UNIT 8 : INTRODUCTION TO CLASSES AND OBJECT

UNIT STRUCTURE

8.1 Learning Objectives
8.2 Introduction
8.3 Classes in C++
8.4 Class Declaration
    8.4.1 Access Control in Class
8.5 Declaring Objects
    8.5.1 Accessing Class Members
8.6 Defining Member Functions
    8.6.1 Member Function inside a Class
    8.6.2 Member Function outside a Class
8.7 Inline Member Function
8.8 Array of Objects
8.9 Objects as Function Argument
    8.9.1 Pass by Value
    8.9.2 Pass by Reference
    8.9.3 Pass by Pointer
8.10 Friend Function and Friend Class
8.11 Static Data Member and Member Function
8.12 Let Us Sum Up
8.13 Further Reading
8.14 Answers to Check Your Progress
8.15 Model Questions

8.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- identify the basic components of a class
- define a class and create objects
- define member function of a class
- describe array of objects
- use objects as function arguments

8.2 INTRODUCTION

So far, we have learnt that C ++ lets you create variables which can be of a range of basic data types : *int, long, double* and so on. However, the variables of the basic type do not allow you to model realworld objects (or even imaginary objects) adequately. We come to know that the basic theme of the object oriented approach is to model the real –world problems. So, object oriented programming language C++ introduces a new data type called ***class*** by which you can define your own data types as *class*. Defining the variables of a class data type is known as a class instantiation and such variables are called ***objects***. In this unit, we will introduce you how to declare a class and how to create objects of a class.We will also discuss how a member function declare inside a class or outside a class and how it can be accessed. Moreover, the techniques of passing objects as function arguments are also illustrated in this unit.

8.3 CLASSES IN C++

We are already familiar with the term encapsulation which is a fundamental feature of OOP. The encapsulation is nothing but a mechanism that binds together the data and functions into a single component, and keeps both safe from outside interference and misuse.

The data cannot be accessible by outside functions. With encapsulation data hiding can be accomplished.

In C++, the encapsulation is supported by a construct called- "*class*". First, let us think a bit about-what is an object . From the general concept, we can say that an object is something that has fixed shape or well defined boundary. In other words, an object can be a person, a place, or a thing with which the computer must deal. If you look at your surroundings some objects may correspond to real-world entities such as– student, bank account, book, cars, bags, computer, lock, watch etc. You will observe two characteristics about objects–

(i) each objects has certain attributes.

(ii) each objects has some behaviours or actions or operations associated with it.

For example, the objects 'computer' & 'car' have the following attributes and operations–

- Object : car

  Attributes: company, model, colour, & capacity

  Operation: speed ( ), average ( ), & break ( )

- Object: Computer

  Attributes: brand, price, monitor resolution, hard disk and RAM size

  Operation: Processing( ), display( ) & printing ( )

Each object will have its own identity though its attributes and operation are same, the objects will never become equal. In case of person object, for instance, two person have the same attributes like name, age and sex, but they are not equal. Objects are the basic run time entities in an object – oriented system. Thus, in C++, an object is a collection of **related variables** and **function** bound together to form a high level entity. Thus,

- The variable defines – the state of the object
- function defines – the action or operation that can be performed on the object.

Now, let us come back to the discussion of **class**.

A **class** is a grouping of objects having identical attributes and common behaviour (operations). It means all objects possessing similar attributes or properties are grouped into the same unit which is called a class. A class encloses both the data and function into a single unit as shown in the following Figure 8.1

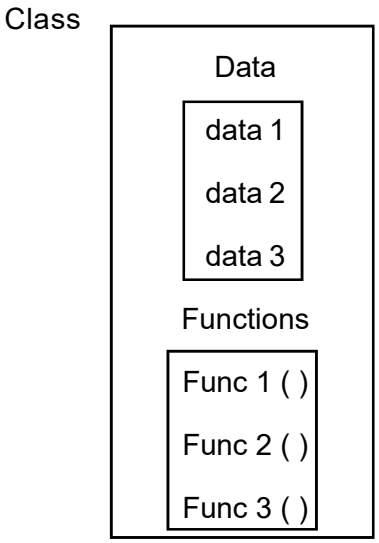
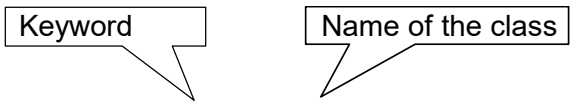


Figure 8.1 Grouping of data and function in a class

Thus, the entire group of data and code of an object can be built as a user-define data type using class. It is obvious that classes are the basic language construct in C++ for creating the user defined data types. Once a class has been declared, the programmer can create a number of objects associated with that class. Objects are nothing but the variable of the class data type. Defining variables of a class data type is known as a **class instantiation.** The syntax used to create an object of the class data type is similar to the syntax used to create an integer variable in C. In the next section, we will learn how to declare a class and an object.

8.4 CLASS DECLARATION

We have come to know that classes contain not only data but also functions. Data and functions within a class are called members of a class. The data inside a class is called a **data member** and the functions are called **member function**. The member functions define the set of operations that can be performed on the data member of a class. The syntax of a class declaration is shown below—




```
class <Class Name>
    {
    private :
        data member;
        member function;
    public:
        data member;
        member function;
    };
```

The keyword **'class'** indicates the name which follows class name, is an abstract data type. The declaration of a class is enclosed with curly braces and terminated by a semi-colon. The body of a class contains declaration of data members and member functions.

The members of a class are usually grouped in two sections i.e. *private* and *public*, which define the visibility of members.

The following declaration illustrates a class specification:

```
class employee
    {
    Private :
        char name[30];
        float age;
    Public :
        void insert_data (void);
        void show_data (void);
    };
```

A class name should be meaningful, reflecting the information it holds. Here in our example, the class 'employee' contains two data members and two member functions. The member function *insert_data( )* is used to assign value to the member variable or data member 'name' and 'age'. The member function *show_data( )* is used to display the values of the

data members. The data member of the class 'employee' cannot be accessed by any other functions that are defined outside the class. It means only the member functions of a class are permitted to access its data members.

In general, the data members are declared as private and member functions are declared as public. In our example, though, we have not specifed the data members as private; yet they are treated as private by default.

Figure 8.2 represent the class 'employee'.

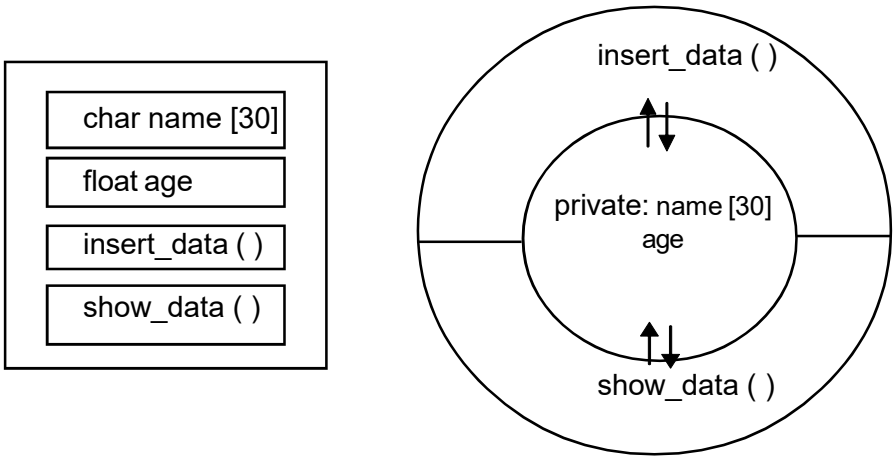


Figure 8.2 Representation of 'employee' class

8.4.1 Access Control in a Class

The members of a class are generally grouped into three sections by using the following keyword–

- private
- public
- protected

These keywords are called ***access control specifiers***. These control specifiers are written inside the class and terminated by this **':'**

symbol. All the members that follow a keyword (upto another keyword) belong to that type. If no keyword is specified, then the members are assumed as private member. We will discuss later about the protected keyword. Let us now briefly discuss about private and public keywords.

- **Private**: Private members are accessible to their own class members only. They cannot be accessed from outside the class by any member functions. The members at the beginning of class without any access specifier are private by default. Hence, writing the keyword 'private' at the beginning of a class is optional.

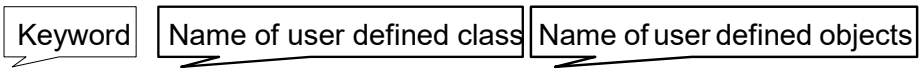
- **Public:** Public members are visible (accessible) outside the class, they should be declared in public section. All data members are not only accessible to their own members of a class but also can be accessible from anywhere in the program, either by functions that belong to the class or by those external to the class.

8.5 DECLARING OBJECTS

A class declaration only builds the structure of objects. In our example, the class *employee* does not define any objects of employee but only specifies what it will contain. Once a class has been declared, we can create variable of that type by using the class name (like any other built-in type variable). The process of creating objects (variables) of the class is called ***class instantiation***. For example–

```
int x, y, z;          // declaration of integer variables
float m, n; // declaration of float variables
employee a; // declaration of object or class variable
```

The syntax for creating objects are shown below :



class classname object name, ……

Remember, the use of the keyword *'class'* is optional at the time creating objects.

For example, ***class employee e1,***

or

employee e1;

In a single statement we can create more than one object as shown below.

employee e1, e2, e3, e4;

As in the case of structures, we can create objects by placing their names immediately just after the closing braces as follows –

```
class employee
    {
        // body of the class
    } e1, e2, e3;
```

This practice is rarely followed because we would like to define the objects as and when required, or at the point of their use.

Always remember that, at the time of declaration of object, necessary memory space is allocated for an object. Suppose, we have declared two objects as –

employee e1, e2;

Both ***e1*** and ***e2*** have the same data members and it is illustrated by the following figure.

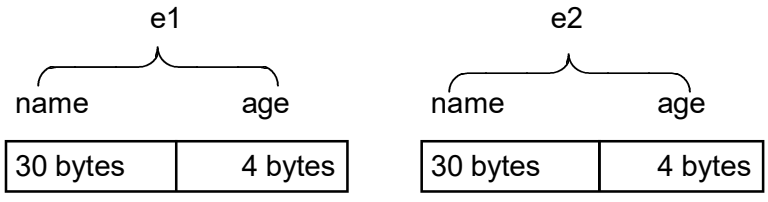


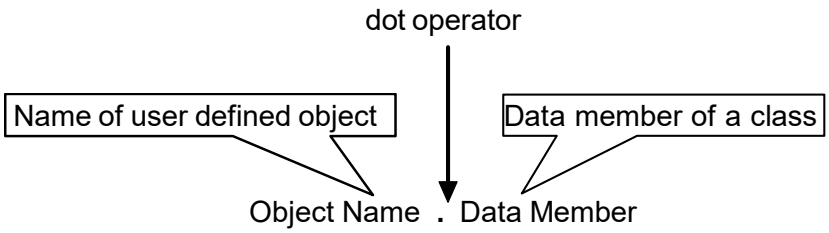
Fig. 8.3 : Allocation of memory for objects

Here, in the figure the objects **e1** and **e2** occupy the memory area. They are not initialized to anything; however the data members of each object will simply contain junk values. So, our main task is to access the data members of the object and setting them to some specific values.
An object is a conceptual entity having the following properties:

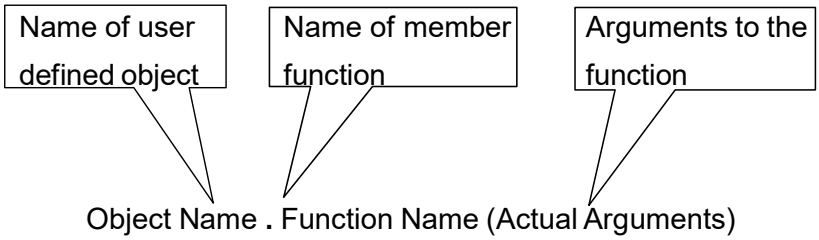
- it is individual
- it points to a thing, either physical or logical that is identifiable by the user.
- it holds data as well as operation method that handles data.
- its scope is limited to the block in which it is defined.

8.5.1 Accessing Class Members

After creating an object of a class, it is the time to access its members. This is the achieved by using the member access operator, dot (.). The syntax for accessing members (data and functions) of a class is shown below–



(a) Accessing data member of a class



(b) Accessing member function of a class

The following program demonstrates how the objects are used for accessing the class data members.

//Program 8.1

```
# include <iostream.h>
# include <string.h>
# include <conio.h>
class employee
{
    private:
    char name [30];          // name of an employee
    float age; // age of an employee
    public :   // initializing data members
    void insert_data (char * name1, float age)
    { strcpy (name, name1);
          age = age1;
    }
    void show_data ( )      //displaying the data members
    {
      cout << "Name :"<<name<<endl;
      cout << "Age :"<<age<<endl;
    }
};
void main ( )
{
    employee e1; //first object of class employee
    employee e2; //second object of class employee
    clrscr( );
    e1.insert_data("Hemanga",30);
    //e1 calls member insert_data ( )
    e2.insert_data ("Prakash",32);
    //e2 calls member insert_data ( )
    cout << "Employee Details:"<<endl;
    e1.show_data ( ); // e2 calls member show_data (
)
    e2.show_data ( ); // e2 calls member show_data (
)
      getch ( );
    }
```

OUTPUT : Employee Details:

Name : Hemanga
Age : 30
Name : Prakash
Age : 32

In the above program, in main( ) we have declared two objects through the statements

```
employee e1;      and   employee e2;
```

The statements

```
e1.insert_data ("Hemanga", 30);
e2.insert_data ("Prakash", 32);
```

initialize the data members of object e1 and e2. The object e1's data member 'name' is assigned 'Hemanga' and age is assigned 30. Similarly, the object e2's data member 'name' is assigned 'Prakash' and age is assigned 32.

The statements

```
e1.show_data ( );
e2.show_data ( );
```

call their member show_data ( ) to display the contents of data members namely, 'name' and 'age' of employee objects e1 and e2. The two objects e1 and e2 will hold different data values.



CHECK YOUR PROGRESS

1. Answer the following by selecting the appropriate option:

   a) The members of a class are by default.

   (i) Private (ii) Public
   (iii) Protected (iv) None of these

   b) The private data of any class is accessed by -

   (i) Only public member function
   (ii) Only private member function

(iii) Boths (i) & (ii) (iv) None of these

c) Encapsulation means

(i) Protecting data (ii) Allowing global access

(iii) Data hiding (iv) Both (i) & (iii)

d) The size of object is equal to

(i) Total size of member data variables

(ii) Total size of member functions

(iii) Both (i) & (ii) (iv) None of these

e) In a class, only member function can access data which is not accessible to out side. This feature is called

(i) data security (ii) data hiding

(iii) data manipulation (iv) data definition

8.6 DEFINING MEMBER FUNCTIONS

We have already come to know that a class holds both the data and functions which are called *data members and member functions*. The data member of a class must be declared within the body of the class. The member functions of a class can be defined in two places

- inside the class definition
- outside the class definition

It is obvious that a function should perform the same task, no matter where it is defined. But the syntax of a member function definition changes depending on the place of its definition, i.e. inside a class or outside a class. We will now discuss both the approaches.

8.6.1 Member Function Inside a Class

In this method, the function is defined inside the class body. When a function is defined inside a class, it is treated as an **inline** function. We will discuss inline function in the next section.

In the program 8.1 we have defined the member functions–

```
void insert_data (char * name1, float age);
```

and

void show_data ( );

inside the class 'employee'. We have seen that these function definition are similar to the normal function definition except that they are enclosed within the body of a class. They will be treated as an inline function. Remember that if a function contains loop instruction i.e. *for, while do, while* ...etc. then that function will not be treated as inline function.

8.6.2 Member Function Outside a Class

In this method of defining a member function outside a class - first you will have to declare a function prototype within the body of the class and then define the function outside the body of the class.

The function defined outside the body of a class have the same syntax as normal functions i.e. they should have a function header and a function body. But, there must have a mechanism of binding the functions to the class to which they belong. This is done by using the **scope resolution operation (: :)**, in the header of the function. The scope resolution operator acts as an 'identity-label' and tells the compiler the class to which the function belongs. The general form of a member function definition is shown below–

```
ClassName
  {   .....
      .....
      ReturnType MemberFunction (arguments); // Function Prototype
      .....
      .....
  } ;
Return Type ClassName : : MemberFunction (arguments)
  {
        // body of the function
    }
```

Here, the label **ClassName : :** tells the compiler that the function **MemberFunction** is the member of class **ClassName**. The scope of the function is restricted to only the objects and other members of the class. We can modify the Program 8.1 by defining the member functions outside the class body, as shown below:

//Program 8.2

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class  employee
{
    private:
      Char name [30];
      float age;
    public:
      void insert_data (char *name1, float age1);
      void show_data ( );// Member Function
};
void employee::insert_data (char *name1, float age1)
{
    strcpy (name, name1);
    age = age1;
}
void employee ::show_data ( )// Function definition
{
    cout <<"Name :"<<name<<endl;
    cout <<"Age:"<<age<<endl;
}
void main ( )
{
    employee e1, e2;
    clrscr( );
    e1 . insert_data ("Hemanga", 30);
    e2 . insert_data ("Prakash", 32);
    cout<<"EMPLOYEE DETAILS...."<<endl;
    e1 . show_data ( );
          e2 . show_data ( );
          getch ( );
      }
```

OUTPUT : EMPLOYEE DETAILS:

Name : Hemanga
Age : 30
Name : Prakash
Age : 32

In the above definitions, the label '***employee : :***' informs the compiler that the functions insert_data ( ) and show_data ( ) are the members of the employee class.

The member functions have some special characteristics. There are–

- A program can have several different classes and they can use the same name for different member functions. The 'membership label' (ClassName : :) resolves the ambiguity of the compiler in deciding which function belong' to which class.
- Member functions can access the private data of the class, whereas non-member functions are not allowed to access. But 'friend function' can access the private data member of a class we will discuss later the friend functions.
- o Member functions of the same class can access all other members of their own class without the use of dot operator.

8.7 INLINE MEMBER FUNCTION

Let us first see what an inline function is. C++ provides a mechanism called *inline function*. When a function is declared as inline, function body is inserted in place of function call during compilation. In this mechanism, passing of control between *caller* and *callee* functions is avoided. The use of the inline function is most effective when calling function is small. If the calling function is very large, in such a case also, compiler copies the content of the inline function in the called function which reduces the program's execution speed. So, in such a case inline function should not be used.

Now, let us see how an inline function behaves with class specification. We have come to know that we can define a member function outside the class definition. The same member function can be defined as inline function by just using the qualifier inline in the header line of the function defining. In the followng, the syntax for defining inline function outside the class declaration is shown

Keyword indicates function defined outside a class body is inline

inline ReturnType ClassName : : FunctionName (arguments)

In fact, the inline mechanism reduces overhead relating to accessing the member function. It provides better efficiency and allows quick execution of functions. An inline member function is treated like a **macro** i.e. any call to this function in a program is replaced by the function itself. This is known as *inline expansion*. Inline functions are also called open subroutines because their code is replaced at the place of function call in the caller function. The normal functions are known as closed subroutines because when such functions are called, the control passes to the function. By default, all member functions defined inside the class are inline function.

We can modify the *Program 8.1* by defining the member functions as inline function as shown below:

// **Program 8.3**

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class employee
{
    private:
      char name [30];
      float age;
    public :
      void insert_data (char *name1, float age1);
      void show_data ( );
```

```
} ;
inline void employee::insert_data(char *name1, float age)
{
     strcpy (name, name1);
     age = age1;
}
inline void employee : : show_data ( )
{
       cout <<"Name :"<<name <<endl;
       cout <<"Age:" <<age <<endl;
}
void main()
{
       employee e1, e2;
       clrscr( );
       e1.insert_data ("Hemanga", 30);
       e2.insert_data ("Prakash", 32);
       cout<< "Employee Details .." << endl;
       e1.show_data ( );
       e2.show _data ( );
       getch ( );
}
```

OUTPUT :

Employee Details–

Name : Hemanga
Age : 30
Name : Prakash
Age : 32

8.8 ARRAY OF OBJECTS

We know that arrays hold data of similar type. Arrays can be of any data type including user defined data type, created by using *struct, class* etc. We can create an array of variables by using class data type. Such an

array of variables of class data type is also known as an array of objects which handle a group of objects.

Let us consider the following class definition :

```
class employee
{
      private :
      char name [30]
      float age ;
      Public :
      void insert_data (char* name1, float age1);
      void show_data ( );
};
```

Here, the identifier '*employee*' is a user defined data type and can be used to create objects that relate to different categories of employees. For example, the following definition will creates an array of objects of '*employee*' class–

```
employee consultant [30];  // array of cosultant
employee clerk [15];       // array of clerk
employee lecturer [20];    // array of lecturer
```

The array consultant contain 30 objects (consultant), namely, consultant [0], consultant [1], ...consultant [29], of type employee class. Similarly, clerk array contains 15 objects and lecturer array contains 20 objects.

