m1.show ( );
   }
```

7.4.3 Protect Inheritance

When the base class is inherited by using the protected access specifier, all public and protected members of base class become protected members of the derived class.

The following program shows the single inheritance using protected derivation

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

```
class worker                          //Base class declaration
{ protected:
   int age; char name [20];
   public:
         void get( );
         void show( );
   };
   void worker :: get( )
   {
   cout >> "your name please";
   cin >> name;
   cout << "your age please";
   cin >> age;
   }
   void worker :: show( )
   {
cout << "in my name is: "<< name << "in my age is "<<age
}
class manager : protected worker // protected inheritance
{
int now;
public:
void get( );
void show( );
};
void manager : : get( )
{
cout<<"please enter the name In";
cin>>name;
cout<<"please enter the age In"; //Directly inputting the data
cin>>age;                              members of base class
cout<<" please enter the no. of workers under you:";
cin>>now;
}
void manager :: show( )
{
cout << "your name is : "<<name<<" and age is : "<<age;
cout <<"In no. of workers under your are : "<<now;
```

```
main( )
{
clrscr( );
manager m1;
m1.get( );
cout<< "\n \n";
m1.show( );
}
```

7.5 AMBIGUITY IN SINGLE INHERITANCE

7.5.1 Member Access Control

Members of base class can be accessed using the visibility modes.

Data that are private, protected or public, their accessibility is depend upon, how the class is inherited to the other class.

When the access specifier of the base class is public all public members of the base become public member of the derived class, and all protected members of the base class become protected member of the derived class.

In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class.

7.5.2 Accessing the Private Data

Basically we have visibility modes to specify that in which mode you are deriving the another class from the already existing base class. They are:

a. ***Private:*** When a base class is privately inherited by a derived class, 'public members' of the base class become private members of the derived class and therefore the public members of the base class can be accessed by its own objects using the dot operator. The result is that we have no member of base class that is accessible to the objects of the derived class.

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

c. ***Protected:*** C++ provides a third visibility modifier, protected, which serve a little purpose in the inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by functions outside these two classes.

The table 7.1 summarizes how the visibility of members undergo modifications when they are inherited

Table 7.1

Base Class Visibility ↓	Derived Class Visibility		
	Public	**Private**	**Protected**
Private	X	X	X
Public	Public	Private	Protected
Protected	Protected	Private	Protected

7.5.3 Accessing the Protected Data

When a member of a class is declared as protected, that member is not accessible by other, nonmember elements of the program. With one important exception, access to a protected member is the same as access to a private member–it can be accessed only by other members of its class. The sole exception to this is when a protected member is inherited. In this case, a protected member differs substantially from a private one.

If the base class is inherited as public, then the base class protected members become protected members of the derived class. By using protected, you can create class members that are private to their class but that can still be inherited and accessed by a derived class.

When a derived class is used as a base class for another derived class, any protected member of the initial base class that is inherited (as public) by the first derived class may also be inherited as protected again by a second derived class.

The private and protected members of a class can be accessed by:

a. A function i.e. friend of a class.

b. A member function of a class that is the friend of the class.

c. A member function of a derived class.

7.5.4 Accessing Private Member by Friend Class

It is possible for one class to be a friend of another class. When this is the case, the friend class and all of its member functions have access to the private members defined within the other class.

Let us consider an example:

```
// Using a friend class
# include <iostream>
using namespace std;
class TwoValue {
  intx;
  inty;
  public :
  Twovalue (int a, int b)
{
  x = a;
```

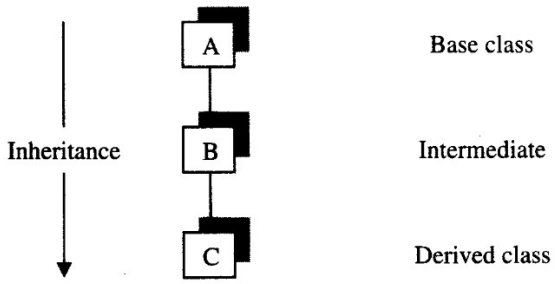
```
    y = b; } ;
friend class Min;
};
class min
{
  public:
  int min (Twovalue p);
};
int Min: min (Twovalue p)
{
  return p.x < p.y? p.x : p.y;
}
int main ( )
{
  Twovalue obj (10, 20);
  Min m;
cout << m.min (obj);
return 0;
}
```

In this example, class Min has access to the private variables x and y declared within the Twovalue class. It is critical to understand that when one class is a friend of another, it only has access to names defined within the other class. It does not inherit the other class. Specifically, the members of the first class do not become member of the friend class.

Friend classes are seldom used. They are supported to allow certain special case situations to be handled.

7.6 MULTIPLE INHERITANCE

When the inheritance is such that, the class A serves as a base class for a derived class B which in turn serves as a base class for the derived class C. This type of inheritance is called 'MULTILEVEL INHERITENCE'. The class B is known as the 'INTERMEDIATE BASE CLASS' since it provides a link for the inheritance between A and C. The chain ABC is called 'INHERITENCE*PATH' for e.g.



The declaration for the same would be:

```
Class A
{
    //body
}
Class B: public A
{
    //body
}
Class C: public B
{
    //body
}
```

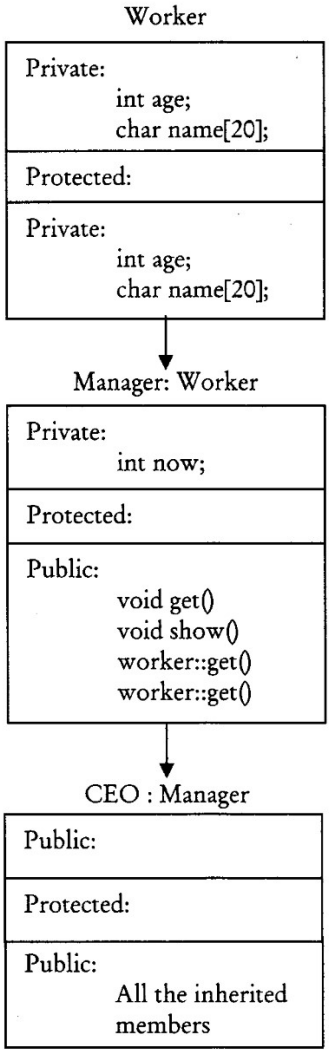
This declaration will form the different levels of inheritance.

Following program exhibits the multilevel inheritance.

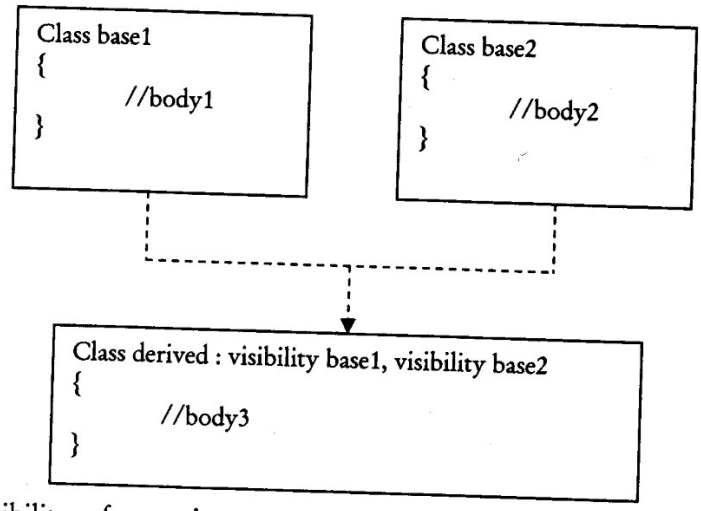
```
#include<iostream.h>
#include<conio.h>
class worker  //Base class declaration
{
int age;
char name [20];
public;
void get();
void show();
}
  void worker::get()
  {
        cout<<"your name please";
        cin>>name;
        cout<<"your age please";
  }
  void worker::show()
  {
cout<<"In my name is :"<<name<<"In my age is :"<<age;
}
class manager : public worker    //Intermediate base class derived
{                                //publicly from the base class
int now;
public:
  void get();
  void show();
};
```