We know that as array elements occupy continuous memory locations like the same way as array of objects occupy contiguous memory locations as shown in the fig. 8.4

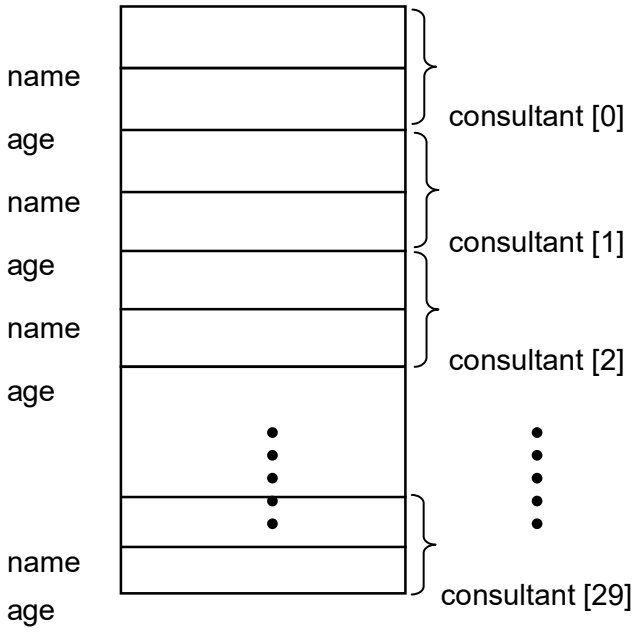


Fig.8.4 Stoage of data items in '*consultant'* array of objects.

By using index an individual element of an array of objects can be referred i.e. ***consultant [15], consultant [9] ..... etc***. By using the dot operator [**.**] we can access any member of an object. For example

consultant [30] . show_data ( )

will display the data of 30th consultant.

We can rewrite the Program 8.1 by using array of objects as shown below:

// Program 8.4

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class employee
{
    private:
    char name [30];
    float age;
    public:
    void insert_data (char *name1, float age)
```

```
        {
              strcpy (name, name1);
              age = age1;
          }
          void show_data ( )
          {
              cout<<"Name:"<<name<<endl;
              cout<<"Age:"<<age<<endl;
          }
};
void main ( )
{
    int i, age, count;
    char name [30], tag;
    employee consultant [30];
    clrscr ( );
    count = 0;
    for (i=0; i<30; i++)
    {
      cout<<"Enter Data For Employee (Y/N):";
      cin >> tag;
      if (tag = = 'y' || tag = = 'Y')
      {
          cout <<"\n Enter Name of Employee:";
          cin >> name;
          cout << "Age:";
          cin >> age;
          consultant[i].insert_data(name, age);
          count ++;
        }
      else
          break;
    }
    cout <<"\n\n Employee Details ......\n" ;
```

```
    for (i=0; i < count; i ++)
    consultant[i].show_data();
          getch ( );
    }
```

OUTPUT :

```
Enter Data for Employee (Y/N) : y
Enter Name of Employee        : Prakash
Age                           : 30

Enter Data for Employee (Y/N) : y
Enter Name of Employee        : Hemanga
Age : 28
Enter Data for Employee (Y/N) : n

Employee Details...
Name                          : Prakash
Age                           : 30
Name                          : Hemanga
Age                           : 28
```

8.9 OBJECTS AS FUNCTION ARGUMENTS

Objects can be passed as an argument to a function. There are three ways of passing objects as function arguments:

- a copy of the entire object is passed to the function, which is also called ***pass-by-value***
- only the address of the object is sent implicitly to the function, which is also called – ***pass by-reference***.
- the address of the object is sent explicitly to the function, which is also called – ***pass-by-pointer***.

8.9.1 Pass-by-value

In this technique, a copy of the object is passed to the called function (callee) from the calling function (caller). Since a copy of the object is passed so any changes made to the object inside the called function do

not affect the object used to call the function

The following program demonstrates the use of objects as function arguments in pass-by-value mechanism.

//Program 8.5

```
#include<iostream.h>
#include<conio.h>
class age
{
    private:
    int birthyr ;
    int presentyr ;
    int year ;
    public:
    void getdata ( );
    void period (age);
};
void age : : getdata ( )
{
    cout<<" \ n Year of Birth:";
    cin >> birthyr ;
    cout << "Current year:" ;
    cin >> presentyr ;
}
void age : : period (age x1)
{
    year = x1 . presentyr - x1 . birthyr ;
    cout << "Your Present Age :" <<year<<"Years" ;
}
void main ( ){
    clrscr ( );
    age a1 ;
    a1 . getdata ( );
    a1 . period (a1) ;
    getch( );
```

```
}
```

OUTPUT:
```
Year of Birth      : 1990
Current Year       : 2002
Your Present Age   : 19 years
```

In the above program, the class age has three data member. The function *getdata ( )* reads integers through keyboard. The function *period ( )* calculates the difference between the two integers. In function main ( ), a1 is an object to the class age. The object a1 calls the function *getdata ( )*. The same object a1 is passed to the function *period ( )*, which calculates the difference between the two integers. Thus, an object can be passed to a function.

8.9.2 Pass-by-Reference

In this technique, only the address of the object is sent to the function. When an address of the object is passed, the address acts as reference pointer to the actual object in the calling funciton. Therefore, any change made to the objects inside the called function will reflect in the actual object in the calling funciton. We can modify the program 8.5 by using the pass by reference mechanism

// **Program 8.6**

```
#include<iostream.h>
#include<conio.h>
class age
  {
      private:
          int birthyr ;
          int presentyr ;
          int year ;
      public:
          void getdata ( );
          void period (age);
```

```
        };
void age : : getdata ( )
{
     cout<<" Year of Birth:";
     cin >> birthyr ;
     cout << "Current year:" ;
     cin >> presentyr ;
}
void age : : period (age & x1)
{
     x1. year = x1 . presentyr - x1 . birthyr ;
     cout << "Your Present Age :" <<year<<"Years" ;
}
void main ( )
{
     clrscr ( );
     age a1 ;
     a1 . getdata ( );
     a1 . period (a1) ;
     getch ( );
}
```

RUN :

Year of Birth	:	1990
Current Year	:	2009
Your Present Age	:	19 years

8.9.3 Pass-by-Pointer

In this mechanism also, the address of the object is passed explicitly to the called function from the calling function. The program 8.6 is modified by using the mechanism pass-by-pointer as follows:

//Program 8.7

```
#include<iostream.h>
#include<conio.h>
class age
{
      private:
          int birthyr ;
          int presentyr ;
          int year ;
      public:
          void getdata ( );
          void period (age * );
};
void age : : getdata ( )
{
      cout<<" \n Year of Birth:";
      cin >> birthyr ;
      cout << "current year:" ;
      cin >> presentyr ;
}
void age : : period (age * x1)
{
    year = x1→presentyr - x1→birthyr ;
    cout << "Your Present Age :" <<year<<"Years" ;
}
void main ( )
{
    clrscr ( );
    age a1 ;
    a1 . getdata ( );
    a1 . period (&a1) ;
    getch ( );
}
```

OUTPUT:
```
Year of Birth      : 1990
Current Year       : 2009
Your Present Age   : 19 years
```

In *Program 8.6* and *Program 8.7* we need to keep an eye on the symbols '.', '→', '*', '&', and the statements(bold lines) where we have appropriately used them.

8.10 FRIEND FUNCITON AND FRIEND CLASS

We have already discussed the fact that the private members of a class cannot be accessible from the outside of the class. Only the member functions of that class have permission for accessing the private members. This policy enforces the encaptulation and data hiding techniques.

Let us think of a situation where a user need a funciton to operate on objects of two different classes. It means that the function will be allowed to access the private data of both the classes. In C++, this situation is over come by using the concept of friend function. It permits a friend function to access the private members of different classes.

The declaration of a friend function must be prefixed by the keyword "**friend**". In the following class is shown a declaration of a friend function.

```
class test
{
        private:
                ---------
                ---------
        public :
                ---------
                ---------
                friend void sum( ) ;
};
```

The function can be defined anywhere in the program similar to any normal C++ function. The function definition does not use either the keyword friend or the scope operator **': :'**. The functions that are declared with the keyword 'friend' are called friend functions. A friend function can be a friend to a multiple classes. The friend function have the following properties :

- There is no scope restriction for the friend function; hence they can be called directly without using objects.
- Unlike member functions of class, friend function cannot access the members directly. On the other hand, it uses object and dot operator to access the private and public member variables of the class.
- Use of friend function is rarely done, because it violates the rule of encapsulation and data hiding.
- The function can be declared in public or private sections without changing its meaning.

The following program demonstrates the use of friend function:

// Program 8.8

```
#include<iostream.h>
#include<conio.h>
class first ;     /*forward declaration like function Prototype*/
class second
{
    int x ;
    public :
    void get value ( )
    {
      cout << "\n Enter a number :" ;
      cin >> x ;
    }
    friend void sum(second, first);//declaration of friend
dunction
} ;
class first
{
```

```
            int y ;
            public :
            void getvalue ( )
            {
                  cout << "\n Enter a number:" ;
                  cin >> y ;
            }
            friend void sum (second, first) ;
} ;
void sum (second m, first n)
{
     cout << "\n Sum of two numbers :" << n.y + m.x
}
void main( )
{
     clrscr ( );
     first a ;
     second b ;
     a.get value ( ) ;
     b.get value ( ) ;
     sum(b,a);//funciton is called like a general function in C++
}
```

OUTPUT:

```
Enter a number      : 9
Enter a number      : 12
Sum of two numbers  : 21
```

In the above program each of the two classes 'first' and 'second' has a member function named getvalue ( ) and one private data member. Notice that, the function sum ( ) is declared as friend function in both the class. Hence, this function has the ability to access the members of both the classes. Using sum ( ) function, addition of integers is calculated and displayed.

It is possible to declare all the member functions of a class as the friend functions of another class. When all the functions need to access another class in such a situation we can declare an entire class as ***friend class***. Always remember that friendship is not exchangeable. Its meaning is that

- declaring *class A* to be a friend of *class B* does not mean that *class B* is also a friend of *class A*. The declaration of a friend class is as follows:

```
class second
  {
       -------
       -------
       friend class first;
  };   /* all member functions of class first are friends to
          class second */
```

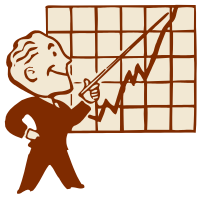
The following program demonstrates the use of friend class :

//Program 8.9

```
#include<iostream.h>
#include<conio.h>
class  smallvalue;
class  value
{
     int a;
     int b;
     public:
     value (int i, int j)   // declaration of constructor with
arguments
     {
       a = i;
       b = j;
     }
 friend class smallvalue;
};
class  smallvalue
{
     public:
     int minimum(value x)
     {
       return x.a < x.b ? x.a : x.b;
     }
```

```
};
void main ( )
{
        value x (15, 25);
        clrscr ( ) ;
        smallvalue y;
        cout << y. minimum(x);
        getch ( );
}
```

In the above program we have used the constructor with arguments. The concept of constructor is illustrated in unit 9 'Constructors and Destructors'.



CHECK YOUR PROGRESS

2. Fill in the blanks of the following :
   (i) Member functions defined within the class definition are implicity ___.
   (ii) When only the address of the object is sent explicity, it is called ____
   (iii) ____ function can access the private data members of a class.
3. State whether the following statements are true or false:
   (a) To reference an object using a pointer to object, uses the<> operator.
   (b) In the prototype void sum (int &) arguments are passed by reference.
   (c) If class A is a friend class of class B then a member function of class B can access the data members of class A.

8.11 STATIC DATA MEMBER AND MEMBER FUNCTON

After studying public and private members, let us study about the static members of a class. Recall what we have learnt from C Programming:

(i) A variable can be declared as static inside a function or outside main().

(ii) Static variables value do not disappear when function is no longer active; their last updated value always persists. That is, when the control comes back to the same function again the static variables have the same value as they leave at the last time.

in C++ also. However, C++ has objects. Hence, the meaning of static with respect to member variables of an object is different.

We have already gained the idea that each object has its separate set of data member variable in memory. The member functions are created only once and all object share the function. No separate copy of the function of each object is created in the memory like data member variables. Figure 8.5 shows the accessing of member function by objects.

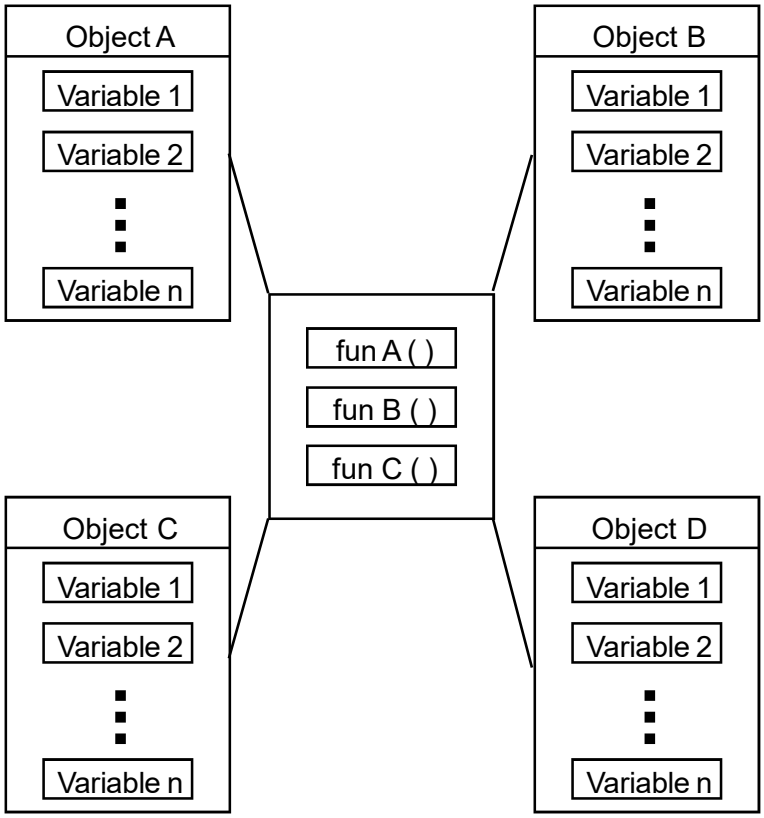


Fig. 8.5 Data members and member functions in memory

In C++, it is possible to create common member variables like function using the **static** keyword. Once a data member variable is declared as static, only one copy of the member is created for the whole class and all objects of the class will share that variable.

Always remember–

- A static variable preserves the value of a variable.
- When a variable is declared as static it is initialized to zero.
- A static data member or member function is only recognized inside the scope of the present class.
- A static variable can be a public or private.

The syntax for declaring static data member or member function within a class is shown below:

static <variable name> ;

static <function name> ;

When you declare a static data member within a class, you are not defining it i.e. you are not allocating storage for it. Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the static variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated.

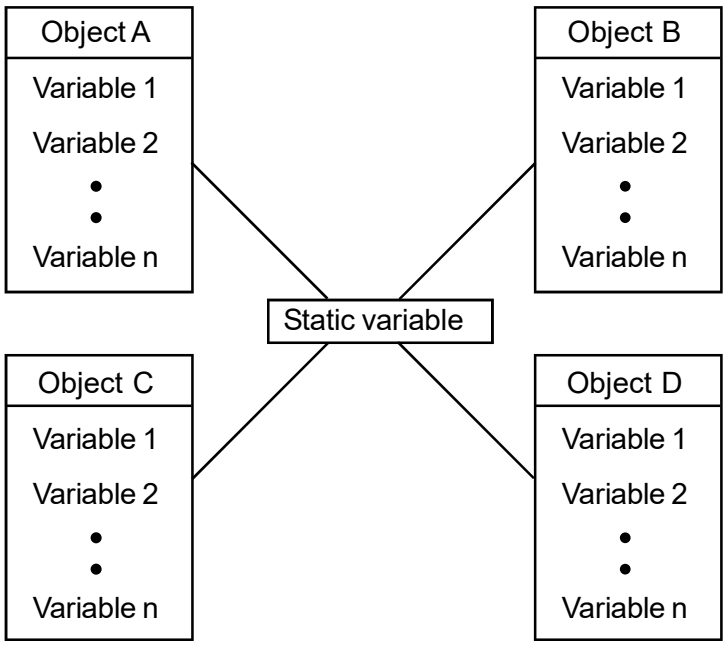


Fig. 8.6 Static member in memory

The declaration of static member is shown below :

```
class number
{
        static int C;
        public:
                ----------
                ----------
};
int number : : C = 0  // initializaiton of static member variable
```

The following program demonstrates the use of static data member in a class–

// Program 8.10

```
#include<stdio.h>
#include<conio.h>
class number
{
        static int C;
        public:
        void count ( )
        {
                C ++;
                cout << "\n C =" << C;
        }
};
int number : : C = 0;
void main( )
 {
        number a, b, c;
        clrscr ( ) ;
        a.count ( );
        b.count ( );
        c.count ( );
        getch ( );
 }
```

OUTPUT: c = 1
c = 2
c = 3

In the above program, the class number has one static data variable C. The *count( )* is a member funciton, increment value of static member variable C by 1 when called. The statement *int number : : C = 0* initiatize the static member with 0. It is possible to initialize the static data members with other values. In the function *main( )*, a, b and c are three objects of *class number*. Each object calls the funciton *count( )*. At each call to the function *count( )* the variable C gets incremented and the count statement displays the value of variable C. The objects a, b and c share the same copy of static data member C.

STATIC MEMBER FUNCTION

In C++, like member variables, funcitons can also be declared as static. When a function is defined as static, it can access only static member variable and functions of the same class. The non-static members are not available to these functions. The static member function declared in public section can be invoked using its class name without using its objects. The static keyword makes the funciton free from the individual object of the class and its scope is global in the class without creating any side effect for other part of the program.

The following points should be remembered while declaring static function:

a) Just one copy of the static member is created in the memory for the entire class. All objects of the class share the same copy of static member.

b) Static member functions can access only static data member, or funcitons.

c) Static member functions can be invoked using class name.

d) It is also possible to invoke static member funcitons using objects.

e) When one of the objects changes the value of data member variables, the effect is visible to all the objects of the class.

The following program demonstrates the use of the static member function in a class.

//Program 8.11

```
#include<iostream.h>
#include<conio.h>
class number
{
    private:
    static int X;
    public:
    static void count ( ) {X++; }
    static void display ( )
    {
          cout << "\n value of X =" << X;
    }
};
int number : : X = 0;
void main ( )
{
clrscr ( ) ;
number::display( );  //invokes display function
number::count( );    //invokes count function
number::count( );    //invokes display function
number::display( );  //invokes display function
getch ( );
}
```

OUTPUT: Value of X : 0
Value of X : 2

In the above program, the member variable X and functions *count ( ) & display ( )* of *class number* are static. The function *count ( )* when called, increases the value of static variable X. The function *display ( )* prints the current value of the variable X. The static functions can be called using class name and scope resolution operator as shown in the program–

number : : count ( );
number : : display ( );



8.12 LET US SUM UP

- Classes are the basic language construct in C++ for creating the user defined data types.
- A class contains member variable or data members and member functions.
- The members of a class are grouped into two sections, namely, private and public.
- Defining variables of a class data type is known as a class instantiation and such variables are called objects.
- Using the member accessed operator, dot(**.**), the class members can be access by the objects.
- The member function can be defined as a) private or public b) inside the class or outside the class.
- The scope resolution operator (**::**) is used, when a member function is defined outside the class body.
- Inline member function is treated like a macro, when a function is declared as inline, function body is inserted in place of function call during compilation.
- We can create an array of variables by using the class data type, then these variables are called array of objects, which occupies contiguous memory locations inmemory.
- There are three methods of passing objects to function, namely, pass-by-value, pass-by-reference, and pass-by-pointer.
- The function that are declared with the keyword ***friend*** are called friend function. A function can be a friend to multiple classes.
- static is the keyword used to preserve value of a variable. When a variable is declared as static, it is initialized to zero. A static function or data element is only recognized inside the scope of the present class.
- When a function is defined as static, it can access only static member variables and functions of the same class. The static member functions are called using its class name without using its objects.



8.13 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education



8.14 ANSWERS TO CHECK YOUR PROGRESS

1. a) i. b) iii. c) iv. d) i. e) ii
2. i) inline, ii) pass-by-pointer, iii) friend
3. a) False, b) True, c) False



8.15 MODEL QUESTIONS

1. What is a class ? How does it accomplish data hiding ?
2. What is an object ? How is an object created ?
3. How is a member function of a class defined or declared ?
4. Explain the use of *private* and *public* keywords. How are they different from each other ?
5. What is the significance of scope resolution operator **::** ?
6. When will you make a function inline and why ?
7. Explain the different methods of passing objects to functions.
8. What is a friend function and a friend class ? Explain with example.

****

UNIT 9 : CONSTRUCTORS AND DESTRUCTORS

UNIT STRUCTURE

9.1 Learning Objectives
9.2 Introduction
9.3 Constructors
    9.3.1 Parameterized Constructors
    9.3.2 Copy Constructors
9.4 Overloading of Contructors
9.5 Destructors
9.6 Dynamic Initialization of Objects
9.7 Let Us Sum Up
9.8 Further Reading
9.9 Answers to Check Your Progress
9.10 Model Questions

9.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- define and use constructors in programming
- learn about default constructor
- learn to use parameterized contructors
- define and use copy constructor
- learn about destructors
- describe the way of declaring a destrcutor
- initialize object dynamically

9.2 INTRODUCTION

In the previous unit we have studied data members and member functions of class. We have seen so far a few examples of classes being implemented. In all cases, we defined a separate member function for reading input values for data members. With the use of object, member

functions are invoked and data members are initialized. These functions cannot be used to initialize the data members at the time of creation of object. C++ provides a pair of special member functions called *constructor* and *destructor*. Constructor enables an object to initialize itself when it is created and destructor destroys the object when it is no longer required.

In this unit we will discuss the constructor and destructor. Different types of contructor and their implementation will also be discussed in this unit.

9.3 CONSTRUCTORS

A constructor is a special member function in a class that is called when a new object of the class is created. It, therefore, provides the opportunity to initialize objects as they are created and to ensure that data members only contain valid values. ***It is a member function whose task is to initialize the object of its class*** and allocate the required resources such as memory. It is distinct from other members of the class because it has the same name as its class.

A constructor can be declared and defined with the following syntax:

```
class classname
{                          Private members
      ..... ;
      public :                    Constructor declaration
            classname( );
};                                Constructor definition
classname :: classname( )
{
            //Body of constructor
}
```

Constructors have some special constraints or rules that must be followed while defining them. These are as follows:

- Constructor is executed automatically whenever the class is instantiated i.e., object of the class is created.
- It always has the same name as the class in which it is defined
- It cannot have any return type, not even *void*.
- It is normally used to initialize the data members of a class.
- It is also used to allocate resources like memory to the dynamic data members of a class.
- It is normally declared in the public access within the class.

For example, let us declare a class with class name 'circle'. If we use constructor, then the constructor name must also be 'circle'. The program can be written as follows:

/***Program 9.1:*** Program to demostrate constructor while calculating the area of a circle*/

```
#include<iostream.h>
#include<conio.h>
#define PI 3.1415
class circle
{
    private:
    float radius;
    public:
    circle();          //constructor declaration
    void area()        // member function
    {
        cout<<"\nArea of the circle is " ;
        cout<<PI*radius*radius<<" sq.units\n";
    }
};
circle :: circle()   //constructor definition
{
    cout<<"\nEnter radius of the circle:  ";
    cin>>radius;
}
```

```
int main()
{
     clrscr();
     circle c;//constructor invoked automatically
     c.area();
     getch();
     return 0;
}
```

The output of the above program will be :

Enter radius of the circle: 5
Area of the circle is 78.5375 sq. units

In the above program, the constructor **circle( )** takes the value of radius from the keyboard. Although the data member is private, the constructor could initialize them. The statement **circle c** declares a variable **c** as object of (type) class **circle**. It also calls the constructor implicitly.

The Default Constructors

The constructor which takes no arguments is called the ***default constructor***. The following code fragment shows the syntax of a default constructor.