```
    void manager::get()
    {
    worker::get();              //calling get ( ) fn. of base class
    cout << "no. of workers under you:";
    cin >> now;
    }
    void manager : : show ( )
    {
    worker : : show ( ); //calling show( ) fn. of base class
    cout << "In no. of workers under me are: "<< now;
    }
    class ceo: public manager  //declaration of derived class
    {                          //publicly inherited from the
    int nom;                   //intermediate base class
    public:
      void get( );
      void show( );
      }
      void ceo : : get( )
      {
      manager : : get( );
    cout << "no. of managers under you are:"; cin >> nom;
      }
            manager : : show ( );
            cout << "In the no. of managers under me are: In";
            cout << "nom;
      }
      main ( )
      {
      clrscr ( );
            ceo c1;

c1.get ( ); cout << "\n\n";
            c1.show ( );
      }
```



A class can inherit the attributes of two or more classes. This mechanism is known as 'MULTIPLE INHERITENCE'. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like the child inheriting the physical feature of one parent and the and the intelligence of another. The syntax of the derived class is as follows:



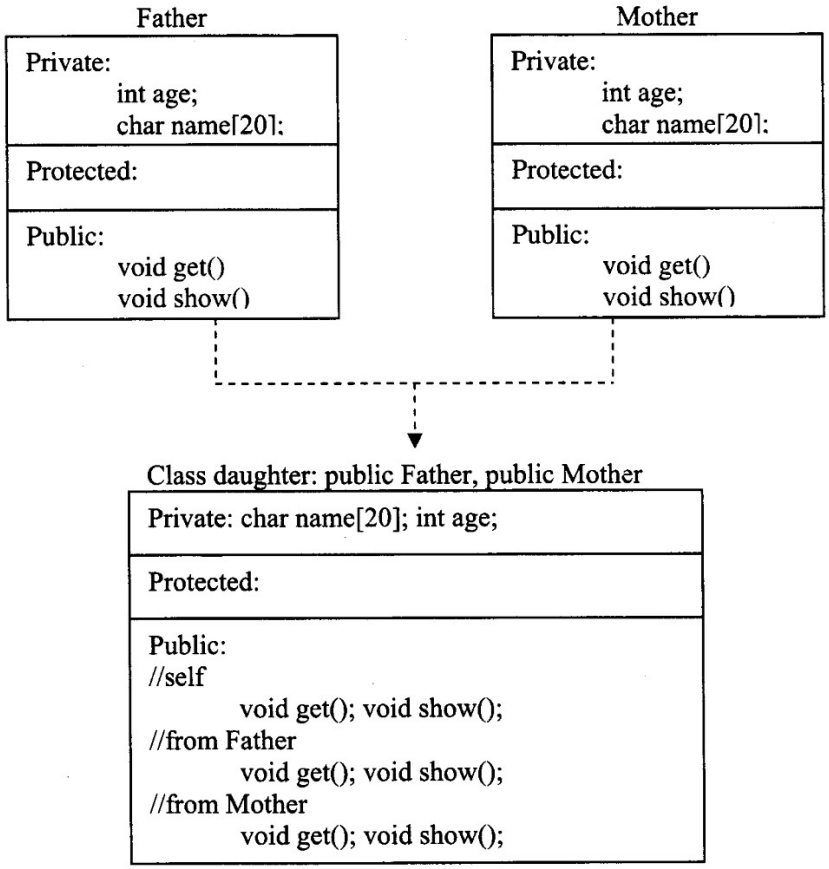
Where the visibility refers to the access specifiers i.e. public, private or protected. Following program shows the multiple inheritance.

```
#include<iostream.h>
#include<conio.h>
class father                    //Declaration of base class1
{
int age;
char name [20];
public:
  void get ( );
  void show ( );
};
void father : : get ( )
{
  cout << "your father name please";
  cin >> name;
  cout << "Enter the age";
  cin >> age;
}
void father : : show ( )
{
cout<<"In my father's name is:"<<name<<"In my father's age is:<<age;
}
class mother                            //Declaration of base class 2
{
```

```
    char name [20];
    int age;
    public:
      void get( )
      {
      cout << "mother's name please" << "In";
      cin >> name;
      cout << "mother's age please" << "in";
      cin >> age;
      }
      void show( )
      {
      cout << "In my mother's name is: "<<name;
      cout << "In my mother's age is: "<<age;
class daughter : public father, public mother //derived class: inheriting
{                              //publicly
char name [20];          //the features of both the base class
    int std;
    public:
      void get ( );
      void show ( );
    };
    void daughter :: get ( )
    {
      father :: get ( );
      mother :: get ( );
      cout << "child's name: ";
      cin >> name;
      cout << "child's standard";
      cin >> std;
    }
    void daughter :: show ( )
    {
      father :: show ( );
      nfather :: show ( );
      cout << "In child's name is : "<<name;
      cout << "In child's standard: "<<std;
    }
```

```
main( )
{
clrscr( );
   daughter d1;
   d1.get ( );
   d1.show ( );
}
```

Diagramatic Representation of Multiple Inheritance is as follows:



7.7 CONTAINER CLASSES

Inheritance is the mechanism of deriving certain properties of one class into another. We have seen in detail how this is implemented using the concept of derived classes. C++ supports get another way of inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects. That is, a class can contain objects of other classes as its member as shown below:

```
class A { };
class B { };
```

```
class C
{
  A a1; // creation of object of class A
  B b1; // creation of object of class B
};
```

All objects of C class will contain the objects a1 and b1. This kind of relationship is called containership or nesting. Creation of an object that contains another object is very different than the creation of an independent object. An independent object is created by its constructor when it is declared with arguments. On the other hand, a nested object is created in two stages. First, the member objects are created using their respective constructors and then the other "ordinary" members are created. This means, constructors of all the member objects should be called before its own constructor body is executed. This is accomplished using an initialization list in the constructor of the nested class.

Example:

```
class C
{
  A a1; // creation of object of class A
  B b1; // creation of object of class B public:
  C (arglist): a1 (arglist 1), b1(arglist 2)
  {
        // constructor body
  }
};
```

arglist is the list of arguments that is to be supplied when a C object is defined. These parameters are used for initializing the members of C. arglist 1 is the argument list for the constructor of a and arglist 2 is the argument list for the constructor of b1. Remember a1 (arglist 1) and b1 (arglist 2) are function calls and therefore the arguments do not contain the data types. They are simply variables or constants.

for example,

```
( (int x, int y, float z) : a(x), (bx, z)
{
  //Assignment section
}
```

We can use as many member objects as are required in a class.

Check Your Progress

Fill in the blanks:

1. The mechanism of deriving a new class from an old one is called .....
2. Members of base class can be accessed using the .....
3. When a class inherits another, the members of the base class become members of the ..... class.
4. A class can inherit the ..... of two or more classes.

7.8 LET US SUM UP

Inheritance is the capability of one class to inherit properties from another class. It supports reusability of code and is able to simulate the transitive nature of real life objects. Inheritance has many forms: Single inheritance, multiple inheritance, hierarchical inheritance, multilevel inheritance and hybrid inheritance.

A subclass can derive itself publicity, privately or protectedly. The derived class constructor is responsible for invoking the base class constructor, the derived class can directly access only the public and protected members of the base class.

When a class inherits from more than one base class, this is called multiple inheritance. A class may contain objects of another class inside it. This situation is called nesting of objects and in such a situation, the contained objects are constructed first before constructing the objects of the enclosing class.

7.9 KEYWORDS

Abstract Class: A class serving only a base class for other classes and no objects of which are created.

Base Class: A class from which another class inherits (also called super class).

Containership: The relationship of two classes such that the objects of a class are enclosed within the other class.

Derived Class: A class inheriting properties from another class (also called sub-class).

Inheritance: Capability of one class to inherit properties from another class.

Inheritance Graph: The chain depicting relationship between a base class and derived class.

Visibility Mode: The public, private or protected specifier that controls the visibility and availability of a member in a class.

7.10 QUESTIONS FOR DISCUSSION

1. Define derived classes.
2. What is multi level inheritance?
3. Write a note on hierarchical inheritance.

4. Consider a situation where three kinds of inheritance are involved.
5. What is the difference between protected and private members?
6. What is the major use of multilevel inheritance?
7. Discuss a situation in which the private derivation will be more appropriate as compared to public derivation.
8. Write a C++ program to read and display information about employees and managers. Employee is a class that contains employee number, name, address and department. Manager class and a list of employees working under a manager.

Check Your Progress: Model Answer

1. inheritance.
2. visibility modes.
3. Derived
4. attributes

7.11 SUGGESTED READINGS

Robert Lafore, *Object-oriented Programming in Turbo* C++, Galgotia Publications.

E Balagurusamy, *Object-Oriented Programming with* C++, Tata Mc Graw-Hill

Herbert Schildt, *The Complete Reference* C++, Tata Mc Graw Hill

LESSON

8

OVERLOADING

CONTENTS

8.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the concept of function overloading
- Define operator overloading
- Describe the overloading of binary operators
- Identify and explain the overloading of unary operators

8.1 INTRODUCTION

C++ provides a rich collection of various operators. We have already seen the meaning and uses of many such operators in previous lesson. One special feature offered by C++ is operator overloading. This feature is necessary in a programming language supporting objects oriented features.

8.2 FUNCTION OVERLOADING

C++ allows you to define several different functions with the same name, provided their parameter list differs. Here's a small example of doing so.

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