```
class class_name
{
     private:
     data members;
     public:
     class_name( );        //default constructor
};
class_name : : class_name( )
{
     /* definition of constructor without any arguments and body */
}
```

If no constructor is defined for a class, then the compiler supplies the default constructor.

Instantiation of Object

Instantiating an object is what allows us to actually use objects in our program. We can write hundreds and hundreds of class declarations, but none of that code will be used until we create an instance of an object. A class declaration is merely a template for what an object should look like. When we instantiate an object, C++ follows the class declaration as if it were a blueprint for how to create an instance of that object.



CHECK YOUR PROGRESS

1. State whether the following statements are True (T) or False (F):
   (i) The constructor is not a member function.
   (ii) It is wrong to specify a return type for a constructor.
   (iii) It is possible to define a class which has no constructor at all.
   (iv) The name of a constructor need not be same as that of the class to which it belongs.
   (v) A class may have two default constructors.
2. Choose the appropriate option:
   (i) A function that is automatically called when an object is created is known as :
   (a) constructor (c) delete
   (b) destructor (d) free( ) function
   (ii) Constructor with no parameter is known as
   (a) stand-alone constructor (c) copy constructor
   (b) default constructor (d) none of these
   (iii) For a class namely *Shape*, the contructor will be like
   (a) void Shape() {.....} (b) shape() {.....}
   (c) Constructor Shape(){.....} (d) Shape() {.....}

3. Write a program in C++ that displays the factorial of a given number using a constructor member function.
4. Write a program in C++ to find the area of a rectangle of length 10.0 and breadth 6.0 by using a class “rectangle”. The program should also contain a constructor along with other member function.

9.3.1 Parameterized Constructors

The constructor that can take arguments are called ***parameterized constructor.*** The arguments can be separated by commas and they can be specified within braces similar to the argument list in function.

When a constructor has been parameterized, the object declaration without parameter may not work. In case of parameterized constructor, we must provide the appropriate arguments to the constructor when an object is declared. This can be done in two ways:

- By calling the constructor ***explicitly***.
- By calling the constructor ***implicitly***.

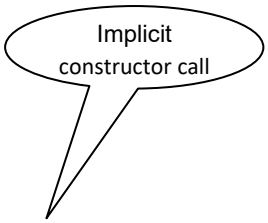
The implicit call method is sometimes known as shorthand method as it is shorter and is easy to implement.

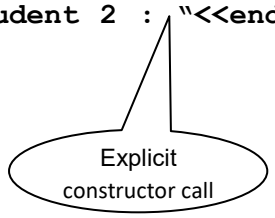
Let us consider the following example to demonstrate how parameterized constructor works.

// **Program 9.2:** Program creating parameterized constructor

```
#include<iostream.h>
#include<conio.h>
class student
{
    private:
    int roll,age, marks;
    public:
    student(int r,int m,int a);//parameterized
constructor
```

```
    void display( )
    {
            cout<<"\nRoll number :" <<roll <<endl;
            cout<<"Total marks : <<marks<<endl;
            cout<<"Age:"<<age<<endl;
        }
};  //end of class declaration
student::student(int r, int m, int a) //constructor
definition
{
    roll = r;
    marks =m;
    age=a;
}
int main( )
{
    student manoj(5,430,16);  //object creation
    cout<<"\nData of student 1 : "<<endl;
    manoj.display();
    student rahul = student(6,380,15);//object
creation
    cout<<"\n\nData of student 2 : "<<endl;
    rahul.display();
    getch();
    return 0;
}
```





The output of the above program will be like this:

```
Data of student 1  :
Roll number        : 5
Total Marks        : 430
Age                : 16
Data of student 2  :
Roll number        : 6
Total Marks        : 380
Age                : 15
```

In the above program, the statement **student(int r, int m, int a,);** is a parameterized constructor with two arguments. In the main( ) function, we see that there are two objects, **manoj** and **rahul** of class type student. The object manoj is initialized with the values 5,430 and16 with the *implicit call* statement **student manoj(5, 430,16);**. The object rahul is initialized with the values 6,380 and 15 with the *explicit call* statement **student rahul = student(6, 380,15);** This statement creates a student object rahul and passes the values 6, 380 and 15 to it.

Constructors with Default Arguments

It is possible to have a constructor with arguments having default values. It means that if we have a parameterized constructor with n parameters, then we can invoke it with less than n parameters specified in the call.

It is useful when most of the objects to be created are likely to have the same value for some data members. We need not specify that in every invocation. For example, suppose we want to record the data of class x (ten) students. For this, we can consider the same class **student** as shown in *Program 8.2.* Most of the students are of age 16. Hence, their birth year is likely to be the same. Only a few of them may have a different year of birth. For this we can use the statement

student(int r, int m, int a=16);

where the third parameter is given the default value. When we write the following statement

student s(1,360);

it assigns the values **1** and **360** to the argument **r** and **m** respectively and **16** to the argument **a** by default. Again, if we write

student t(2,400,15);

it assigns **2**, **400** and **15** to **r**, **m** and **a** respectively. Thus, the actual arguments, when specified, overrides the default values.



EXERCISE

Q. Write a C++ program to create a class "EMPLOYEE" to initialize EMP_ID, DESIGNATION and BASIC_PAY using a constructor and display data for three employees.

9.3.2 Copy Constructors

Object of the same class cannot be passed as argument to constructor to that class by value method.

```
class student
{
     private:
          .......          This is not a valid
     public:               declaration
          student(student  s);
};
```

But it is possible to pass an object of the same class as argument to a constructor by reference method. Such type of constructor, i.e., a constructor having parameter which is a reference to object of the same class is called a ***copy constructor***. Thus, the following declaration is a valid declaration.

```
     class  student
     {
          private:
                                int roll, marks, age;
//data members
          public:
                                student(student  &s);
//copy constructor
     };                              Reference to an object
                                     of class student
```

Copy constructor is also called one argument constructor as it takes only one argument. The main use of copy constructor is to initialize the objects while in creation, also used to copy an object. This constructor allows the programmer to create a new object from an existing one by initialization. Let us demostrate copy constructor with the following program :.

//***Program 9.3:*** Program to demonstrate copy constructor

```
#include<iostream.h>
#include<conio.h>
class  student
{
     private:
          int roll, marks, age;
     public:
          student(int r,int m,int a)
// parameterized constructor
          {
               rol l = r;
               marks = m;
               age = a;
          }
          student(student  &s) //copy constructor
          {
               roll = s.roll;
               marks = s.marks;
               age=s.age;
          }
          void display( )
          {
               cout<<"Roll number :" <<roll <<endl;
               cout<<"Total marks  :"<<marks<<endl;
               cout<<"Age:"<<age<<endl;
          }
};
```

```
int main( )
{
     clrscr();
     student t(3,350,17);// or student k(t);
     student k = t;// invokes copy constructor
     cout<<"\nData of student t:"<<endl;
     t.display();
     cout<<"\n\nData of student k:"<<endl;
     k.display();
     getch();
     return 0;
}
```

The outout of the above program will be like this:

```
Data of student t :
 Roll number :  3
  Total marks :  350
          Age :  17
Data of student k :
 Roll number :  3
  Total marks :  350
          Age :  17
```

The statements

```
student t(3,350,17);      //invokes parameterized constructor
student k = t;            //invokes copy constructor
```

initialize one object **k** with another object **t**. The data members of **t** are copied member by member into **k**. When we see the output of the program we observe that the data of both the objects t and k are same.

9.4 OVERLOADING OF CONTRUCTORS

A class may contain multiple constructors i.e., a class can have more than one constructor with the same name. The constructors are then recognized depending on the arguments passed. It may differ in terms of arguments, or data types of their arguments, or both. This is called ***overloading of constructors*** or ***constructor overloading***.

Program 9.3 is also an example of constructor overloading as it has two constructors: one is parameterized and the other is a copy constructor. Let us take a suitable example to demostrate overloading of constructors.

//**Program 9.4:** Demonstration of constructor overloading

```
#include<iostream.h>
#include<math.h>
#include<conio.h>
class complex
{
    private:
    float real,imag;
    public:
    complex( )//constructor with no argument
    {
        real = imag = 0.0;
    }
    complex(float r, float i)
    //constructor with two arguments
    {
        real = r;
        imag = i;
    }
    complex(complex &c)   //copy constructor
    {
        real = c.real;
        imag = c.imag;
```

```
    }
    complex addition(complex d);
  /*member function returning object and taking ob-
                ject as argument */
    void display( )  //Display member function
    {
        cout<<real;
        if(imag < 0)
            cout<<"-i";
        else
            cout<<"+i";
        cout<<fabs(imag);/*fabs calcute the absolute
value
                                of a  floating point
number */
        }
    };
    complex complex::addition(complex d)
    {
        complex temp;   //temporary object of type
complex class
        temp.real=real+d.real;  //real parts added
        temp.imag=imag+d.imag;   //imaginary parts
added
        return(temp);
    }
int main( )
{
    clrscr( );
    complex x1,x4;  //invokes default constructor
    cout<<"\nThe complex numbers in a+ib form :\n\n";
    cout<<"First complex number: ";
    x1.display();
    complex x2(1.5,5.3);//invokes parameterized construc-
tor
```

```
    cout<<"\nSecond complex number: ";
    x2.display();
    complex x3(2.4,1.9);
    cout<<"\nThird complex number: ";
    x3.display();
    cout<<"\nAddition of 2nd and 3rd complex number: ";
    x4=x2.addition(x3);   //function call
    x4.display( );
    cout<<"\nThe result is copied to another object:
";
        complex x5(x4); //invokes copy constructor
        x5.display();
        getch();
        return 0;
    }
```

The output of the above program will be:

The complex numbers in a+ib form:

First complex number: 0+i0

Second complex number: 1.5+i5.3

Third complesx number: 2.4+i1.9

Addition of second and third complex number: 3.9+i7.2

The result is copied to another object: 3.9+i7.2

We have the following three constructors

```
        complex( );
        complex(float r, float i) ;
    and complex(complex &c);
```

The above program indicates overloading of constructors. These constructors are invoked during the creation of an object depending on the number and types of arguments passed. The default constructor **complex( );** initializes the data members *real* and *imag* to 0.0. In main( ), the statement **complex x2 (1.5,5.3);** passes two parameters to the constructor explicitly with the help of the parameterized constructor **complex (float r, float i);** With the help of copy constructor **complex (complex &c);** data members of one object is copied member by member into another.

9.5 DESTRUCTORS

Like constructors, destructors are also special member functions used in C++ programming language. Destructors have the opposite function of a constructor. The main function of destructors is to free memory and to release resources. Destructors take the same name as that of the class name preceded by a ***tilde ~***. A destructor takes no arguments and has no return type.The general syntax of a destructor is as follows:

```
class classname
{
     private :            //Private members
          ........ ;
     public :
          ~classname( );       //Destructor declaration
};
       classname :: ~classname( )//Destructor definition
{
          //Body of destructor
}
```

In the above, the symbol tilde ~ represents a destructor which precedes the name of the class. Like the default constructor, the compiler always creates a default destructor if we donot create one. Similar to constructors, a destructor must be declared in the public section of a class. Destructor cannot be *overloaded* i.e., a class cannot have more than one destructor.

/* **Program 9.5 :** Program demonstrating destructor */

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class student
{
     private:
```

```
        int age;
        char name[25];
    public:
        student(int a, char n[25])
        {
            age=a;
            strcpy(name,n);
        }
        void show()
        {
            cout<<"\nThe name of the student is:
"<<name;
            cout<<"\nHis age is: "<<age;
        }
        ~student() //destructor defined
        {
            cout<<"\nObject Destroyed";
            }
    };
    int main()
    {
        student s1(21,"Rahul");
        student s2(23,"Dipankar");
        clrscr();
        s1.show();
        s2.show();
        return 0;
    }
```

To see the execution of destructor, we have to press Alt+F5 after compiling and running the program. The ouput will be like this:

The name of the student is : Rahul
His age is : 21

The name of the student is : Dipankar
His age is : 23

Object Destroyed
Object Destroyed

The following points should be kept in mind while defining and writing the syntax for the destructor

- A destructor must be declared with the same name as that of the class to which it belongs. But destructor name should be preceeded by a tilde (~).
- A destructor should be declared with no return type.
- A destructor must have public access in the class declaration.

9.6 DYNAMIC INITIALIZATION OF OBJECTS

We can dynamically initialize objects through constructors. Object's data members can be initialized dynamically at run time even after their creation.

//Program 9.6 Demonstration of dynamic initialization of object

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
class number
{
    public:
    int num;
    number(int n)
    {
        num=n;
    }
    int sum( )
    {
        num=num+5;
        return(num);
```

```
    }
};
int main( )
{
    number obj1(1);  // parameterized constructor invoked
    number  obj2(2);
    clrscr();
    cout<<"\nValue of object 1 and object 2 are : ";
    cout<<obj1.num<<"\t"<<obj2.num;
    number  obj3(obj1.sum( ));//dynamic initialization of
object
    cout<<"\nValue of object 1 after calling sum() :
"<<obj1.num;
    cout<<"\nValue of object 3 is :"<<obj3.num;
    getch();
    return 0;
}
```

The output of the above program will be :

Value of object 1 and object 2 are	:	1 2
Value of object 1 after calling sum()	:	6
Value of object 3 is	:	6

The statements **number obj1(1);** and **number obj2(2);** will initialize obj1 and obj2 by 1 and 2 respectively by invoking the constructor number(int n). **obj1.sum( )** will return the value 6. This is passed as argument in the statement

```
number  obj3(obj1.sum( ));
```

where obj3 is initialized dynamically with the value returned by **obj1.sum( )**.



CHECK YOUR PROGRESS

5. State whether the following statements are true(T) or false(F) :
   (i) A class may have more than one destructor.
   (ii) When an object is destroyed, destructor is automatically called.
   (iii) Presence of many destructor in a class is called destructor overloading.
   (iv) A destructor can have a return type.
   (v) A destructor can have arguments like destructor.
   (vi) We can pass arguments to a constructor.
   (vii) Destructors cannot take arguments.
   (viii) Destructors can be overloaded.
6. Choose the appropriate option:
   (i) Return type of a destructor is
   (a) int (b) tilde
   (c) void (d) nothing, destructor has no return value
   (ii) A class may have _____ destructor
   (a) two (b) only one
   (c) many (d) none of these
7. Distinguish between the following two statements:

```
time t1(14, 10, 30);          //statement1
time t1 = time(14, 10, 30);   //statement2
```



9.7 LET US SUM UP

Based on the discussion so far, the key points to be kept in mind from this unit are:

- A constructor is a special member function for automatic initialiation of an object. Whenever an object is created, the constructor will be called.

- Constructor should have the same name as that of class name to which it belong.
- Constructor has no return type.
- The constructor without argument is called a default constructor.
- Constructors may be overloaded to provide different ways of initializing an object.
- We may have more than one constructor with the same name, provided each has a different signature or arguments list.
- A constructor can accept a reference to its own class as a parameter. Such a constructor having a reference to an instance of its own class as an argument is known as copy constructor.
- C++ also provides another member function called destructor that releases memory by destroying objects when they are no longer required.
- A destructor never takes any argument nor does it return any value.



9.8 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education



9.9 ANSWERS TO CHECK YOUR PROGRESS

1. (i) False (ii) True (iii) True (iv) False (v) False
2. (i) (a) constructor (ii) (b) default constructor (iii) (d) Shape() {.....}
3. // Program for finding factorial of a number

```
#include<iostream.h>
#include<conio.h>
class factorial
{
```

```
        private:
            long n;
        public:
        factorial();
    };
    factorial::factorial()
    {
        cout<<"\nEnter the number to find factorial:
";
        cin>>n;
        long fact =1;
        while(n>1)
        {
            fact =fact *n;
            n=n-1;
        }
        cout<<"\n\nThe factorial is : "<<fact;
    }
    int main()
    {
        factorial f;
        getch();
        return 0;
    }
```

4. /*Program for finding the area of a rectangle of length 10.0 and breadth 6.0*/

```
#include<iostream.h>
#include<conio.h>
class rectangle
{
    private:
        float length,breadth;
    public:
        rectangle( )
        {
```

```
            length = 10.0;
            breadth=6.0;
        }
        float area( )
        {
            return( length*breadth);
        }
};
void main( )
{
    rectangle r;
    clrscr();
    cout<<"\n The area of the rectangle is:
"<<r.area()
        <<" square units";
    getch( );
    return 0;
}
```

5. (i) False (ii)True (iii) False (iv) False
   (v) False (vi) False (vii) True (viii) False
6. (i) (d) nothing, destructor has no return value,
   (ii) (b) only one
7. The first statement creates the object *t1* by calling the *time* constructor implicitly. On the other hand, the second statement creates the object *t1* by calling the *time* constructor explicitly.



9.10 MODEL QUESTIONS

1. What are constructor and destructors? Expalin how they differ from normal member functions.
2. What are the characteristics of a constructor?
3. Write short notes on (i) Default Constructor
   (ii) Copy Constructor

4. Differentiate between constructor and destructors.
5. Give the rules governing the declaration of a constructor and destructor.
6. What is a parameterized constructor ?
7. Write a C++ program to show overloading of constructors?
8. Define a class ‘complex_no’ which has two data members, one for representing the real part and the other for complex part. Define a constructor to initialize the object.
9. Define suitable constructor(s) for the STRING class defined below;

```
STRING
{
        int length;
        char s[50];
    public:
        //define suitable constructors
};
```

****

UNIT 10 : OPERATOR OVERLOADING

UNIT STRUCTURE

10.1 Learning objectives
10.2 Introduction
10.3 Basic Concept of Overloading
10.4 *operator* Keyword
10.5 Overloading Unary Operators
10.6 Operator Return Type
10.7 Overloading Binary Operators
10.8 Strings and Operator Overloading
10.9 Type Conversion
10.10 Let Us Sum Up
10.11 Further Readings
10.12 Answers to Check Your Progress
10.13 Model Questions

10.1 LEARNING OBJECTIVES

After going through this unit, you will able to :

- describe the fundamental concept of overloading
- describe the use of the keyword *operator*
- illustrate the overloading of unary and binary operators
- describe manipulation of strings using operators

10.2 INTRODUCTION

So far, we have discussed the concept of class and objects and how to allocate required resource such as memory and initialize the objects of classes using constructors and how to deallocate the memories using destructors. C++ offers an another important feature namely *operator overloading*, through which operators like +,-,<=,>= etc. can be used with user defined data types, with some additional meaning.

In this unit, we will concentrates on the dicussion of overloading of operators (unary and binary) as well as the string manipulations using operators.

10.3 BASIC CONCEPT OF OVERLOADING

We know that operators (+,-,<=,>= etc.) are used to perform operation with the constants and variables. Without the use of the operators a programmer cannot write or built an expression. We have already used these operators with the basic data types such as *int* or *float* etc. The operator +(plus) can be used to perform addition of two variables but we cannot apply the + operator for addition of two objects. If we want to add two objects using the + operator then the compiler will show an error message. To avoid this error message you must have to make the compiler aware about the addition process of two objects. To perform operation with objects you need to redefine the definition of various operators. For example, for addition of objects X and Y, we need to define operator +(plus). Re-defining an operator does not change its original meaning. It can be used for both variables of built-in data types as well as objects of user defined data types.

Operator overloading in C++, permits to provide additional meaning to the operators such as +, *, >=, –, = etc., when they are applied to user defined data types. Hence, the operator overloading is one of the most valuable concepts introduced by C++ language. It is a type of polymorphism. We will discuss polymorphism in a later unit. C++ allows the following list of operators for overloading.

Table 10. 1:C++ Overlodable Operators

Operator Category	Operators
Arithmetic	+, -, *, /, %
Bit-Wise	&, |, ~, ^
Logical	&&, ||, !
Relational	<, >, ==, !=, <=, >=
Assignment	=
Arithmetic assignment	+=, -=, *=, /=, %=, &=, |=, ^=
Shift	>>, <<, >>=, <<=
Unary	++, --
Subscripting	[ ]
Function call	( )
Dereferencing	->
Unary sign prefix	+, -
Allocate and free	new, delete

10.4 *operator* KEYWORD

The keyword *operator* helps in overloading of the C++ operators. The general format of operator overloading is shown below :

```
ReturnType opeator OperatorSymbol ([arg1], [arg2])
{
     // body of the function
}
```

Here, the keyword **operator** indicates that the *OperatorSymbol* is the name of the operator to be overloaded. The operator overloaded in a class is known as *overloaded operator function.*

The following statements shows the use of the *operator* keywords.