```
void print (const char* string)
 cout << "print(\ "  " << string << " \" )" << endl;
void print(int i)
cout << "print ("<< i <<")" << endl;
int main (void)
print ("string printing");     // calls print (const char*)
print (5)                      // calls print(int)
return 0;
```

Compiling and Running Yields

Print ("string printing")

Print (5)

Handled right, this can severely reduce the member of names you have to remember. To do something similar in C, you'd have to give the function different names, like "print_int" and "print_string" of course, handled wrong, it can cause mess. Only overloaded functions when they actually do the same things- in this case, printing their parameters, had the functions been doing different things you would soon forget which of them did what. Function overloading is powerful, and it will be used throughout this course, but everything with power is dangerous, so be careful.

To differentiate between overloaded function, the parameter list is often included when mentioning their names. In the example above, we have the two functions "print (int)" and "print (const char*)", and not just two functions called "print".

Function over loading is a general concept in C++. To understand what is meant by it, consider an example first to illustrate the point.

```
void print (char *p)
cout << "Print a string" << p << "\n".
void print (int:)
cout << "Print an integer "<<p<< "\n";
void Dosomething ( )
char *p [ ] = "My code \0";
int mynumber = 33;
print (p); // call character Print
print (i); // call
```

It makes sense to use one function name for the same functionality to have a good readability of a program, even though different types have to be printed. Above, we have defined two functions print for different arguments and the compiler will decide which one is the right one to take in accordance with the arguments provided. On this level of our C++ knowledge this has only the meaning of good readable programs, but concept becomes essential when we introduce templates in section <mode 40. html>.

When we write function templates we do not know what type a variable will have. That will be specified at compile time. As long as we associate a name with a functionality, which is declared else

where, we can write fully general programs on a high abstraction level. Here is an example of how we can use function overloading already is our Point class

```
Class Point
Private:
double x; // x coordinate
double y; // y coordinate;
Public:
Point ( ) {x = y =0;}; // constructor.
void set (double vx =0) {x = v x;};
Void set (point p) {
x = P.x; // the methods of Point have access to private data
y = P.y; };
```

We use the function name set twice. If we provide an argument of type double, we set the x value, otherwise, we copy the storage of the provided Point into the data section.

In the line set (double vx = 0) { x = vx; }; we used another interesting feature of C++, the default argument: If we use a point p as p.set (7), the x value is set to 7, while the line p.set ( ) would set x to zero.

Scope Rules for Function Overloading

The ability to redefine the building blocks of the language can be a blessing in that it can make your listing more intuitive and readable. It can also have the opposite effect, making it more obscure and hard to understand. Here are few guidelines.

Using Similar Meanings

Use overloaded operators to perform operations that are as similar as possible to those performed on basic data types.

Use Similar Syntax

Use overloaded operators in the same way they are used for basic types. For example, if **alpha** and **beta** are basic types, the assignment operator in the statement,

counter alpha + = beta;

set alpha to the sum of alpha and beta. Any overloaded version of this operator should do something analogous. It should probably do the same thing as

counter alpha = alpha + beta;

where the + is overloaded.

Show Restraint

If you have overloaded the + operator, anyone unfamiliar with your listing will need to do considerable research to find out what a statement like

counter a = b + c;

really means. If the number of overloaded operators grows too large, and if they are used in nonintuitive ways, then the whole point of using them is lost, and the listing becomes less readable instead of more.

Avoid Ambiguity

If you use both a one-argument constructor and a conversion function to perform the same conversion, how will the compiler know which conversion to use? The answer is that it won't. The compiler does not like to be placed in a situation where it doesn't know what to do, and it will signal an error. Avoid doing the same conversion in more than one way.

Not all Operators can be Overloaded

The following operators cannot be overloaded, the member access or dot operator (.), the scope resolution operator (: :), and the conditional operator (? :). And, the pointer-to-member operator (.*), which we have not yet encountered, cannot be overloaded.

8.3 OPERATOR OVERLOADING

Overloading an operator simply means attaching additional meaning and semantics to an operator. It enables an operator to exhibit more than one operations polymorphically, as illustrated below:

You know that additional operator (+) is essentially a numeric operator and therefore, requires two number operands and evaluates to a numeric value equal to the sum of the two operands. Evidently this cannot be used in adding two strings. We can extend the operation of addition operator to include string concatenation. It implies that the additional operator would work as follows:

"COM" + "PUTER"

should produce a single string

"COMPUTER"

This redefining the effect of an operator is called operator overloading. The original meaning and action of the operator however remains as it is.

An operator is overloaded

We will consider overloading a unary operator - minus (-) to enable it to be applicable on a set of numbers instead of a single number.

Function overloading allows different functions with different argument list having the same name. Similarly an operator can be redefined to perform additional tasks. Operator overloading is accomplished using a special function which can be a member function or friend function. The general syntax of operator overloading is:

<return_type> operator <operator_being_overloaded> (<argument list>);

operator is the keyword and is preceded by the return_type of the function.

To overload the addition operator (+) to concatenate two characters, the following declaration, which could be either member or friend function, would be needed:

char * operator + (char *s2);

Rules for Overloading Operators

To overload any operator, we need to understand the rules applicable Let us revise some of them which have already been explored...

Following are the operators that cannot be overloaded.

Operator	Purpose
.	Class member access operator
.*	Class member access operator
::	Scope Resolution Operator
?:	Conditional Operator
Size of	Size in bytes operator
#	Preprocessor Directive

Table 8.1: Operators that cannot be overloaded

Operator	Purpose
=	Assignment operator
0	Function call operator
0	Subscripting operator
->	Class member access operator

- Operators already predefined in the C++ compiler can be only overloaded. Operator cannot change operator templates that is for example the increment operator ++ is used only as unary operator. It cannot be used as binary operator.
- Overloading an operator does not change its basic meaning. For example assume the + operator can be overloaded to subtract two objects. But the code becomes unreachable.

```
class integer
{intx, y;
  public:
  int operator + ();
}
  int integer: : operator + ( )
{
  return (x-y);
}
```

- Unary operators, overloaded by means of a member function, take no explicit argument and return no explicit values. But, those overloaded by means of a friend function take one reference argument (the object of the relevant class).
- Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

Operator to Overload	Arguments passed to the Member Function	Argument passed to the Friend Function
Unary Operator	No	1
Binary Operator	1	2

8.4 OVERLOADING OF BINARY OPERATORS

Binary operators are operators which require two operands to perform the operation. When they are overloaded by means of member function, the function takes one argument, whereas it takes two arguments in case of friend function. This will be better understood by means of the following program.

The following program creates two objects of class integer and overloads the + operator to add two object values.

```
#include<iostream.h>
#include<como.h>
class integer
{
private :
int val;
public:
integer ();
integer(int one );
integer operator+ (integer objb);
void disp ();
};
integer :: integer ()
{
    val = 0;
}
integer:: integer (int one)
{
    val = one;
}
integer integer:: operator+ (integer objb)
{
    integer objsum;
    objsum.val = val+objb. val;
    return (objsum);
}
```

```
void integer:: disp ( )
{
    cout<< "value ="<< val<< endl;
}
void main ()
{
    integer obj1 (11);
    integer obj2 (22);
    integer objsum;
    objsum = obj1 +obj2;
    obj1.disp ();
    obj2.disp ( );
    objsum.disp ( );
    getch ( );
}
```

You should see the following output.

value = 11

value = 22

value = 33

Note that the operator overloading function is invoked by S3=S1+S2. We have passed only one argument to the function. The left hand object S1 invokes the function and S2 is actually the argument passed to the function.

The following program is the same as previous one. The only difference is that we use a friend function to find the sum of two objects. Note that an argument is passed to this friend function.

```
#include<iostream.h>
#include<conio.h>
class integer
{
private:
int val;
public:
integer ();
integer (in tone );
friend integer operator+ (integer obja,integer objb);
void disp ( );
};
integer:: integer ()
```

```
{
    val = 0;
}
integer:: integer (int one)
{
    val = one;
}
integer operator+ (integer obja,integer objb)
{
    integer objsum;
    objsum.val = obja.val+objb.val;
    return (objsum);
}
Void integer :: disp ( )
{
    cout<< "value ="<< val <<endl;
}
void main ( )
{
    integer obj 1 (11);
    integer obj2 (22);
    integer objsum;
    objsum = obj 1 +obj2;
    obj1.disp ();
    obj2.disp ( );
    objsum.disp ( );
    getch ( );
}
```

You should see the following output.

value = 11

value = 22

value = 33

friend function being a non-member function does not belong to any class. This function is invoked like a normal function. Hence the two objects that are to be added have to be passed as arguments exclusively.

8.5 OVERLOADING OF UNARY OPERATORS

In case of unary operator overloaded using a member function no argument is passed to the function whereas in case of a friend function a single argument must be passed.

Following program overloads the unary operator to negate an object. The operator function defined outside the class negates the individual data members of the class integer.