```
class Index
{
     // class data and member function
     Index operator ++( )
     {
            index temp;
```

```
            value = value+1;
            temp.value = value;
            return temp;
        }
};
```

Here, return type of the operator function is the name of a class within which it is declared. It can be defined as follows :

```
class Index
  {
      // class data and member function
         Index operator ++( );
};

Index Index :: operator ++( )
{
    index temp;
    value = value+1;
    temp.value = value;
    return temp;
}
```

The operator function should be either a member function or a friend function. When the operator function is declared as member function and takes no argument, it is known as *unary operator overloading* and when it takes one argument it is known as *binary operator overloading*.

10.5 OVERLOADING UNARY OPERATORS

When an operator function takes no argument, it is called as unary operator overloading. You are already familiar with the operators ++, --, and -, *which have only single operands* are called unary operators. The unary operators ++ and -- can be used as **prefix** or **suffix** with the functions. The following program demonstrates the overloading of unary '--' operators.

// **Program 10.1**

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

class unary
{
     int x, y, z;

     public :
     unary (int i, int j, int k) // parameterized construc-
tor
     {
          x=i; y=j; z=k;
     }
     void display();//displays contents of member variables
     void operator --();//overloads the unary operator --
};

void unary :: operator --()
{
     --x; --y; --z;// values of variables will decrease by 1
}
void unary :: display()
{
       cout<<"X="<<x<<"\n";
       cout<<"Y="<<y<<"\n";
       cout<<"Z="<<z<<"\n";
}
void main()
{
     clrscr();
     unary A(31, 41, 51);
     cout<<"\n Before Decrement of A :\n";
     A.display();
     --A;// calls the function operator --()
     cout<<"\n After Decrement of A :\n";
     A.display();
     getch();
}
```

OUTPUT : Before Decrement of A : X=31
Y=41
Z=51
After Decrement of A : X=30
Y=40
Z=50

10.6 OPERATOR RETURN TYPE

In the above example, we have declared the operator function of type *void* i.e. it will not return any value. But it is possible to return value and assign it to other object of same type. The return value of the operator is always of the class type, it means that class name will be in the place of the return type specification because we are applying the operator overloading properties only for the objects. Always remember that an operator cannot be overloaded for basic data types, so the return value of operator function will be of class type. The following program demonstrates the operator return types :

// **Program 10.2**

```
#include<iostream.h>
#include<conio.h>
class unary
{
     int x;
     public :
     unary () { x=0; }
     int getx() // returns the current value of variable x
     { return x; }
     unary operator ++();
};
unary unary :: operator ++()
{
     unary temp;
```

```
    x=x+1;
      temp.x=x;
      return temp;
}
void main()
{
    clrscr();
    unary A1,A2;
    cout<<"\n A1="<<A1.getx();
    cout<<"\n A2="<<A2.getx();
    A1=A2++;//first increment the value of A2 and assigns it to
A1
    cout<<"\n A1="<<A1.getx();
    cout<<"\n A2="<<A2.getx();
    A1++; // object A1 is incresed
    cout<<"\n A1="<<A1.getx();
    cout<<"\n A2="<<A2.getx();
    getch();
}
```

RUN : A1 = 0
A2 = 0
A1 = 1
A2 = 1
A1 = 2
A2 = 1

10.7 OVERLOADING BINARY OPERATORS

Binary operators are overloaded by using member functions and friend functions. The difference is in the number of arguments used by the function. In the case of binary operator overloading, when the function is a member function then the number of arguments used by the operator member function is one. When the function defined for the binary operator overloading is a friend function, then it uses two arguments. Here, we will dis-

cuss the overloading of binary operator when the operator function is a member function.

Binary operator overloading, as in unary operator overloading, is performed using a keyword *operator*. The following program demonstrates the overloading of binary operators.

// Program 10.3

```
#include <iostream.h>
#include<conio.h>
class Binary
{
    private:
      int x;
      int y;
    public:
      Binary()  //Constructor
      { x=0; y=0; }
    void getvalue() //Member Function for Inputting values
    {
      cout <<"\n Enter value for x:";
      cin >> x;
      cout << "\n Enter value for y:";
      cin>> y;
    }
    void displayvalue( )//Member Function for Outputting Values
    {
      cout<<"\n\nThe resultant value : \n";
      cout <<" x =" << x <<"; y ="<<y;
    }
    Binary operator +(Binary);//Binary is class name
};
Binary Binary :: operator +(Binary e2)
//Binary operator overloading for + operator defined
{
    Binary temp;
    temp.x = x+ e2.x;
```

```
    temp.y = y+e2.y;
    return (temp);
}
void main( )
{
    Binary e1,e2,e3;  //Objects e1, e2, e3 created
    clrscr();
    cout<<"\nEnter value for Object e1:";
    e1.getvalue( );
    cout<<"\nEnter value for Object e2:";
    e2.getvalue( );
    e3= e1+ e2;  //Binary Overloaded operator used
    e3.displayvalue( );
    getch();
}
```

OUTPUT :

```
Enter value for Object e1  :
        Enter value for x  :  10
        Enter value for y  :  20
Enter value for Object e2  :
        Enter value for x  :  30
        Enter value for y  :  40
  The Resultant Value      :  x = 40; y = 60
```

In the above example, the class Binary has created three objects e1, e2, e3. The values are entered for objects e1 and e2. The binary operator overloading for the operator '+' is declared as a member function inside the class Binary. The definition is performed outside the class Binary by using the scope resolution operator and the keyword *operator*.

The important aspect is that the following two statements are equivalent.

```
e3= e1 + e2; // e3= e1.operator +(e2)
```

The binary overloaded operator '+' is used. When the compiler encounters such expressions, it examines the argument type of the operator. In this statement, the argument on the left side of the operator '+', e1, is the object

of the class Binary in which the binary overloaded operator '+' is a member function. The right side of the operator '+' is e2. This is passed as an argument to the operator '+' i.e. the expression means **e3 = e1.operator +(e2).** The operator returns a value (binary object temp in this case), which can be assigned to another object (e3 in this case).

Since the object e2 is passed as argument to the operator'+' inside the function defined for binary operator overloading, the values are accessed as e2.x and e2.y. This is added with e1.x and e1.y, which are accessed directly as x and y.

Always remember that, in the overloading of binary operators, the left-hand operand is used to invoke the operator function and the right-hand operand is passed as an argument to the operator function.

10.8 STRINGS AND OPERATOR OVERLOADING

We are already familiar with the *strcat()* function which is used for concatenation of strings. Consider the following two strings

```
char str1[50]   = "Bachelor of Computer";
char str2[20]   = "Application";
```

The string str1 and str2 are combined, and the result is stored in str1 by invoking the function *strcat()* as follows :

```
strcat(str1, str2);
```

The same operation can be done by defining a string class and overloading the + operator. The following program demonstrates the concatenation of two string using the overloading concept.

// Program 10.4

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

class String
{
```

```
    private:
    char str[100];
    public:
    String()    //Constructor
    { strcpy(str," "); }
    String(char *msg)   //Constructor
    { strcpy(str, msg); }
    void display() //Member Function for Display strings
    {
        cout <<str;
    }
    String operator +(String s);
};
String String :: operator +(String s)
//Binary operator overloading for + operator defined
{
        String temp = str;
        strcat(temp.str, s.str);
        return temp;
}
void main( )
{
    clrscr();
    String str1 = "Bachelor of Computer";
    String str2 = "Application";
    String str3;
    str3= str1+str2;
    cout<<"\n str1 =";
    str1.display();
    cout<<"\n str2 =";
    str2.display();
    cout<<"\n The String after str3=str1+str2 \n \n";
    str3.display();
    getch();
    }
```

In this program, the concatenation is performed by creating a temporary string object *temp* and initializing it with the first string. The second string is added to first string in the object temp using the *strcat()* and finally the resultant temporary string object temp is returned. Here, in the program, the length of str1+str2 should not exceed the array size 100 (i.e. char str[100]).

Thus, we have seen that, in C++ programming language, operator overloading adds new functionality to its existing operators. The programmer must add proper comments concerning the new functionality of the overloaded operator. The program will be efficient and readable only if operator overloading is used only when necessary. The operators that cannot be overloaded are - **.**, **?:**, **sizeof**, **::**, **.*** , #, ##.

10.9 TYPE CONVERSION

We cannot convert between user-defined data types(classes) just as we can convert between basic types. This is beacuse the compiler does not know anything about the user-defined type.

Now, let us look into how C++ handles conversions for its **built-in types (int, float, char, double etc.)**. When you make a statement assigning a value of one standard type to a variable of another standard type, C++ automatically will convert the value to the same type as the receiving variable, provided the two types are compatible.

For example, the following statements all generate numeric type conversions:

```
long count = 8;     // int value 8 converted to type long
double time = 11;   // int value 11 converted to type double
int side = 3.33;    // double value 3.33 converted to type int 3
```

These assignments work because C++ recognizes that the diverse numeric types all represent the same basic thing, a number, and because C++ incorporates built-in rules for making the conversions. However, you can lose some precision in these conversions. For example, assigning

3.33 to the int variable results in only getting the value 3, losing the 0.33 part.

The C++ language does not automatically convert types that are not compatible.

For example, the statement

```
int * p = 10;  // type clash
```

fails because the left-hand side is a pointer-type, whereas the right-hand side is a number. And even though a computer may represent an address internally with an integer, integers and pointers conceptually are quite different. For example, you wouldn't square a pointer. However, when automatic conversions fail, you may use a type cast:

```
int * p = (int *) 10;  // ok, p and (int *) 10 both pointers
```

This sets a pointer to the address 10 by type casting 10 to type pointer-to-int (that is, type int *).

Now, let us look into how C++ handles conversions from basic type to user-defined types vice-versa.

Basic type to user defined type :

This type of conversion can be easily carried out. It is automatically done by the compiler with the help of in-built routines or by type casting. In this type the left hand operand of = sign is always class type or user defined type and the right hand side operand is always basic type. The following program explains this type of conversion.

```
//Program 10.5
#include <iostream.h>
#include<conio.h>
class Test
{
  private:
```

```
        int x;
        float y;
        public:
        Test()   //Constructor
        { x=0; y=0; }
        Test(float z)  //Constructor with one argument
        { x=2; y=z; }
        void display()  // Function for displaying values
        {
             cout <<"\n  x =" << x <<" y ="<<y;
             cout <<"\n  x =" << x <<" y ="<<y;
        }
};
void main( )
{
        Test a;
        clrscr();
        a=9;
        a.display();
        a=9.5;
        a.display();
        getch();
}
```

OUTPUT : x=2 y=9
x=2 y=9
x=2 y=9.5
x=2 y=9.5

In the above program, the class Test has two data member of type integer and float. It also has two constructors one with no arguments and the second with one argument. In *main()* function, **a** is an object of class Test. When a is created the constructor with no argument is called and data memberms are initialize to zero. When a is initialized to 9 the constructor with float argument i.e. ***Test(float z)*** is invoked. The integer value is converted to float type and assigned to data member y. Again, when a is

assigned to 9.5, same process repeated. Thus, the conversion from basic to class type is carried out.

User defined type to basic type :

As we know, the compiler does not have any prior information about user defined data type using class, so in this type of conversion it needs to inform the compiler how to perform conversion from class to basic type. For this purpose, a conversion function should be defined in the class in the form of the operator function. The operator function is defined as an overloaded basic data type which takes no arguments. The syntax of such a conversion function is shown below-

```
operator Basic type( )
    {
        // steps for converting
    }
```

In the above syntax, you have noticed that, the conversion function has no return type. While declaring the operator function the following condition should always remember :

i) the operator function should not have any argument.
ii) it has no any return type.
iii) it should be a class member.

The following program demonstrates this conversion mechanism :

// Program 10.6

```
#include <iostream.h>
#include<conio.h>
class Time
{
    private:
    int hour;
      int minute;
      public:
```

```
        Time(int a)
        {
             hour=a/60;
             minute=a%60;
        }
        operator int()
        {
             int a;
             a=hour*60+minute;
             return a;
        }
};
void main( )
{
     clrscr();
     Time t1(500);
     int i=t1; //  operator int() is invoked
     cout<<"\n"<<"The value of i:"<<i;
     getch();
}
```

OUTPUT : The value of i : 500

In the above program, the statement ***int i=t1***, invokes the operator function which finally converts a time object to corresponding magnitude (of type int).



CHECK YOUR PROGRESS

1. Choose the correct answer from the following :
   a) Operator overloading is
      i) making C++ operators work with objects
      ii) giving C++ operators more than they can handle
      iii) giving new meaning to existing C++ operators
      iv) making new C++ operators

b) To convert from a user-defined class to basic type, you would use
   i) a built-in conversion function
   ii) a one argument constructor
   iii) an overloaded = operator
   iv) a conversion function that is a member of the class

c) To convert from a basic type to user-defined class, you would use
   i) a built-in conversion function
   ii) a one argument constructor
   iii) an overloaded = operator
   iv) a conversion function that is a member of the class

d) _____operator must have one class object.
   i) + ii) new
   iii) all iv) none of these

e) Binary overload operators are passed _____ arguments.
   i) one ii) two
   iii) no iv) none of the above

2. Fill in the blanks :
   i) The statement x=y will cause _____ if the objects are of different classes.
   ii) _____ is making operators to woork with user defined data types.
   iii) Single argument constructor is usually defined in the _____ class.
   iv) _____ function must not have a return type.
   v) _____ are operators that act on only one operand.



10.10 LET US SUM UP

- Operator overloading is one of the important concepts in C++ which allows to provide additional meaning to operators +, -, >=, <= etc. when they are applied to user defined data types.

- Overloaded operators are redefined within a class using the keyword ***operator*** followed by an operator symbol. When an operator is overloaded, the produced symbol is called the operator function name.
- Overloading of operator cannnot change the basic meaning of an operator. When an operator is overloaded, its prooperties like systax, precedence and associativity remain constant.
- Operators ++, --, and -, *which have only single operands* are called unary operators. The unary operators ++ and -- can be used as **prefix** or **suffix** with the functions.
- The binary operators require two operand. Binary operators are overloaded by using member functions and friend functions.
- The operators which cannot be overloaded are - **.**, **?:**, **sizeof**, **::**, **.*** , #, ##.
- The concept of operator overloading can also be applied to data conversion. C++ offers automatic conversion of primitive data types.
- Actually there are three possibilities of data conversion :
  a) Basic type to user defined type(class type)
  b) User defined type(class type) to basic type
  c) Class type to another class type (we have not discussed here)



10.11 FURTHER READINGS

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education

10.12 ANSWERS TO CHECK YOUR PROGRESS

1. a) iii, b) iv, c) iii,
   d) ii, e) i.

2. i) Compiler error,
   ii) operator overloading,
   iii) destination,
   iv) casting operator,
   v) unary operator

10.13 MODEL QUESTIONS

1. What is operator overloading? Give the advantage of operator overloading.
2. What is operator function? Describe operator function with syntax and examples.
3. What is the difference between overloading of binary operators and of unary operators?
4. Explain the conversion from basic type to user defined type(class type) with examples.
5. Explain the conversion from user defined type(class type) to basic type with examples.
6. Write a program to overload the -- operator.
7. Write a program to overload the binary operator + in order to perform addition of complex numbers.
8. Write a program to overload the relational operator (>, <, ==) in order to perform the comparision of two strings.

****

UNIT 11 : INHERITANCE

UNIT STRUCTURE

11.1 Learning Objectives
11.2 Introduction
11.3 Inheritance
  11.3.1 Defining a Derived Class
  11.3.2 Accessing Base Class Members
11.4 Types of Inheritance
  11.4.1 Single Inheritance
  11.4.2 Multiple Inheritance
  11.4.3 Hierarchical Inheritance
  11.4.4 Multilevel Inheritance
  11.4.5 Hybrid Inheritance
  11.4.6 Multipath Inheritance
11.5 Virtual Base Classes
11.6 Abstract Classes
11.7 Let Us Sum Up
11.8 Further Reading
11.9 Answers to Check Your Progress
11.10 Model Questions

11.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- describe the concept of inheritance in C++
- create new classes by reusing the members and properties of existing classes
- describe the advantages and disadvantages of inheritance in pro gramming
- describe how to use base class access specifier *public*, *private* and *protected*
- describe about the use of virtual base class and abstract class

11.2 INTRODUCTION

In this unit, we shall discuss one important and useful feature of Object - Oriented Programming (OOP) which is called ***inheritance***. C++ supports the concept of inheritance. We have already discussed the concept of classes and objects in our previous unit. The knowledge of class and objects is a prerequisite for this unit.

With the help of inheritance we can reuse (or inherit) the property of a previously written class in a new class. There are different types of inheritance which will be discussed in this unit. The concept of abstract and virtual base class will also be covered in this unit.

11.3 INHERITANCE

In Biology, *inheritance* is a term which represents the transformation of the hereditary characters from parents or ancestors to their descendent. In the context of Object-Oriented Programming, the meaning is almost same. The process of creating a new class from existing classes is called ***inheritance***. The newly created class is called ***derived class*** and the existing class is called the ***base class***. The derived class inherits some or all of the characteristics of the base class. Derived class may also possess some other characteristics which are not in the base class.

For example, let us consider two classes namely, “employee ” and “manager”. Whatever information is present in “employee” class, the same will be present in “manager’ also. Apart from that, there will be some extra information in “manager” class due to other responsibilities assinged to managers. Due to the facility of inheritance in C++, it is enough only to indicate those pieces of information which are specific to manager in its class. In addition, the “manager” class will inherit the information of “employee” class.

Before discussing the different types of Inheritance and their implementation in C++, we will first denote the advantages and disadvantages of inheritance.

Advantages of Inheritance

- ***Reusability***

  Reusability is an important feature of Object Oriented Programming. We can reuse code in many situations with the help of inheritance. The base class is defined and once it is compiled, it need not be rewritten. Using the concept of inheritance the programmer can create as many derived classes from the base class as needed. New features can also be added to each derived class when required.

- ***Reliability and Cost***

  Reusability would not only save time but also increase reliability and decrease maintenance cost.

- ***Saves Time and Effort***

  The reuse of class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

Disadvantages of Inheritance

- Inappropriate use of inheritance makes a program more complicated.
- In the class hierarchy various data elements remain unused, the memory allocated to them is not utilized.

11.3.1 Defining a Derived Class

A derived class is defined by specifying its relationship with the base class in addition to its own details. The general form of deriving a new class from an existing class is as follows:

```
class DerivedClassName : access specifier  BaseClassName
  {
        .....
        .....    //members of derived class
        .....
  };
```

The derived class name and the base class name are separated by a colon "**:**". We can derive classes using any of the three base class access specifiers: ***public***, ***private*** or ***protected***. If we do not specify the access specifier, then by default it will be *private.* Generally, it is convenient to specify the access specifier while deriving a class. Access specifiers are sometime called *visibility mode*. It determines the access control of the base class members inside the derived class. In the previous unit, we have already learnt about *private* and *public* access specifier while declaring classes. ***Protected*** specifier has a significant role in inheritance.

11.3.2 Accessing Base Class Members

There may be three types of base class derivation:

- Public derivation
- Private derivation
- Protected derivation

- **Public derivation**

When the base class is inherited by using ***public access specifier*** then all public members of the base class become public members of the derived class, and all protected members of the base class become protected members of the derived class. The private members of the base class remain private and are not accessible by members of derived class. Let us examine this with the following example:

// **Program 11.1:** Base class with public access specifier

```
#include<iostream.h>
#include<conio.h>
class base
{
  private:
       int num1,num2;
```

```
    public:
        void input(int n1, int n2 )
                    {
                          num1=n1;
                          num2=n2;
                    }
        void display( )
        {
             cout<<"Number 1 is:   "<<num1<<endl;
             cout<<"Number 2 is: "<<num2;
        }
    };   //end of base class

class derived : public base
    {
        private:
             int num3;
        public:
             void enter(int n3)
             {
                  num3=n3;
             }
             void show()
             {
                  cout<<"\nNumber 3 is: "<<num3;
             }
    };  //end of derived class
    int main()
    {
    derived d;     // d is an object of derived class
    clrscr();
    d.enter(15); //enter() function is called by object d
    d.input(5,10);       /* accessing base class member. input()
is a public member of base class */
    d.display(); /* accessing base class member display() is a
```

```
public member of base class */
        d.show();
        getch();
        return 0;
    }
```

OUTPUT:
Number 1 is : 5
Number 2 is : 10
Number 3 is : 15

Here, ***d*** is a derived class object. With the statement ***d.input(5,15);*** we have made a call to the function *input()* of base class by the derived class object. Similarly, we have called *display( )* member function of base class.

- **Private derivation**

When the base class is inherited by using ***private access specifier***, all the public and protected members of the base class become private members of the derived class. Therefore, public members of base class can only be accessed by the member functions of the derived class and they are not accessible to the objects of the derived class.

For example, if we use the statement

```
class derived : private base
```

instead of `class derived : public base`

in the above program, it will give two error messages while compiling:

Error: base::input(int,int) is not accessible

Error: base::display() is not accessible

As the base class is privately inherited, *input()* and *display()* become private to the derived class althogh they were public in the base class. So other functions like *main( )* cannot access them. Statement like **d.input(5,10);** and **d.display( );** will be invalid in that case.

Protected Members and Inheritance:

Protected members provide greater flexibility in case of inheritance. By using ***protected*** instead of private declaration, we can create class members that are private to their class but that can still be inherited and accessed by derived class. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it.

When a ***protected*** member is inherited in ***public*** mode, it becomes ***protected*** in the derived class too and therefore is accessible by the member functions of the derived class. It also becomes ready for further inheritance. If a base class is inherited as ***private***, then the ***protected*** member of base class becomes ***private*** in the derived class. Although it is available to the member function of the derived class, it is not available for further inheritance since ***private*** members cannot be inherited.

/***Program 11.2:** Program showing protected members inherited in public mode */

```
#include<iostream.h>
#include<conio.h>
class base
{
    protected:
int num1,num2;     /*private to base, protected to derived and accessible by derived class member function*/
    public:
    void input(int n1, int n2 )
    {
        num1=n1;
        num2=n2;
    }
    void display( )
    {
```

```
        cout<<"Number 1 is:  "<<num1<<endl;
        cout<<"Number 2 is: "<<num2;
    }
};  //end of base class
class derived : public base//base class is publicly
inherited
{
    private:
    int s;
    public:
    void add( )
    {
    s=num1+num2  ;      /* derived class accessing base class pro-
tected member num1, num2 */
    }
    void show( )
    {
        cout<<"\nSummation is : "<<s;
    }
};  //end of derived class
int main()
{
    derived d;    // d is an object of derived class
    clrscr();
d.input(10,20);  /* accessing base class member. input() is a func-
tion of base class */
d.display(); /* accessing base class member. display() is a function
of base class */
    d.add();
    d.show();
    getch();
    return 0;
}
```

OUTPUT: Number 1 is : 10
Number 2 is : 20
Summation is : 30

The derived class member function *void add()* can access *num1* and *num2* of the base class because *num1* and *num2* are declared as *protected* and the base class access specifier is *public*.

- **Protected derivation**

When the base class is inherited by using ***protected access specifier***, then all protected and public members of base class become protected members of the derived class. Let us consider the following example:

//***Program 11.3*** : Base class derived as protected

```
#include<iostream.h>
#include<conio.h>
class base
{
    protected:
        int num1,num2;
    public:
        void input(int n1, int n2 )
        {
            num1=n1;
            num2=n2;
        }
        void display( )
        {
            cout<<"Number 1 is:  "<<num1<<endl;
            cout<<"Number 2 is: "<<num2;
        }
};  //end of base class
class derived : protected base
{
    private:
```

```
        int s;
    public:
        void add( )
        {
        input(30,60);      /*member function add() of derived
class can access input() as it is inherited as protected */
        s=num1+num2; // num1,num2 are inherited as
                          // protected, so add() can access
            }
            void showall( )
            {
            display();     //display() is inherited as protected
            cout<<"\nSummation is : "<<s;
            }
    };     //end of derived class
    int main()
    {
        derived d;
        clrscr();
    //d.input(10,20); /* invalid. input() is inherited as protected member
of derived. main() can't access it */
    d.add(); /* accessing base class member display() is a function of
base class */
    d.showall(); // public member of derived
    // d.display();     /* invalid. display() can be accessible by derive
class member function only */
        getch();
        return 0;
    }
```

OUTPUT:

```
Number 1 is   :   30
Number 2 is   :   60
Summation is  :   90
```

In the program we can see that *input()*, *display()*, *num1*, *num2* of base class are inherited as protected to derive class. The member functions *add()*, *showall()* of derived class can use them; as pro-

tected members are accessible by derive class members. But main() is not a member function and it cannot access *input()* and *display()*. Although *num1*, *num2* are protected to derived class, but they behave as private to base class.



CHECK YOUR PROGRESS

1. Answer the following by selecting the appropriate option:

   (i) By using protected, one can create class members that
   (a) cannot be inherited and accessed by a derived class
   (b) can be accessed by a derived class
   (c) can be public
   (d) none of these

   (ii) Class members are by default _____
   (a) protected (b) public
   (c) private (d) none of these

   (iii) When base class access specifier is protected, then public members of base class can be accessible by
   (a) member function of derived class
   (b) main() function
   (c) objects of derived class
   (d) none of these

   (iv) Private data members of base class can be inherited by declaring them as
   (a) private (b) public
   (c) protected (d) none of these

   (v) If we donot specify the visibility mode in base class derivation then by default it will be
   (a) protected (b) private
   (c) public (d) none of these

(vi) Private data members can be accessed

(a) from derive class

(b) only from the base class itself

(c) both from the base class and from its derived class

(d) None of these

11.4 TYPES OF INHERITANCE

A program can use one or more base classes to derive a single class. It is also possible that one derived class can be used as base class for another class. Depending on the number of base classes and levels of derivation inheritance is classified into the following forms:

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

11.4.1 Single Inheritance

The programs discussed so far in this unit are examples of single inheritance. In ***single inheritance*** the derived class has only one base class and the derived class is not used as base class. The pictorial representation of single inheritance is given in Fig. 11.1

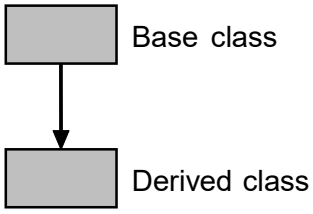


Fig. 11.1: Single Inheritance

The arrow directed from base class towards the derived class indicates that the features of base class are inherited to the derived class. In the following program, we have derived "***employee***" class from "***person***" class. Data member "***name***" and "***age***" are common to both of the two classes. "***name***" and "***age***" are declared as ***protected*** so that derive class can inherit "***name***" and "***age***" from the base class "***person***". Base class is inherited in public mode. Employee may have some other data like designation, salary. So the other data members of "***employee***" class are "***desig***" and "***salary***".

/* **Program 11.4**: Single inheritance with protected data member and public inheritance of base class */

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class person //base class "person"
{
    protected:
    char name[30];     //protected to derived class 'employee'
    float age;
    public:
    void enter(char *nm, float a)/* base class member function */
    {
        strcpy(name,nm);
        age=a;
    }
    void display()     //base class member function
    {
        cout<<"Name:  "<<name<<endl;
        cout<<"Age:  "<<age<<endl;
    }
};
class employee : public person     */base class "person" is publicly
```

```
                                    inherited */
{                                   */by derived class “employee” */
     private:
     float salary;
     char desig[20];
     public:
     void enter_data(char *n,char *d,float ag,float
s)
     {
     strcpy(name,n);   //”name” of base class can be accessible
                                    //by derived class member function
               strcpy(desig,d);
               salary=s;
               age=ag;  //age can be accessible by
                        //enter_data() of derived class
               }
               void display_all()      //derived class member
function
               {
                    display();     //can be used here as publicly
inherited
                    cout<<"Designation:
"<<desig<<endl;
                    cout<<"Salary:  "<<salary<<endl;
               }
};
int main()
{
employee e1,e2;  //e1,e2 are objects of derived class "employee"
     person p;      // p is an object of base class "Person"
     clrscr();
     e1.enter_data("Raktim","Clerk",32,5000);
     cout<<"Employee Details....."<<endl;
```

```
    e1.display_all();
e2.enter("Vaskar",41);      /*erived class object e1 accessing public member of base enter() */
e2.display();       /*derived class object e2 accessing public member of base display() */
    cout<<endl<<"Person Details....."<<endl;
    p.enter("Pragyan",24);
    p.display();
    getch();
    return 0;
}
```

Here, the derived class "***employee***" uses ***name*** and ***age*** of base class "***person***" with the help of derived class member function ***enter_data( )***.

Two different classes may have member functions with the same name as well as the same set of arguments. But in case of inheritance, an ambiguous situation arises when base class and derived classes contain member functions with the same name. In main(), if we call member function of that particular name of base class with derived class object, then it will always make a call to the derived class member function of that name. This is because, the function in the derived class *overrides* the inherited function. However, we can invoke the function defined in base class by using the ***scope resolution operator (::)*** to specify the class. For example, let us consider the following program.

/***Program 11.5**: When base and derived class has member functions with same name*/

```
#include<iostream.h>
#include<conio.h>
class B
{
```

```
  protected:
  int p;
  public:
  void enter()
  {
       cout<<"\nEnter an integer:";
       cin>>p;
  }
  void show()
  {
       cout<<"\n\nThe number in Base Class is:
"<<p;
  }
};
class D : public B
{
  private:
  int q,r;
  public:
       void enter() //overrides enter() of "B"
       {
            B::enter();
            cout<<"\nEnter an integer:";
            cin>>q;
       }
       void show()
       {
            r=p*q;
            cout<<"\nEntered numbers in Base and
                 Derived class are:"<<p<<"\t"<<q;
            cout<<"\n\nThe product is :"<<r;
       }
};
int main()
{
  D d;          //d is an object of class derived class "D"
```

```
    clrscr();
    d.enter();          //invokes enter() of "D"
    d.show();           //invokes show() of "D"
    d.B::show(); //invokes show() of "B"
    getch();
    return 0;
}
```

In the program, the function name *show()* is same in both base “B” and derived class “D”. To call show() of base class “B”, we have used the statement **d.B::show();** If we use simply **d.show();** then it will invoke show() of derived class “D”.

When a derive class implements a function that has the same name as well as the same set of arguments as the function in the base class, it is called ***function overriding***. When such a function is called through a object of derived class, then the derived class function would be invoked. However, that function in base class would remain hidden.

But there are certain situations where function overriding plays an important role.

11.4.2 Multiple Inheritance

When one class is derived from two or more base classes then it is called ***multiple inheritance***. This type of inheritance allows us to combine the properties of more than one existing classes in a new class. Fig. 11.2 depicts multiple inheritance

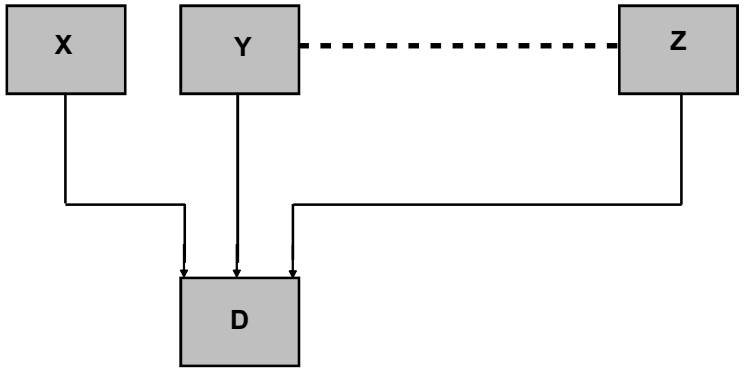


Fig.11.2: Multiple inheritance

We have to specify the base classes one by one separated by commas with their access specifiers. The general form of deriving a derived class from numbers of base class is as follows:

```
class D : public X, public Y, public Z
{
     ......  //body of the derived class
};
```

where X,Y, Z are base classes and D is the derived class. There may be numbers of base classes in multiple inheritance which is indicated by the dotted line in the figure.

For demonstration of multiple inheritance let us consider the following program. There are three base classes and one derived class. The derived class CHARACTER has one private member “n” and two public member functions “enter()” and “show()”. The function “enter()” is used to read a number, a vowel, a consonent and a symbol from the keyboard and the “show()” function is used to display the contents on the screen. The class members of all the three base classes are publicly derived.

// **Program 11.6** : Example of Multiple inheritance

```
#include<iostream.h>
#include<conio.h>
class V //base class
{
    protected:
         char v;
};
class C //base class
{
    protected:
         char c;
};
class S //base class
```

```
{
    protected:
        char s;
};
class CHARACTER : public V, public C, public S
{
    private:
        int n;
    public:
        void enter() //derived class member function
        {
            cout<<"\nEnter a vowel:";
        cin>>v;//accessing protected member v of class
"VOWEL"
            cout<<"\nEnter a consonent:";
        cin>>c; //accessing c of "CONSONENT" class
        cout<<"\nEnter a symbol:";
        cin>>s  //accessing s of "SYMBOL" class
        cout<<"\nEnter a number:";
        cin>>n; //accessing n of "NUMBER" class
    }
    void show()
    {   cout<<"\nThe entered characters are
:\n\n";
        cout<<"\nVowel: "<<v;
        cout<<"\nConsonent: "<<c;
        cout<<"\nSymbol: "<<s;
        cout<<"\nNumber: "<<n;
    }
};
int main()
{
    CHARACTER o; //o is an object of derived class "char-
acter"
    clrscr();
```

```
    o.enter();
    o.show();
    getch();
    return 0;
}
```

One suitable example of the implementation of multiple inheritance is shown in the program below:

/***Program 11.7**: Program showing multiple inheritance with two base class (practical, theory) and one derived class (result) /*

```
#include<iostream.h>
#include<conio.h>
class practical  //base class "practical"
{
    protected:
    float p1_marks, p2_marks,total;
    public:
    void practical_marks()
    {
    cout<<"Enter marks of practical paper 1 and
paper 2: ";
        cin>>p1_marks>> p2_marks;
    }
    float add()  //returns the total of practi-
cal
    {
        total=p1_marks+p2_marks;
        return total;
    }
    void display_practical()
    {
        cout<<endl<<"Total      Practical
marks:"<<total;
    }
```

```
};      //end of "practical" class
class  theory // base class "theory"
{
    protected:
    float phy, chem, math, total_marks;
    public:
    void  theory_marks(){
cout<<"Enter marks of Physics, Chem and Maths:";
    cin>>phy>>chem>>math;
}
float sum()
{
    total_marks=phy+chem+math;
    return  total_marks;
}
void  display_theory()
{
    cout<<endl<<"Total                Theory
marks:"<<total_marks;
}
};  //end of base class "theory"
class result : public practical,public theory
{
    protected:
    int roll;
    float  grand_total,t,p;
    public:
    void enter() {
cout<<"ENTER  STUDENT  INFORMATION......"<<endl;
    cout<<"Enter Roll no.:";
    cin>>roll;
theory_marks();   //inherited publicly from base class "theory"
practical_marks();      //inherited from base class "practical"
}
void  theory_practical()
{
```

```
        t=sum()
        p=add();
        grand_total=t+p;
cout<<"\nThe total marks of the student is:"
<<grand_total;
}
};  //end of derived class "result"
int main()
{
        result s1;   //object of derived class
        clrscr();
        s1.enter();
        s1.theory_practical();// accessing derived class member
s1.display_theory(); //s1 accessing "display_theory()" of
"theory"
        s1.display_practical();
        getch();
        return 0;
}
```

When we execute the program entering marks for practical and theory papers for a particular student, then it will display the result as follows:

ENTER STUDENT INFORMATION.....
Enter Roll no.: 1
Enter marks of Physics, Chemistry and Mathematics: 65 72 81
Enter marks of practical paper 1 and paper 2 : 25 26
The total marks of the student is : 269
Total Theory marks : 218
Total Practical marks : 51

The above program consists of three classes: two base classes ("***practical***" and "***theory***") and one derived class ("***result***"). The member function "***enter()***" of derive class inherits member functions "***practical_mark()***" and "***theory_ marks()***" of base class

"***practical***" and "***theory***" respectively. Similarly, member function "***theory_practical()***" uses "***sum()***" and "***add()***" of base class to calculate the "*grand_total*" marks of student. Thus, in the derived class we need not have to write functions for entering practical and theory marks. We just inherit them from the base classes.



EXERCISE

Q. Suppose a class D is derived from class B. B has two public member functions *getdata()* and *showdata()* and D has two public functions *readdata()* and *displayall()*. Define the classes such that both function *getdata()* and *showdata()* should be accessible in the *main()* function.

11.4.3 Hierarchical Inheritance

Derivation of more than one classes from a single base class is termed as ***hierachical inheritance***. This is a very common form of inheritance in practice. The rules for defining such classes are the same as in single inheritance. The pictorial representation of hierarchical inheritance is shown in Fig. 11.3

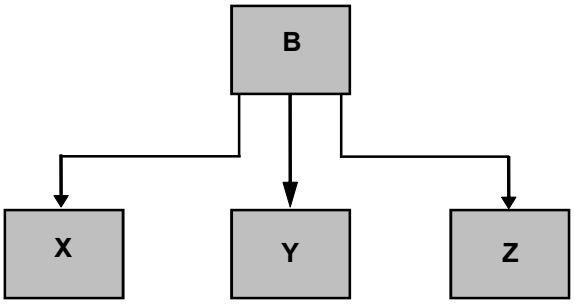


Figure11 3: Hierarchical Inheritance

For demonstration of hierarchical inheritence let us consider a program with one base class ("student") and three derived classes ("arts", "science" and "commerce").

//**Program 11.8**: Demonstration of hierarchical inheritance

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

```
#include<conio.h>
class student    // base class "student"
{
    protected:
    char  fname[20],lname[20];
    int age,roll;
    public:
    void  student_info()
    {
        cout<<"Enter the first name and last name:
";
        cin>>fname>>lname;
        cout<<"\nEnter the Roll no.and Age: ";
        cin>>roll>>age  ;
    }
    void display()
    {
        cout<<"\nRoll Number = "<<roll;
        cout<<"\nFirst        Name        =
"<<fname<<"\t"<<lname;
        cout <<"\nAge = " << age;
    }
};
class arts : public student  //derived class arts
{
    private:
    char asub1[20], asub2[20], asub3[20] ;
    public:
    void enter_arts()
    {
        student_info(); //base class member function
        cout <<"\n Enter the subject1 of the arts
student:";
    cin >> asub1 ;
    cout<<"\nEnter the subject2 of the arts stu-
dent:";
```

```
    cin >> asub2 ;
    cout<<"\nEnter the subject3 of the arts stu-
dent:";
    cin >> asub3 ;
}
void display_arts()
{
    display();//base class member function
    cout<<"\n\t Subject1 of the arts student="<<
asub1;
    cout<<"\n\t Subject2 of the arts student="<<
asub2;
    cout<<"\n\t Subject3 of the arts student="<<
asub3;
}
};
class commerce : public student    //derived class
"commerce"
{
    private:
    char csub1[20], csub2[20], csub3[20] ;
    public:
    void enter_com(void)
    {
    student_info();  //base class member function
cout<<"\tEnter the subject1 of the commerce student:";
    cin>> csub1;
cout<<"\tEnter the subject2 of the commerce student:";
    cin>>csub2 ;
cout<<"\tEnter the subject3 of the commerce student:";
    cin>> csub3 ;
    }
    void display_com()
    {
    display();   //base class member function
cout<<"\nSubject1 of the commerce student="
```

```
<<csub1;
cout<<"\nSubject2 of the commerce student="
<<csub2;
cout<<"\nSubject3 of the commerce student="<< csub3
     }
     };
class science : public student  //derived class "sci-
ence"
{
     private:
     char ssub1[20], ssub2[20], ssub3[20] ;
     public:
     void enter_sc(void)
     {
     student_info();  //base class member function
cout<<"\nEnter the subject1 of science student:";
     cin>>ssub1;
cout<<"\nEnter the subject2 of science student:";
     cin>>ssub2 ;
cout<<"\nEnter subject3 of the science student:";
     cin>>ssub3 ;
     }
     void display_sc()
     {
     display();   //base class member function
cout<<"\nSubject1 of the science student="<< ssub1
cout<<"\nSubject2 of the science student="<< ssub2;
cout<<"\nSubject3 of the science student="<< ssub3;
     }
};
int main()
{
     arts a ; //a is an object of derived class "arts"
     clrscr();
     cout << "\n Entering details of the arts
```

```
student\n";
    a.enter_arts( );
    cout <<"\nDisplaying the details of arts
student\n";
    a.display_arts( );
    science s;   //s is an object of derived class "sci-
ence"
    cout <<"\n\n Entering details of science
student\n"
    s.enter_sc( );
    cout<<"\n Display details of the science
student\n";
    s.display_sc( );
    commerce c ;    //c is an object of derived class
"commerce"
    cout<<"\n\nEnter details of commerce
student\n";
    c.enter_com( );
    cout << "\n Display details of commerce
student\n";
    c.display_com() ;
    getch();
    return 0;
}
```

11.4.4 Multilevel Inheritance

C++ also provides the facility of ***multilevel inheritance***, according to which the derived class can also be derived by an another class, which in turn can further be inherited by another and so on. For instance, a class X serves as a base class for class Y which in turn serves as base class for another class Z. The class Y which forms the link between the classes X and Y is known as the ***intermediate base class***. Further, Z can also be used as a base class for another new class. The following figure depicts multilevel inheritance.

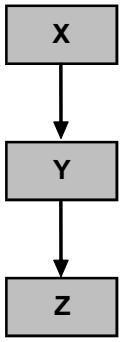


Figure11.4: Multilevel Inheritance

11.4.5 Hybrid Inheritance

It is possible to derive a class involving more than one type of inheritance. ***Hybrid inheritance*** is that type of inheritance where several forms of inheritance are used to derive a class. There could be situations where we need to apply two or more types of inheritance to design a particular program.

For example, let us assume that we are to design a program which will select players for a particular compitetion. For this purpose we could consider four classes PLAYER, GAME, RESULT and PHYSIQUE. PLAYER class contains the player details including name, address, location etc. GAME class can be derived from PLAYER class. Again, if weightage for physical test should be added before finalizing the result, then we can inherit that from PHYSIQUE. RESULT class is derived from two base classes GAME and PHYSIQUE. The following diagram gives us the inhertance relationship between various classes.

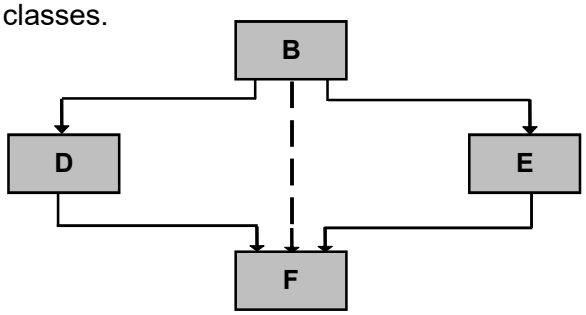


Fig.11.5: Hybrid Inheritance

In the diagram, RESULT has two base classes,GAME and PHYSIQUE. GAME is not only a base class but also a derived class. Here we can see that two types of inheritance *multiple* and *multilevel* are combined to create the RESULT class.

11.4.6 Multipath Inheritance

The inheritance where a class is derived from two or more classes, which are in turn derived from the same base class is known as ***multipath inheritance***. There may be many types of inheritance such as multiple, multilevel, hierarchical etc. in multipath inheritance. Certain difficulties may arise in this type of inheritance. Suppose we have two derived classes D and E that have a common base class B, and we have another class F that inherits from D and E.

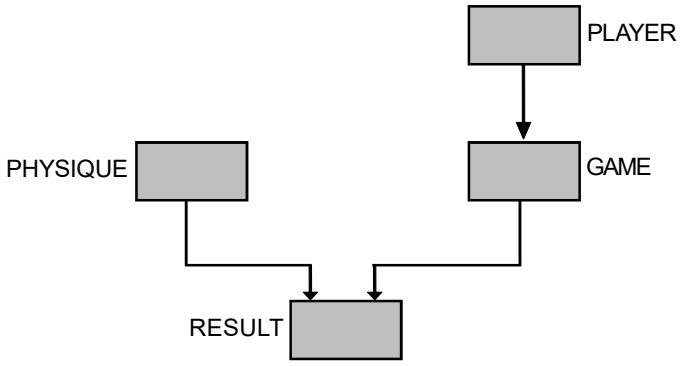


Fig.11.6 : Multipath Inheritance

In the above diagram, we can observe three types of inheritances, i.e., multiple, multilevel and hierarchical. For better illustration let us consider the following program.

// **Program 11.9:** Demosntration of multipath inheritance

```
#include<iostream.h>
#include<conio.h>
class B
{
    protected:
    int b;
```

```
};
class D : public B    //B is publicly inherited by D
{
    protected:
    int d;
};
class E : public B    //B is publicly inherited by E
{
    protected:
    int e;
};
class F : public D, public E//D,E are publicly inher-
ited by F
{
    protected:
    int f;
    public:
    void enter_number()
    {
        cout<<"Enter some integer values for
b,d,e,f:";
        cin>>b>>d>>e>>f;
    }
    void display()
    {
    cout<<"\nEntered numbers are:\n";
    cout<<"\nb= "<<b<<"\nd= "<<d<<"\ne= "<<e<<"\nf=
"<<f;
    }
};
int main()
{
    F obj;        //instantiaton of class F
    clrscr();
    obj.enter_number();  //enter_number() of F is called
```

```
    obj.display();
    getch();
    return 0;
}
```

Now, if we instantiate class F and call the functions ***enter_number()*** and ***display()***, then the compiler shows the following types error messages:

Error M.cpp 28: Member is ambiguous:'B ::b' and 'B::b'

Error M.cpp 33: Member is ambiguous:'B ::b' and 'B::b'

This is due to the duplication of members of class B in F. The member **b** of class B is inherited twice to class F: one through class D and another through class E. This leads to ambiguity. To avoid such type of situation, ***virtual base class*** is introduced.

11.5 VIRTUAL BASE CLASSES

C++ provides the concept of ***virtual base class*** to overcome the ambiguity occurring due to *multipath inheritance*. While discussing multipath inheritance, we have faced a situation which may lead to duplication of inherited members in the derived class F (Fig.11.6). This can be avoided by making the common base class (i.e.,B) as *virtual base class* while derivation. We can declare the base class B as ***virtual*** to ensure that D and E share the same data member B.

This is shown in the following program which is the modification of the previous program 11.9.

/***Program 11.10:** Virtual base class and removal of ambiguity occuerd in multipath inheritance */

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

```
{
    protected:
    int b;
};
class D:public virtual B
//B is publicly inherited by D and made virtual
{
    protected:
    int d;
};
class E:public virtual B
//B is publicly inherited by E and made virtual
{
    protected:
    int e;
};
class F : public D, public E
//D,E are publicly inherited by F
{
    protected:
    int f;
    public:
    void enter_number()
    {
        cout<<"Enter some integer values for b,d,e,f:
";
        cin>>b>>d>>e>>f;
    }
    void display()
    {
        cout<<"\nEntered numbers are:\n";
        cout<<"\nb= "<<b<<"\nd= "<<d<<"\ne=
"<<e<<"\nf= "<<f;
```

```
    }
};
int main()
{
    F obj;  //instantiaton of class F
    clrscr();
    obj.enter_number();
    obj.display();
    getch();
    return 0;
}
```

Here we have used the keyword virtual in front of the base class specifiers to indicate that only one subobject of type B, shared by class D and class E, exists.. When a class is made a ***virtual base class***, C++ takes the necessary action that only one copy of that class is inherited, regardless of how many paths exist between the virtual base class and a derived class.

11.6 ABSTRACT CLASSES

The objects created often are the instances of a derived class but not of the base class. The base class just becomes a structure or foundaion with the help of which other classes are built and hence such classes are called ***abstract class*** or ***abstract base class***. In other words, when a class is not used for creating objects then it is called *abstract class*. In the *Program 10.10*, B is an abstract class since it was not used for creating any object.



LET US KNOW

Inheritance and Constructors, Destructors

Although constructors are suitable for initializing objects , we have not used them in any program in this unit for the sake of simplicity. But if we use contructors in program, then we must follow certain definite rules

while inheriting derive classes. If the base class contains no argument constructor then the derived class does not require a constructor. If any base class contains parameterized constructor, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors. In case of inheritance, normally derived classes are used to declare objects. Hence it is necessary to define constructor in the derived class. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in derived class is executed. Destructors are executed in the reverse order of constructor execution.



CHECK YOUR PROGRESS

2. Select whether the following statements are True (T) or False (F):
   (i) A class can serve as base class for many derived classes.
   (ii) When one class is derived from another derived class then that is called *multiple inheritance*.
   (iii) When one class is derived from more than one base class then that is called *multiple inheritence*.
   (iv) When more than one form of inheritance is used in designing a class then that type is called *hybrid inheritance*.
3. Answer the following by selecting the appropriate option:
   (i) In multilevel inheritance, the middle class acts as
   (a) only derived class (b) only base class
   (c) base class as well as derived class
   (d) none of the above
   (ii) A class is declared “virtual” when
   (a) more than one class is derived
   (b) two or more classes have common base class
   (c) there are more than one base classes

(d) none of the above

(iii) When a class is not used for creating objects, it is called

(a) abstract class (b) virtual base class

(c) derived class (d) none of these

(iv) *Intermediate base class* is present in case of

(a) single inheritance (b) multiple inheritance

(c) multilevel inheritance (d) hierarchical inheritance



11.7 LET US SUM UP

- ***Inheritance*** is one of the most useful and essential characteristics of object-oriented programming language. If we have developed a class and we want a new class that is almost similar, but slightly different, the principles of inheritance come handy. The existing class is known as ***base*** class and the newly formed class is known as ***derived*** class.The derived class can have some other characteristics which are not in base class.
- Private members of a class cannot be inherited either in public mode or in private mode.
- When a public member inherited in public, protected and private mode, then in derived class it remains with the same access specifiers as in base class i.e., public, protected and private respectively.
- A protected member inherited in public mode becomes protected, whereas inherited in private mode becomes private in derived class.
- The protected and public data
- In ***single inheritance***, one new class is derived from a single base class.
- When a class is derived using the properties of several base classes, then it is called ***multiple inheritance.***

- The process of deriving a class from another derived class is called ***multilevel inheritance***.
- More than one class can be derived from only one base class i.e., characteristics of one class can be inherited by more than one class. This is called ***hierarchical inheritance***.
- When different types of inheritance are applied in a single program then it is termed as ***hybrid inheritance***.
- When a class is derived from two or more classes, which are derived from the same base class, such type of inheritance is known as ***multipath inheritance***.
- We can make a class ***virtual*** if it is a base class that has been used by more than one derived class as their base class. When classes are declared as *virtual*, the compiler takes necessary caution to avoid the duplication of the member variables.
- When a class is not used for creating objects, it is called an ***abstract class***.



11.8 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education



11.9 ANSWERS TO CHECK YOUR PROGRESS

1. (i) (b) can be accessed by a derived class
   (ii) (c) private
   (iii) (a) member function of derived class
   (iv) (c) protected

(v) (b) private

(vi) (b) only from the base class itself

2. (i) True (ii) False

(iii) True (iv) True

3. (i) (c) base class as well as derived class

(ii) (b) two or more classes have common base class

(iii) (a) abstract class

(iv) (c) multilevel inheritance



11.10 MODEL QUESTIONS

1. What does inheritance mean in C++?
2. What are the types of inheritance? Explain any three of them with examples.
3. What are the different types of visibility modes of base class?
4. Write a program to derive a class from multiple base classes.
5. When do we make a class virtual?
6. What are abstract base classes?
7. Explain multipath inheritance.
8. Write a C++ program involving appropriate type of inheritance which will inherit two classes *triangle* and *rectangle* from *polygon* class. Use member functions for entering appropriate parameters like *width*,*height* etc. and to calculate the area of triangle and rectangle.
9. Consider a case of University having the disciplines of Engineering, Management, Science, Arts and Commerce. There are many colleges in the University. Assuming a college can run a course partaining to only one discipline, draw the class diagram.To which type of inheritance does this structure belong?

****

UNIT 12 : VIRTUAL FUNCTIONS AND POLYMORPHISM

UNIT STRUCTURE

12.1 Learning Objectives
12.2 Introduction
12.3 Polymorphism
    12.3.1 Types of Polymorphism in C++
12.4 Virtual Functions
12.5 Pure Virtual Functions
12.6 Let Us Sum Up
12.7 Further Reading
12.8 Answers to Check Your Progress
12.9 Model Questions

12.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- describe polymorphism and its types
- define the rules for virtual function
- describe how to use virtual function to achieve run-time polymorphism
- describe and implement pure virtual function

12.2 INTRODUCTION

In the previous unit, we have studied the concept of inheritance and its importance in Object-Oriented Programming language like C++.

In this unit, we will discuss one useful feature of Object-Oriented Programming, ***polymorphism***. It is the ability of objects to take different forms. The ability to display variable behavior depending on the situation is a great facility in any programming language. In the earlier units, we have seen operator overloading and function overloading. Those are also one kind of polymorphism. C++ supports a mechanism known as ***virtual function*** to

achieve run-time polymorphism. The necessity and usefulness of virtual functions in programming will also be covered in this unit.

12.3 POLYMORPHISM

The word *polymorphism* is a combination of two Greek words, *poly* and *morphism*. “Poly” means “many” and “morphism” means “form”. The functionality of this feature accords with its name.

12.3.1 Types of Polymorphism in C++

Polymorphism is supported by C++, at both compile-time and at run-time. Hence, there are two types of polymorphism:

- ***Compile-time Polymorphism***
- ***Run-time Polymorphism***

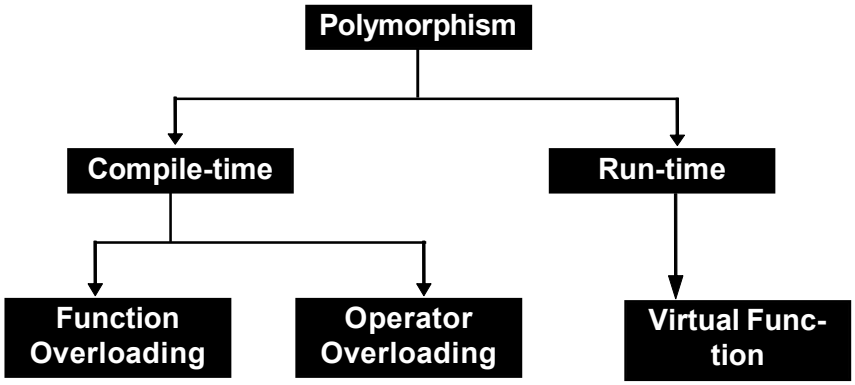


Fig. 12.1: Polymorphism in C++

Operator overloading is achieved by allowing operators to operate on the user defined data type with the same manner as that of built-in data types. For example, plus “+” operator produces different actions in case of integers, complex numbers or strings. With the help of ***function overloading***, we can write different functions by using the same *function name* but with *different argument lists*. The functon would perform different operations depending on the argument list in the functon call. The overloaded member functions are selected

for invoking by matching the number of arguments and type of arguments. This information is known to the compiler at the compile time itself and, therefore, the selection of the appropriate function is made at the compile time only.

In both cases, the compiler is aware of the complete information regarding the type and number of operands. Hence, it is possible for the compiler to select a suitable function at compile time. This is known as ***compile-time polymorphism***. It is also termed as ***static binding*** or ***early binding***.

Let us consider a program where the function name and argument list are same in both the base and derived class.

//**Program 12.1:**

```
#include<iostream.h>
#include<conio.h>
class B   //base class
{
     protected:
          int n;
     public:
          void enter()
          {
               cout<<"Enter a number in base class:\n";
               cin>>n;
          }
          void display()
          {
               cout<<"\nThe number in base class is: "<<n;
          }
};   //end of base class declaration
class D:public B   //derived class D
{
     private:
          int num;
     public:
```

```
        void input()
        {
    cout<<"\nEnter a number in derived class:";
    cin>>num;
    }
    void display( )
    {
    cout<<"\nThe number in derived class: "<<n;
    }
};
int main()
{
    D d;
    clrscr();
    d.enter();   //will call the enter() of base class
    d.display( );//display() of derived class will be invoked,
    getch();
    return 0;
}
```

Output of the above program will be like this:

Enter a number in base class : 6

The number in derived class : 6

But our intention is to display is :

The number in base class : 6

It has been observed that prototype of **display()** is same in both base and derived class and we cannot term it as *function overloading*. Thus *static binding* does not apply in this case. We have already used statement like **d.B::show();** in such situation (*program 10.5* of unit *Inheritance);* i.e., we used the scope resolution operator (**::**) to specify the class while invoking the functions with the derived class objects. But it would be nice if the appropriate member function could be selected while the program is running. With the help of inheritance and virtual functions, C++ determines which version of that function to call. This determination is made at run-time and is known as ***run-time polymorphism***. Here, the function is linked with a particular class much later after the compilation and thus it is also

known as ***late binding*** or ***dynamic binding***. In the following section, we will discuss how to implement virtual function to achieve run-time polymorphism.

12.4 VIRTUAL FUNCTIONS

The concept of virtual functions is different from function overloading. A ***virtual function*** is a member function that is declared within a base class and redefined by a derived class. The whole function body can be replaced with a new set of implementation in the derived class. To make a function virtual, the ***virtual*** keyword must precede the function declaration in the base class. The redefinition of the function in any derived class does not require a second use of the *virtual* keyword. The difference between a non virtual member function and a virtual member function is that the non virtual member functions are resolved at compile time whereas the virtual member functions are resolved during run-time.

The concept of *pointers to object* is prior to knowing before implementing virtual function. We have already studied the concept of pointers in earlier units. At this point, we shall discuss how class members are accessible with the help of pointers.

Pointers to Objects

A pointer can point to a class object. This is called ***object pointer***. Object pointers are useful in creating objects at run time and public members of class can be accessible by object pointers. For example, we can create pointers pointing to classes, as follows:

```
polygon *optr;
```

i.e., class name followed by an asterik (*) and then the variable name. Thus, in the above declaration, ****optr*** is a pointer to an object of class ***polygon***. To refer directly to a member of an object pointed by a pointer we can use arrow operator (**->**). Here is a program for the illustration of object pointers:

//**Program 12.2**: Demonstration of pointer to object

```
#include<iostream.h>
#include<conio.h>
class polygon
{
     protected:
     int width, height;
     public:
     void set_values (int w, int h)
     {
          width=w;
          height=h;
     }
     void display()
     {
          cout<<"Width :"<<width<<endl<<"Height : <<height;
     }
};
int main ()
{
     polygon  p;        // p is an object of type polygon
     polygon *optr = &p;// creation and initiazation of
                        // object pointer
     optr->set_values (8,6);//object pointer accessing member
     optr->display( );//function "set_values()" and
                      //"display( )"
     getch();      // with arrow operator.
     return 0;
}
```

With the statement polygon *optr = &p;

we have created object pointer ***optr*** of type ***polygon*** and initialized it with the address of ***p*** object. We can also create the objects using pointers and ***new*** operator as follows:

polygon *optr = **new** polygon;

This statement allocates enough memory for the data members in the object of the particular class and assigns the address of the memory space to ***optr***.

Pointer to base and derived class objects

Pointers can also be used to point base or derived class object. Pointers to object of base class is a type-compatible with a pointer to object of

derived class. If we create a base class pointer, then that pointer can point to object of base as well as object of derived class.

For example, let us consider the following program :

//**Program 12.3**:

```
#include<iostream.h>
#include<conio.h>
class polygon
{
    protected:
        int width, height;
    public:
        void set_values(int w, int h)
        {
            width=w;
            height=h;
        }
};
class rectangle: public polygon //derived class rectangle
{
    public:
        int area()
        {
            return  (width*height);
        }
};
class triangle: public polygon  //derived class triangle
{
    public:
        int area()
        {
            return (width*height / 2);
        }
};
int main ()
{
```

```
    rectangle r; // derived class object r
    triangle t;  // derived class object t
    clrscr();
    polygon *p1 = &r;//base class pointer pointing derived class object r
    polygon *p2 = &t;   // p2 pointing to object t of triangle class
    p1->set_values(5,6);
    p2->set_values(5,6);
    cout<<"\nArea of the rectangle is :"<<r.area()
<<endl;
    cout<<"\nArea of the triangle is :" <<t.area()
<<endl;
        getch();
        return 0;
    }
```

The output of the programm will be like this:

Area of the rectangle is : 30
Area of the triangle is : 15

In function main, we create two pointers ***p1*** and ***p2*** that point to objects of class ***polygon***. Then we assign references to ***r*** and ***t*** to these pointers. Both are valid assignment operations as because both are objects of classes derived from ***polygon***. The only limitation in using ****p1*** and ****p2*** instead of ***r*** and ***t*** is that both ****p1*** and ****p2*** are object pointers of type ***polygon*** and therefore we can only use these pointers to refer to the members that ***rectangle*** and ***triangle*** inherit from ***polygon***.