```
#include<iostream.h>
#include<conio.h>
class integer
{
    int x,y,z;
    public:
    void getdata(int a, int b, int c);
    void disp(void);
    void operator- ();   // overload unary operator minus
};
void integer:: getdata (int a, int b, int c)
{
    x=a;
    y=b;
    z=c;
}
void integer::disp (void)
{
    cout << x << " ";
    cout<< y<<" ";
    cout<< z<< "\n";
}
void integer:: operator- ()    // Defining operator- ()
{
    x = -x;
    y =-y;
    z = -z;
}
main 0
{
    integer S;
    S.getdata(11,-21,-31);
```

```
    Cout<< "S: ";
    S.disp ( );
    -S;
    cout<< "s : ";
    S.disp ( );
    getch( );
}
```

You should see the following output.

S: 11 -21 -31

S:-11 21 31

Note the function written to overload the operator is a member function. Hence no argument is passed to the function. In the main() function, the statement -S invokes the operator function which negates individual data elements of the object S.

The same program can be rewritten using friend function. This is demonstrated in the following Program. In this program, we define operator function to perform unary subtraction using a friend function.

```
#include <iostream.h>
#include <conio.h>
class integer
{
    intx;
    int y;
    intz;
    public:
    void getdata(int a, int b, int c);
    void disp(void);
    friend void operator- (integer &s );   // overload unary minus
};
void integer::getdata (int a, int b, int c)
{
    x = a;
    y = b;
    z = c;
}
void integer::disp (void)
{
    cout << x << " ";
    cout << y <<" ";
```

```
    cout << z << "\n";
}
void operator- (integer &s )          // Defining operator- ( )
{
    s.x = -S.x;
    s.y = -s.y;
    s.z = -S.z;
}
main 0
{
    integer S;
    S.getdata(11 ,-21 ,-31);
    Cout<< "S: ";
    S.disp ( );
    -S;                          //activates operator-( )
    cout<< "S : ";
    S.dis ( );
    getch ( );
}
```

You should see the following output.

S: 11 -21 -31

S : -11 21 31

Note how only one argument is passed to the friend function. The operator function declared as friend is not the property of the class. Hence when we define this friend function, we should pass the object of the class on which it operates.

Check Your Progress

Fill in the blanks:

1. Overloading an operator simply means attaching additional meaning and ..... to an operator.
2. There are no operators for manipulating the .....
3. Binary operators are operators which require ..... operands to perform the operation.
4. In case of unary operator overloaded using a member function no ..... is passed to the function.

8.6 LET US SUM UP

In this lesson, we have seen how the normal C++ operators can be given new meanings when applied to user-defined data types. The keyword operator is used to overload an operator, and the resulting operator will adopt the meaning supplied by the programmer.

Closely related to operator overloading is the issue of type conversion. Some conversions take place between user defined types and basic types. Two approaches are used in such conversion: a one argument constructor changes a basic type to a user defined type, and a conversion operator converts a user-defined type to a basic type. When one user-defined type is converted to another, either approach can be used.

8.7 KEYWORD

Operator overloading: Attaching additional meaning and semantics to an operator. It enables to exhibit more than one operations polymorphically.

8.8 QUESTIONS FOR DISCUSSION

1. What is operator overloading?
2. How many arguments are required in the definition of an overloaded unary operator?
3. When used in prefix form, what does the overloaded + + operator do differently from what it does in postfix form?
4. Write the complete definition of an overloaded + + operator that works with the string class from the STRPLUS example and has the effect of changing its operand to uppercase. You can use the library function toupper ( ), which takes as its only argument the character to be changed, and returns the changed character.
5. Write a note on unary operators.
6. What are the various rules for overloading operators?

Check Your Progress: Model Answer

1. Semantics
2. Strings
3. Two
4. argument

8.9 SUGGESTED READINGS

Robert Lafore, *Object-oriented Programming in Turbo C++*, Galgotia Publications.

E Balagurusamy, *Object-Oriented Programming with C++*, Tata Mc Graw-Hill

Herbert Schildt, *The Complete Reference C++*, Tata Mc Graw Hill

UNIT V

LESSON

9

POLYMORPHISM

CONTENTS

9.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the concept of polymorphism
- Discuss polymorphism with pointers
- Describe the significance of virtual functions
- Identify and explain the late binding abstract classes
- Discuss the constructors and destructors under inheritance
- Explain the concept of virtual destructors
- Discuss the virtual base classes

9.1 INTRODUCTION

Polymorphism is an Anglicization of two Greek words - 'poly' means many, and 'moop' meaning shape. Polymorphism allows the program to use the exactly same function name with the exactly same argument in both a base class and its subclasses. This allows having a function that behaves in a different way depending upon the object class.

Polymorphism may be the most powerful aspect of object-oriented programming because the program can be written to call member functions without regard for what class the object belongs to. This can be done by defining the sub-class.

9.2 POLYMORPHISM

Polymorphism refers to the implicit ability of a function to have different meanings in different contexts. Consider the class hierarchy that contains Number and ComplexNumber. If a number is defined as a pointer to Number then a Number can be instantiated as either a Number or a ComplexNumber.