The use of pointer to objects of base class with the objects of its derived class does not allow access even to public members of a derived class. That is, it allows access only to those members inherited from the base class but not to the members which are defined to the derived class. For that reason when we call the ***area()*** members at the end of the program we have had to use directly the objects ***r*** and ***t*** instead of the pointers ****p1*** and ****p2***.

In order to use ***area()*** with the pointers to base class ***polygon***, this member should also have been declared in the class ***polygon***, and not only in its derived classes. But the problem is that, ***rectangle*** and ***triangle*** implement

different versions of ***area()***. Therefore, we cannot implement it in the base class ***polygon***. In such situations, ***virtual functions*** are necessary.

A pointer to a derived class object may be assigned to a base class pointer, and a ***virtual function*** called through the pointer. If the function is virtual and occurs both in the base class and in derived classes, then the right function will be picked up based on what the base class pointer really points at.

//**Program 12.3**: Program demonstrating the use of virtual function

```
#include<iostream.h>
#include<conio.h>
class polygon
{
    protected:
        int width, height;
    public:
        void set_values(int w, int h)
        {
            width=w;
            height=h;
        }
        virtual int area()    //virtual function
        {
            return (0);
        }
};
class rectangle: public polygon
{
    public:
        int area()    //virtual function redefined
        {
            return (width*height);
        }
};
class triangle: public polygon
```

```
{
    public:
        int area()    //virtual function redefined
        {
            return (width * height / 2);
        }
};
int main()
{
    rectangle r; //r is an object of derived class rectangle
    triangle t;  //t is an object of derived class triangle
    polygon p;   //p is an object of base class polygon
    clrscr();
    polygon *p1=&r;  //pointer to a derived class object r
    polygon *p2=&t;
    polygon *p3=&p;
    p1->set_values(5,6);
    p2->set_values (5,6);
    p3->set_values (5,6);
    cout<<"Area of rectangle is:"<<p1-
>area()<<endl;
    cout<<"Area of triangle is: "<<p2-
>area()<<endl;
    cout<<"Area in polygon is:"<<p3->area()<<endl;
    getch();
    return 0;
}
```

In the above program, the three classes ***polygon***, ***rectangle*** and ***triangle*** have one common member function: ***area()***. The member function ***area()*** has been declared as ***virtual*** in the base class and it is later redefined in each derived class. The output of the program willl be like this:

Area of rectangle is : 30

Area of triangle is : 15

Area in polygon : 0

If we remove the ***virtual*** keyword from the declaration of ***area()*** within *polygon* and run the program, the result will be 0 for the three polygons instead of 30, 15 and 0. That is because instead of calling the corresponding ***area()*** function for each object (*rectangle::area()*, *triangle::area()* and *polygon::area()*, respectively), ***polygon::area()*** will be called in all cases since the calls are via a pointer of type ***polygon***. A class that declares or inherits a virtual function is called a ***polymorphic class***.

When functions are declared as virtual, the compiler adds a data member secretly to the class. This data member is referred to as a ***virtual pointer (VPTR)***. A table called ***virtual table (VTBL)*** contains pointers to all the functions that have been declared as virtual in a class, or any other classes that are inherited. Whenever a call to a virtual function is made in the C++ program, the compiler generates code to treat VPTR as the starting address of an array of pointers to functions. The function call code simply indexes into this array and calls the function located at the indexed addresses. The binding of function call always requires this dynamic indexing activities which always happens at run-time. If a call to a virtual function is made, while treating the object in question, as a member of its base class, the correct derived class function will be called. Thus, dynamic binding is achieved with the help of virtual functions.

There are some definite rules for writing virtual function. These rules are:

- The virtual functions must be members of some class.
- Object pointers should be used to access virtual function.
- A virtual function in a base class must be defined even though it may not be used.
- The prototype of the function which we declare as *virtual* in the base class must be same with all its derived class versions.
- A base pointer can point to any type of the derived object. But we cannot use a pointer to a derived class to access an object of the base class.
- Constructors cannot be virtual but destructors can be virtual.



CHECK YOUR PROGRESS

1. Choose the appropriate option for the correct answer:
   (i) Run-time polymorphim can be accomplished with the help of
   (a) operator overloading (b) function overloading
   (c) virtual function (d) friend function
   (ii) Static binding is associated with
   (a) compile-time polymorphism
   (b) run-time polymorphism
   (c) virtual function (d) none of these
   (iii) Pionter to object of base class can point
   (a) base class object (b) derived class object
   (c) both (a) and (b) (d) none of these
   (iv) Virtual functions can be accessible by
   (a) scope resolution operator
   (b) object pointer
   (c) object (d) none of these
   (v) The ability to take many forms is called .....
   (a) encapsulation (b) polymorphism
   (c) inheritance (d) none of these
2. State which of the following statements are True (T) or False F) :
   (i) The prototype of the function which we declare as *virtual* in the base class must be different with all its derived class versions.
   (ii) Run-time polymorphism can be achieved only when a virtual function is accessed through a pointer to the base class.
   (iii) Functions and operator overloading are examples of compile-time polymorphism.

12.5 PURE VIRTUAL FUNCTIONS

Generally, we declare a virtual function inside a base class and redefine it in the derived classes. In many situations, there can be no meaningful definition of a virtual function within a base class. Most of the times, the idea behind declaring a function virtual (in the base class), is to stop its execution.Then the question arises why should we define virtual functions? This leads to the idea of pure virtual functions.

For example, in the previous program *12.3*, we have defined a virtual function *area()* within the base class *polygon* . We have also created objects of *polygon* class and made a call to its own *area()* function with object pointer. As the function has minimal functionality, we could leave that *area()* member function without any definition in the base class. This can be done by appending =0 (equal to zero) to the function declaration as follows:

```
virtual int area( ) = 0;
```

Such functions are called pure virtual functions. The general form of declaring a pure virtual function is:

```
virtual return_type function_name(parameter_list) = 0;
```

A ***pure virtual function*** is a virtual function that has no definition within the base class. It only serves as a placeholder. In such cases, the compiler requires each derived class to either define the function or redeclare it as pure virtual function. A class containing pure virtual functions cannot be used to declare objects of its own. Such classes are known as ***abstract base class***. As stated earlier, when a class is not used for creating objects then it is called *abstract class or abstract base class*, similarly, a class containing pure virtual functions cannot be used for creating objects. A class that cannot instantiate objects is not useless. We can create pointers to it and take advantage of all its polymorphic abilities. Let us examine the working of pure virtual functions with an example:

//**Program 12.4**: Demonstration of pure virtual function

```
#include<iostream.h>
#include<conio.h>
class polygon
{
    protected:
        int width, height;
    public:
        void set_values(int w, int h){
            width=w;
            height=h;
        }
        virtual int area() = 0;//pure virtual function
};
class rectangle: public polygon
{
    public:
        int area()
        {
            return  (width*height);
        }
};
class triangle: public polygon{
        public:
            int area()
            {
                return (width * height / 2);
            }
};
int main()
{
    rectangle  r; //r is an object of derived class rectangle
    triangle  t;  //t is an object of derived class triangle
    clrscr();
```

```
    polygon *p1=&r;  //p1 points to object r
    polygon *p2=&t;  //p2 points to object t
    p1->set_values(5,6);
    p2->set_values (5,6);
    cout<<"Area of rectangle is:"<<p1->area()<<endl;
    cout<<"Area of the triangle is:"<<p2-
>area()<<endl;
    getch();
    return 0;
}
```

The output will be like this:

Area of the rectangle is : 30

Area of the rectangle is : 15

We can observe that, here we refer to objects of different but related classes using a unique type of pointer (polygon *p1,*p2). In the *main()* function, if we try to create object of *polygon* class with statement like **polygon p;** then the compiler will give error message of the following type:

Error: Cannot create instance of abstract class ‘polygon’.

We should remember that when a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile time error will occur.

Virtual function and dynamic allocation of objects

Virtual member function can also be implemented with dynamically allocated objects. Let us demonstrate the same example with objects that are dynamicalled allocated.

/***Program 12.5**: Demonstration of pure virtual function and dynamically allocated object */

```
#include<iostream.h>
#include<conio.h>
class polygon
{
```

```
    protected:
        int width, height;
    public:
        void set_values(int w, int h)
        {
            width=w;
            height=h;
        }
        virtual int area()=0;   //pure virtual function
};
class rectangle: public polygon
{
    public:
        int area()
        {
            return  (width*height);
        }
};
    class triangle: public polygon
{
    public:
        int area()
        {
            return (width * height / 2);
        }
};
int main()
{
    polygon *p1=new rectangle;
    polygon *p2=new triangle;
    clrscr();
    p1->set_values(5,6);
    p2->set_values  (5,6);
    cout<<"Area of rectangle is:"<<p1->area()<<endl;
    cout<<"Area of triangle is:"<<p2->area()<<endl;
    delete p1;
```

```
    delete p2;
    getch();
    return 0;
}
```

In the main() function, we have used the following statements:

polygon * p1= new rectangle;

polygon * p2= new triangle;

Here the pointer *p1* and *p2* are declared being of type pointer to *polygon* but the objects dynamically allocated have been declared having the derived class type directly.



CHECK YOUR PROGRESS

3. Choose the appropriate option for the corect answer:
   (i) Dynamic binding is done using the keyword
   (a) static (b) dynamic
   (c) virtual (d) abstract
   (ii) Virtual function helps us in achieving
   (a) run-time polymorphism
   (b) compile-time polymorphism
   (c) both (a) and (b) (d) none of these
   (iii) A base class which is not used for object creation is called
   (a) abstract class (b) derived class
   (c) virtual class (d) none of these
   (iv) The function in the statement *virtual show()=0;* is a
   (a) virtual function (b) pure member function
   (c) friend function (d) pure virtual function
   (v) A ..... pointer can point to ..... object
   (a) derived class, base class
   (b) void, NULL (c) base class, derived class
   (d) none of these
4. State which of the following statements are True(T) or False(F):
   (i) Class containg pure virtual function can instantiate objects of its own.

(ii) Pointers to objects of a base class type are compatible with the pointer to objects of a derived class.

(iii) A virtual function is a member function that expects to be overridden in a derived class.



12.6 LET US SUM UP

- Polymorphism is the ability to use an operator or function in different ways. *Poly*, referring to many, signifies the many uses of these operators and functions. C++ supports polymorphism both at run-time and at compile-time.
- The use of overloaded functions is an example of compile-time polymorphism. Run-time polymorphism can be achieved through the use of pointer to base class and *virtual functions.*
- Object pointers are useful in creating objects at run-time. It can be used to access the public members of an object along with an arrow operator.
- A base class pointer may address an object of its own class or an object of any class derived from the base class.
- A pure virtual function is a virtual function declared in a base class that has no definition.
- A class containing pure virtual functions cannot be used to declare any objects of its own. Such classes are called abstract class or abstract base class.



12.7 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education



12.8 ANSWERS TO CHECK YOUR PROGRESS

1. (i) (c) virtual function (ii) (a) compile-time polymorphism
   (iii) (c) both (a) and (b) (iv) (b) object pointer
   (v) (b) polymorphism
2. (i) False (ii) True (iii) True
3. (i) (c)virtual (ii) (a) run-time polymorphism
   (iii) (a)abstract class (iv) (d)pure virtual function
   (v) (c)base class, derived class
4. (i) False (ii) True (iii) True



12.9 MODEL QUESTIONS

1. What is polymorphism? What are the different types of polymprphism in C++?
2. How is polymorphism achieved
   (i) at compile time (ii) at run-time
3. What is a virtual function?
4. Describe the rules for declaring virtual functions.
5. How can C++ achieve dynamic binding?
6. What are pointer to base and derived classes?
7. Write a C++ program to demostrate the use of abstract classes.
8. Find the error in the following declaration:

```
class Base
{
    public:
    virtual void display()=0;
};
void main()
{
    Base b;
}
```

9. What are virtual and pure virtual functions? Use this concept to calculate the area of a square and a rectangle.

****

UNIT 13 : FILE HANDLING

UNIT STRUCTURE

13.1 Learning Objectives
13.2 Introduction
13.3 File Classes
13.4 Opening and Closing a File
13.5 File Pointers and their Manipulation
13.6 Functions for Input and Output Operations
13.7 Exception Handling
13.8 Let Us Sum Up
13.9 Further Readings
13.10 Answers To Check Your Progress
13.11 Model Questions

13.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- describe file classes in C++
- open and close a file
- learn about different file modes
- define file pointers and them in programming use
- perform functions for input/output operations
- use some additional file handling features

13.2 INTRODUCTION

In the previous units we have come across many useful features of the C++ language like operators, arrays, pointers, functions etc, which provide the basic programming platform for programmers. Also, we were introduced to some new Object Oriented features and applications of the C++ language like classes, operator overloading, inheritance, polymorphism etc. In this unit, we will deal with a very important feature of this language which deals with handling files and manipulating them.

Files are a means to store data in a storage device. When we have to deal with handling enormous volumes of data, we use several external storage devices like floppy disks, hard disks etc. In the same way, we can write programs to perform these file manipulation tasks by using C++ file handling features. C++ file handling provides a mechanism to store the output of a program in a file and read from a file on the disk. The file operations of C++ are very much similar to the console oriented input and output operations where a file stream acts as the interface between the program and the file.

13.3 FILE CLASSES

File processing in C++ is very similar to ordinary interactive input and output because the same kind of stream objects are used. Input from a file is managed by an `ifstream` object the same way that input from a keyboard is managed by the `istream` object `cin`. Similarly, output to a file is managed by an `ofstream` object the same way that output to the monitor or printer is managed by the `ostream` object `cout`. The only difference is that `ifstream` and `ofstream` objects have to be declared explicitly and initialized with the external name of the file which they manage. In other words, as we have been using `<iostream>` header file which provide functions `cin` and `cout` to take input from console and write output to a console respectively, we introduce one more header file `<fstream>` which provides data types or classes (ifstream, ofstream, fstream) to read from a file and write to a file.

Table 13.1 : Stream classes

Data type	Description
ofstream	This data type represents the output file stream and is used to create files and to write information to files.
ifstream	This data type represents the input file stream and is used to read information from files.
fstream	This data type represents the file stream generally, and has the capabilities of both of stream and ifstream which means that it can create files, write information to files, and read information from files.

These classes, designed to manage the disk files, are declared in **fstream** and therefore we have to **#include** the **<fstream>** header file that defines these classes in any program that uses files. These classes, designed to manage the disk files, are declared in **fstream** and therefore we have to **#include** the **<fstream>** header file that defines these classes in any program that uses files.

13.4 OPENING AND CLOSING A FILE

In C++, a file is opened by linking it to a stream. There are three types of streams: **input**, **output** and **input/output**. To open an input stream we must declare the stream to be of class **ifstream**. To open an output stream, it must be declared as class **ofstream**. A stream that will be performing both input and output operations must be declared as class **fstream**. For example, the following fragment creates one input stream, one output stream and one stream that is capable of both input and output.

```
ifstream in;

ofstream out;

fstream both;
```

Once a stream has been created, the next step is to associate a file with it, and thereafter the file is available (opened) for processing.

Opening of files can be achieved in the following two ways:

1. Using the constructor function of the stream class.
2. Using the function **open()**.

The first method is preferred when a single file is used with a stream. However, for managing multiple files with the same stream, the second method is preferred. Let us discuss each of these methods one by one.

Opening a File Using Constructors

We know that a constructor of a class initializes its object when it (the object) is created. Simillarly, constructors of the stream classes (ifstream, ofstream, or fstream) are used to initialize file stream objects with the filenames passed to them, as given below:

To open a file named myfile as an input file (i.e., data will be needed from it and no other operation like writing or modifying would take place on the file), we shall create a file stream object of input type i.e., ifstream type like:

```
ifstream fin("myfile", ios::in);
```

The above statement creates an object, **fin**, of input file stream. After creating the ifstream object **fin**, the file **myfile** is opened and attached to the input stream, **fin**.

To read from this file, this stream object will be used using the operator (“>>”) as,

13.4 OPENING AND CLOSING A FILE

In C++, a file is opened by linking it to a stream. There are three types of streams: **input**, **output** and **input/output**. To open an input stream we must declare the stream to be of class **ifstream**. To open an output stream, it must be declared as class **ofstream**. A stream that will be performing both input and output operations must be declared as class **fstream**. For example, the following fragment creates one input stream, one output stream and one stream that is capable of both input and output.

ifstream in;

ofstream out;

fstream both;

Once a stream has been created, the next step is to associate a file with it, and thereafter the file is available (opened) for processing.

Opening of files can be achieved in the following two ways:

1. Using the constructor function of the stream class.
2. Using the function **open()**.

The first method is preferred when a single file is used with a stream. However, for managing multiple files with the same stream, the second method is preferred. Let us discuss each of these methods one by one.

Opening a File Using Constructors

We know that a constructor of a class initializes its object when it (the object) is created. Simillarly, constructors of the stream classes (ifstream, ofstream, or fstream) are used to initialize file stream objects with the filenames passed to them, as given below:

To open a file named myfile as an input file (i.e., data will be needed from it and no other operation like writing or modifying would take place on the file), we shall create a file stream object of input type i.e., ifstream type like:

```
ifstream fin("myfile", ios::in);
```

The above statement creates an object, **fin**, of input file stream. After creating the ifstream object **fin**, the file **myfile** is opened and attached to the input stream, **fin**.

To read from this file, this stream object will be used using the operator **(">>") as,**

```
char ch;fin>>ch; // read a character from the
filefloat amt;fin>>amt;// read a floating-point
number form the file
```

Similarly, opening an output file (on which there is no operation except writing) can be accomplish by –

1. Creating **ofstream** object to manage the output stream.
2. Associating that object with a particular file.

```
ofstream fout("secret" ios::out);//create ofstream...
// ..object named fout
```

This would create an output stream object named **fout** and attach the file **secret** with it.

Now, to write something to it, we use the << operator like,

```
int code=2193;fout<<code<<"xyz";
```

The connections with a file are closed automatically when the input and the output stream objects expire i.e., when they go out of scope. We may close a connection with a file explicitly by using the close() method:

```
fin.close(); // close input connection to
```

```
filefout.close();// close output connection to file
```

Closing such a connection does not eliminate the stream; it just disconnects it from the file. For example, after the above statements, the streams **fin** and **fout** still exist along with the buffers they manage. We may later reconnect the stream to the same file or to another file, if needed. Closing a file flushes the buffer which means that the data remaining in the buffer (input or output stream) is moved out of it. For example, when an input file's connection is closed, the data is moved from the input buffer to the program and when an output file's connection is closed, the data is moved from the output buffer to the disk file.

Opening Files Using Open() Function

There may be situations requiring a program to open more than one file. The strategy for opening multiple files depends upon how they will be used. If the situation requires simultaneous processing of two files, then we need to create a separate stream for each file. However, if the situation demands sequential processing of files (i.e., processing them one by one), then we can open a single stream and associate it with each file in turn. To use this approach, we declare a stream object without initializing it, then use a second statement to associate the stream with a file. For example,

```
ifstream fin;//create an input stream
fin.open("Master.dat", ios::in);
//associate fin with Master.dat:
// process Master.dat
fin.close();//terminate association with Master.dat
fin.open("Tran.dat", ios::in);
//associate fin with Tran.dat:
process Tran.dat
fin.close();
// terminate association
```

The Concept of File Modes

The **filemode** describes how a file is to be used: to read from it, to write to it, to append it, and so on. When we associate a stream with a file, either by initializing a file stream object with a file name or by using the **open()** method, we can provide a second argument specifying the file mode, as mentioned below:

```
stream_object.open("filename", (filemode));
```

The second method argument of **open()**, the filemode, is of type int, and we may choose one from several constants defined in the **ios class**.

List of File Modes in C++

Following table lists the filemodes available in C++ with their meanings:

Table 14.2 : Different file modes

Constant	Meaning	Stream Type
ios :: in	Opens file for reading, i.e., in input mode.	ifstream
ios :: out	Opens file for writing, i.e., in output mode.	ofstream
ios :: ate	This also opens the file in ios::trunc mode, by default.This means an existing file is truncated when opened, i.e., its previous contents are discarded.	ofstream
ifstream	This seeks to end-of-file upon opening of the file. I/O operations can still occur anywhere within the file.	
ios :: app	This causes all output to that file to be appended to the end. This value can be used only with files capable of output	ofstream
ios :: trunc	This value causes the contents of a pre-existing file by the same name to be destroyed and truncates the file to zero length.	ofstream
ios :: nocreate	This cause the open() function to fail if the file does not already exist. It will not create a new file with that name.	ofstream
ios :: noreplace	This causes the open() function to fail if the file already exists.This is used when you want to create a new file and at the same time.	ofstream
ios :: binary	This causes a file to be opened in binary mode. By default, files are opened in text mode. When a file is opened in text mode, various character translations may take place, such as the conversion of carriage-return into newlines.However, no such character translations occur in file opened in binary mode.	ofstream ifstream

The **fstream** class does not provide a mode by default and, therefore, one must specify the mode explicitly when using an object of the **fstream** class.