```
Eg:   Number *aNumber;
    aNumber = new Number (1);
    aNumber → output ( );
    delete  aNumber;
    aNumber → output ( );
    Delete aNumber;
```

In the first case, the output function in Number would be called and in the second case, it would be called again. This happens because a Number is a pointer to a Number and not a ComplexNumber. To solve this problem, both output functions must be declared as virtual. Then the compiler keeps track of which the actual functions associated with each object and calls the appropriate ones when needed.

9.3 POLYMORPHISM WITH POINTERS

One of the key features of derived classes is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature, that brings Object Oriented Methodologies to its full potential.

We are going to start by rewriting our program about the rectangle and the triangle of the previous section taking into consideration this pointer compatibility property:

```
pointers to base class
#include <iostream>
using namespace std;
class CPolygon {
  protected:
    int width, height;
  public:
```

```
    void set_values (int a, int b)
      { width=a; height=b; }
  };
class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

In function `main`, we create two pointers that point to objects of class `CPolygon` (`ppoly1` and `ppoly2`). Then we assign references to `rect` and `trgl` to these pointers, and because both are objects of classes derived from `CPolygon`, both are valid assignment operations.

The only limitation in using `*ppoly1` and `*ppoly2` instead of `rect` and `trgl` is that both `*ppoly1` and `*ppoly2` are of type `CPolygon*` and therefore we can only use these pointers to refer to the members that `CRectangle` and `CTriangle` inherit from `CPolygon`. For that reason when we call the `area()` members at the end of the program we have had to use directly the objects `rect` and `trgl` instead of the pointers `*ppoly1` and `*ppoly2`.

In order to use `area()` with the pointers to class `CPolygon`, this member should also have been declared in the class `CPolygon`, and not only in its derived classes, but the problem is that `CRectangle` and `CTriangle` implement different versions of `area`, therefore we cannot implement it in the base class.

9.4 VIRTUAL FUNCTIONS

Virtual functions, one of advanced features of OOP is one that does not really exist but it appears real in some parts of a program. This section deals with the polymorphic features which are incorporated using the virtual functions.

The general syntax of the virtual function declaration is:

```
class use_defined_name{
private:
public:
virtual return_type function_name1(arguments);
virtual return_type function_name2(arguments);
virtual return_type function_name3(arguments);
--------------
--------------
};
```

To make a member function virtual, the keyword virtual is used in the methods while it is declared in the class definition but not in the member function definition. The keyword virtual precedes the return type of the function name. The compiler gets information from the keyword virtual that it is a virtual function and not a conventional function declaration.

For example, the following declaration of the virtual function is valid.

```
class point {
intx;
inty;
public:
virtual int length ( );
virtual void display ( );
};
```

Remember that the keyword virtual should not be repeated in the definition if the definition occurs outside the class declaration. The use of a function specifier virtual in the function definition is invalid.

For example

```
class point {
intx;
inty;
public:
virtual void display ();
};
virtual void point: : display () //error
{
```