Both **ios::ate** and **ios::app** place us at the end of the file just opened. The difference between the two is that the **ios::app** mode allows us to add data to the end of the file only, while the **ios::ate** mode lets us write data anywhere in the file.

We may combine two or more filemode constants using the C++ **bitwise OR** operator (symbol |). For example, the following statement:

```
ofstream fout;fout.open("Master", ios::app |
ios::nocreate);
```

will open a file in the append mode if the file exists and will abandon the file opening operation if the file does not exist.

To open a binary file, we need to specify ios :: binary along with the file mode, e.g.,

```
fout.open("Master", ios::app | ios::binary);
```

or,

```
fout.open("Main", ios::out | ios::nocreate |
ios::binary);
```

Closing a File in C++

As already mentioned, a file is closed by disconnecting it with the stream it is associated with. The **close()** function accomplishes this task and it takes the following general form:

```
stream_object.close();
```

For example, if a file Master is connected with an **ofstream** object **fout**, its connections with the stream **fout** can be terminated by the following statement:

```
fout.close();
```

C++ Opening and Closing a File Example

Here is an example for the complete understanding on:

- how to open a file in C++ ?
- how to close a file in C++ ?

Program 13.1: Open a file to store/retrieve information to/from it, and close that file after storing/retrieving the information to/from it.

```
#include<conio.h>
#include<string.h>
#include<stdio.h>
#include<fstream.h>
#include<stdlib.h>
void main()
{
       ofstream fout;
       ifstream fin;
       char fname[20];
       char rec[80], ch;
       clrscr();
       cout<<"Enter file name: ";
       cin.get(fname, 20);
       fout.open(fname, ios::out);
       if(!fout)
       {
           cout<<"Error in opening the file "<<fname;
           getch();
           exit(1);
       }
       cin.get(ch);
       cout<<"\nEnter a line to store in the file:\n";
       cin.get(rec, 80);
       fout<<rec<<"\n";
       cout<<"\nThe line is stored successfully.";
       cout<<"\nPress any key to see...\n";
       getch();
       fout.close();
       fin.open(fname, ios::in);
       if(!fin)
       {
       cout<<"Error in opening the file "<<fname;
       cout<<"\nPress any key to exit...";
       getch();
       exit(2);
}
cin.get(ch);
fin.get(rec, 80);
cout<<"\nThe file contains:\n";
cout<<rec;
cout<<"\n\nPress any key to exit...\n";
fin.close();
getch();
}
```



CHECK YOUR PROGRESS

1. Fill in the blanks

(a) Output to a file is managed by an _____ object.

(b) In C++, a file is opened by linking it to a _____.

(c) A stream that will be performing both _____ and _____ operations must be declared as class _____.

(d) The _____ with a file are closed automatically when the input and the output _____ objects expire.

(e) The _____ operator is used to combine two or more filemode constants.

13.5 FILE POINTERS AND THEIR MANIPULATION

The C++ input and output system manages two integer values associated with a file.

These are:

- **get pointer** – specifies the location in a file where the next read operation will occur.
- **put pointer** – specifies the location in a file where the next write operation will occur. In other words, these pointers indicate the current positions for read and write operations, respectively. Each time an input or an output operation takes place, the pointers are automatically advanced sequentially.

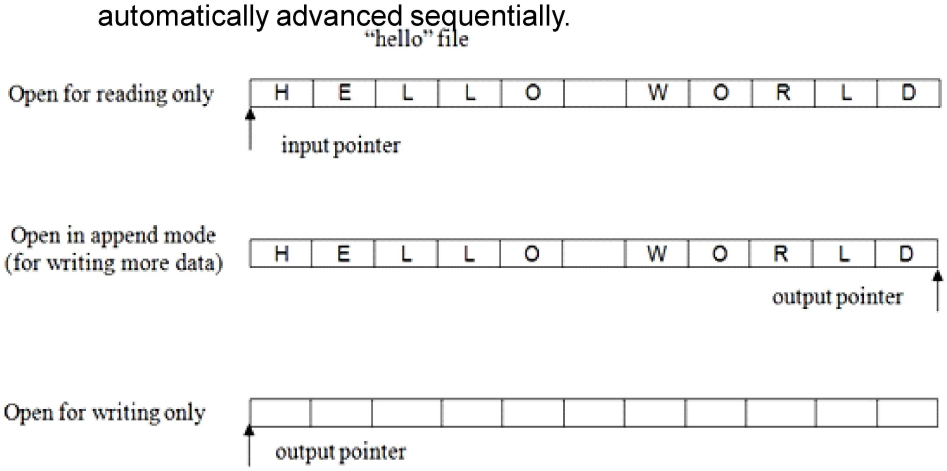


Fig 13.1: Action on file pointers while opening a file

Functions for manipulation of file pointers

The read operation from a file involves the **get** pointer. It points to a specific location in the file and the reading starts from that location. Then, the **get** pointer keeps moving forward which lets us read the entire file. Similarly, we can start writing to a location where **put** pointer is currently pointing. The **get** and **put** are known as file position pointers and these pointers can be manipulated or repositioned to allow random access of the file. The functions which manipulate file pointers are shown in Table 13.1:

Table 13.3 : File pointer

Function	Description
seekg()	Moves the **get** pointer to a specific location in the file
seekp()	Moves the **put** pointer to a specific location in the file
tellg()	Returns the current position of the **get** pointer
tellp()	Returns the current position of the **put** pointer

seekg()

Sets the position of the get pointer. The **get** pointer determines the next location to be read in the source associated to the stream. The function **`seekg(n, ref_pos)`** takes two arguments:

1) **`n`** denotes the number of bytes to move and **`ref_pos`** denotes the reference position relative to which the pointer moves.

2) **`ref_pos`** can take one of the three constants:

- **`ios:: beg`** moves the get pointer `n` bytes from the beginning of the file,
- **`ios:: end`** moves the get pointer `n` bytes from the end of the file
- **`ios:: cur`** moves the get pointer `n` bytes from the current position. If we don't specify the second argument, then **`ios:: beg`** is the default reference position.

Program 13.2: Read a file into memory

```
#include<fstream.h>
#include<iostream.h>
int main (int argc, char** argv)
{
     fstream myFile("test.txt", ios::in | ios::out |
       ios::trunc);
     myFile << "Hello World";
     myFile.seekg(6, ios::beg);
     char buffer[6];
     myFile.read(buffer, 5);
     buffer[5] = 0;
     cout << buffer << endl;
     myFile.close();
}
```

In the above example, we open a new file for input/output discarding any current content in the file. Adding the characters “Hello World” to the file, we seek to read 6 characters from the beginning of the file. Then we read the next 5 characters from the file into a buffer and end the buffer with a null terminating character. Finally, we output the contents read from the file and close it.

seekp()

The behaviour of `seekp(n, ref_pos)` is same as that of `seekg()`.The `seekp` method changes the location of a stream object’s file pointer for output (put or write.) In most cases, `seekp` also changes the location of a stream object’s file pointer for input (get or read).

Program 13.3: Read a file into memory

```
#include <fstream.h>
int main()
{
 long pos;
 ofstream outfile;
 outfile.open("test.txt");
 outfile.write("This is an apple",16);
 pos=outfile.tellp();
 outfile.seekp(pos-7);
 outfile.write(" sam",4);
 outfile.close();
 return 0;
}
```

In this example, `seekp` is used to move the **put pointer** back to a position 7 characters before the end of the first output operation.

The final content of the file shall be:

```
This is a sample
```

tellg()

The tellg() function is used with input streams, and returns the current **get** position of the pointer in the stream.

Syntax: pos_type tellg();

It has no parameters and return a value of the member type pos_type, which is an integer data type representing the current position of the get stream pointer.

tellp()

Returns the absolute position of the **put** pointer. The put pointer determines the location in the output sequence where the next output operation is going to take place.

Syntax: pos_type tellp();

The tellp() function is used with output streams, and returns the current **put** position of the pointer in the stream. It has no parameters and return a value of the member type pos_type, which is an integer data type representing the current position of the put stream pointer.

Program 13.4: To demonstrate example of tellg() and tellp() function

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
     fstream file;
     //open file sample.txt in and Write mode
      file.open("sample.txt",ios::out);
     if(!file)
     {
          cout<<"Error in creating file!!!";
         return 0;
     }
     //write A to Z
file<<"ABCDEFGHIJKLMNOPQRSTUVWXYZ";
     //print the position
    cout<<"Current position is: "<<file.tellp()<<endl;
     file.close();
     //again open file in read mode
```

```
     file.open("sample.txt",ios::in);
    if(!file)
    {
         cout<<"Error in opening file!!!";
        return 0;
    }
    cout<<"After opening file position is:"
     <<file.tellg()<<endl;
    //read characters until end of file is not found
    char ch;
    while(!file.eof())
    {
           cout<<"At position: "<<file.tellg();
           //current position
           file>>ch;   //read character from file
           cout<<" Character \""<<ch<<"\""<<endl;
    }
    //close the file
    file.close();
    return 0;
}
```

The tellg() and tellp() functions can be used to find out the current position of the get and put file pointers respectively in a file.

13.6 FUNCTIONS FOR INPUT AND OUTPUT OPERATIONS

We have seen the use of **cin** and **cout** which are the I/O operators that give us formatting control over the input and output, but these are not character I/O functions. We have also seen the file stream classes that support a number of member functions for performing the input and output operations on files. Some functions like **put()** and **get()** are designed for handling a single character at a time, and some like **write()** and **read()** are designed for writing and reading blocks of binary data.

The functions, put() and get(), which allow reading/writing character by character are called **character I/O functions.**

put()

put() function sends one character at a time to the output stream, where an output stream can be a standard output stream object or user-defined output stream object

Program 13.5:

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<fstream.h>
void main()
{
      clrscr();
      char string[50];
      cout<<"\n Enter a string to write in a file ";
      gets(string);
      fstream FILE;
      FILE.open("MYTEXT.TXT",ios::app);
      for(int i=0;string[i]!='\0';i++)
      {
            FILE.put(string[i]);
      }
      FILE.close();
      getch();
}
```

In the above program **put()** function is used with user-defined output stream object "FILE" which represents a disk file "MYTEXT.TXT".

get()

The **get()** inputs a single character from the standard input device (by default it is keyboard). ***Syntax***: device.get(char_variable); The device can be any standard input device. If we want to get input from a keyboard then we should use **cin** as the device. Because, most of the computers consider the keyboard as the standard input device, **stdin.**

```
char ch;

cin.get(ch);
```

The **get()** function is a buffered input function. When we type in, data does not go into our program unless we hit the Enter key. **get()** function receives one character, including white space, at a time from the input stream. An input stream can be a standard input stream object or user defined stream object of the istream class.

Program 13.6:

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<fstream.h>
#include<ctype.h>
void main()
```

```
{
      ifstream infile;
      infile.open("PARA.TXT");
      char ch;
      int count=0;
      while(infile)
      {
            infile.get(ch);
            if(isdigit(ch))
            count++;
      }
      infile.close();
      cout<<"\n Total digits = "<<count;
}
```

Another way to read and write blocks of binary data is to use C++'s **read()** and **write()** functions. Their prototypes are:

istream &read(char **buf*, streamsize *num*);

ostream &write(const char **buf*, streamsize *num*);

The **read()** function reads ***num*** characters from the invoking stream and puts them in the buffer pointed to by ***buf***. The **write()** function writes ***num*** characters to the invoking stream from the buffer pointed to by ***buf***

Program 13.7: Writing data using write()

```
#include<fstream.h>
#include<conio.h>
class Student
{
       int roll;
       char name[25];
       float marks;
       void getdata()
       {
              cout<<"\n\nEnter Roll : ";
              cin>>roll;
              cout<<"\nEnter Name : ";
              cin>>name;
              cout<<"\nEnter Marks : ";
              cin>>marks;
       }
       public:
       void AddRecord()
        {
              fstream f;
              Student Stu;
```

```
 f.open("Student.dat",ios::app|ios::binary);
              Stu.getdata();
              f.write((char *) &Stu, sizeof(Stu));
              f.close();
       }
};
void main()
{
       Student S;
       char ch='n';
       do
       {
              S.AddRecord();
              cout<<"\nAny more data(y/n): ";
              ch = getche();
       }
        while(ch=='y' || ch=='Y');
        cout<<"\nData written successfully...";
}
```

Program 13.8: Reading data using read()

```
#include<fstream.h>
#include<conio.h>
class Student
{
  int roll;
  char name[25];
  float marks;
  void putdata()
  {
cout<<"\n\t"<<roll<<"\t"<<name<<"\t"<<marks;    }
public:
void Display()
{
fstream f;
StudentStu;
f.open("Student.dat",ios::in|ios::binary);
cout<<"\n\tRoll\tName\tMarks\n";
while((f.read((char*)&Stu,sizeof(Stu)))!=NULL)
Stu.putdata();
f.close();
}
};
void main()
{  Student S;
  S.Display();
}
```

13.7 EXCEPTION HANDLING

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero. Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch:** A program catches an exception with an exception handler at the place in a program where we want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It is followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
   // protected code
}catch(Exception-Name  e1)
{
   // catch block
}catch(Exception-Name  e2)
{
   // catch block
}catch(Exception-Name  eN)
{
   // catch block
}
```

We may list down multiple **catch** statements to catch different type of exceptions in case our **try** block raises more than one exception in different situations.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using throw

statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.
Following is an example of throwing an exception when dividing by zero condition occurs

```
double division(int a, int b)
{
   if(b==0)
   {
       throw "Division by zero condition!";
   }
   return(a/b);
}
```

Catching Exceptions

The **catch** block following the **try** block catches any exception. We can specify what type of exception we want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {
   // protected code
}catch(ExceptionName e) {
   // code to handle ExceptionName exception
}
```

The above code will catch an exception of ExceptionName type. If we want to specify a catch block handling any type of exception that is thrown in a try block, we must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows

```
try {
   // protected code
}catch(...) {
   // code to handle any exception
}
```

Program 13.9:

```
#include <iostream>
using namespace std;
int main()
{
cout<<"Start/n";
try {                    // start a try block
cout<<"Inside Try block\n";
throw 100;   //Throw an error
```

```
cout<<"This will not execute";
}
catch(int i)              //catch an error
{
cout<<"Caught an Exception. Value is"<<i;
}
return 0;
}
```

Because we are raising an exception of type **const char***, so while catching this exception, we have to use const char* in catch block.

Detecting EOF

The physical contents of a file may not be precisely known. C++ provides a special function, **eof()**, that returns nonzero (TRUE) when there are no more data to be read from an input file stream, and zero (FALSE) otherwise. Returns true if the eofbit ***error state flag*** is set for the stream. This flag is set by all standard input operations when the End-of-File is reached in the sequence associated with the stream. Note that the value returned by this function depends on the last operation performed on the stream (and not on the next). Operations that attempt to read at the ***End-of-File*** fail, and thus both the **eofbit** and the **failbit** end up set. This function can be used to check whether the failure is due to reaching the *End-of-File* or to some other reason.

Program 13.10

```
#include <iostream>
using namespace std;
int main()
{
  int a, b;
  cout<<"Enter Values of a and b \n";
  cin>>a;
  cin>>b
  int x=a-b;
  try
  {
   if(x!=0)
    {
      cout<<"Result(a/x)="<<a/x<<"\n";
    }
   else
    {
     throw(x);
    }
  }
```

```
   catch(int i)
    {
      cout<<"Exception caught: x="<<x<<"\n";
    }
  cout<<"END";
return 0;
}
```



CHECK YOUR PROGRESS

2. Fill in the blanks
   (a) The get and put are known as file _____ pointers.
   (b) The tellg() function is used with _____ streams, and returns the current ____ position of the pointer in the stream.
   (c) ____ function sends one character at a time to the output stream.
   (d) The get() function is a _____ input function.
   (e) _____ provide a way to transfer control from one part of a program to another.
   (f) ____ returns nonzero when there are no more data to be read from an input file stream.
3. State TRUE or FALSE
   (a) get pointer specifies the location in a file where the next write operation will occur
   (b) seekg() sets the position of the get pointer.
   (c) The tellp() function is used with output streams, and returns the current put position of the pointer in the stream
   (d) The get() inputs a single character from the standard input device
   (e) Exceptions can be thrown anywhere within a code block using try statements
   (f) The catch block following the throw block catches any exception



13.8 LET US SUM UP

- C++ file handling provides a mechanism to store the output of a program in a file.
- ofstream, ifstream and fstream classes are designed to manage disk files.

- There are three types of streams: input, output and input/output.
- Constructors of the stream classes are used to initialize file stream objects.
- A file is closed by disconnecting it with the stream it is associated with.
- The C++ input and output system manages two integer values associated with a file.
- The seekp() method changes the location of a stream object's file pointer for output.
- The functions, put() and get(), allow reading/writing character by character.
- The read() and write() functions read and write blocks of binary data.



13.9 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education



13.10 ANSWERS TO CHECK YOUR PROGRESS

1

(a) ofstream.
(b) stream.
(c) input, output, fstream.
(d) connections, stream.
(e) bitwise OR.

2.

(a) position.
(b) Input, get.
(c) put().
(d) buffered.
(e) Exceptions.
(f) eof().

3.

(a) False
(b) True
(c) True
(d) True
(e) False
(f) False



13.11 MODEL QUESTIONS

1. What are Files classes in C++? Illustrate their use with examples.
2. Describe opening a file in C++ with an example.
3. Name and describe the different file modes in C++.
4. What are file pointers in C++? How are they used?
5. Illustrate the usage of seekg(), seekp(), tellg() and tellp() functions with the help of examples.
6. Differentiate between the put() and get() functions.
7. Why are read() and write() functions used.
8. What is exception handling in C++. Illustrate its use with an example.
9. How do we throw and catch an exception in C++?
10. Why is it necessary to detect EOF in C++ programs? Illustrate.

UNIT 14 – TEMPLATES

UNIT STRUCTURE

14.1 Learning Objectives

14.2 Introduction

14.3 Template s

14.4 Function Template

14.5 Function Template Overloading

14.6 Class Template

14.7 Default Template Arguments

14.8 Let Us Sum Up

14.9 Answers to Check Your Progress

14.10 Further Reading

14.11 Model Questions

14.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- define templates
- use function templates in programs
- describe how function template overloading works
- use class templates in programs
- describe template arguments

14.2 INTRODUCTION

In the previous unit, we have learnt about file classes and file modes. We have also learnt how to open and close a file, write into a file and read data from a file using functions like read(), write(), get() and put(). Manipulation of file pointers and getting their location by using functions like seekp(), seekg(), tellp() and tellg() were also included in the previous unit.

In this unit we will learn about templates. We will learn in detail about both function templates and class templates in detail. Function template overloading and default template arguments are also covered in this unit. We will learn about the object oriented design, its approaches, techniques and different models in detail in the next unit.

14.3 TEMPLATES

Templates are one of the most powerful features of C++. It is possible to create generic functions and classes using templates. In a generic function or class created using templates, the type of data upon which the function or the class operates is specified as a parameter. So by using templates, a general procedure can be applied to different types of data. For example, the logic behind a sorting function or a printing function is the same whether we use integers or strings or floating point numbers. Only the type of data being sorted or printed will be different. By using a template, we define the nature of the algorithm independent of the data. Once defined, the compiler will automatically generate the correct code for the data type that we have used during the execution of the program. So, basically, we are creating a function or class that can work using different types of data.

14.4 FUNCTION TEMPLATES

A function template is created using the keyword “template”. The meaning of the word template in real world accurately describes its meaning in C++ also. It is used to create a template (or framework) that describes what the function is supposed to do. The compiler will then fill out the missing details using the given parameter values during run time. The general syntax of a function template is given below:

```
template <typename T> return_type function_name (  T
parameter 1,     T parameter 2, ...)
      {
            // body of function
      }
```

Here, "***typename***" is the keyword and '*T*' is the template argument that accepts different data type to be used by the function. This template argument will be replaced with the actual data type by the compiler automatically when it creates a version of the function. In place of the "*typename*" keyword, we can also use the keyword "***class***".

The function clause of the function template definition does not have to be in the same line as the template definition. Another way to represent the syntax for function template is shown below:

```
template <class T>
T function_name (T parameter1, T parameter 2,
...)
{
      // body of function
}
```

Now, let us look at an example to understand the concept of function templates more clearly.

Program 14.1: Program showing function template

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

template<class T>
T sum(T number_1, T number_2){
      return (number_1 + number_2);
}

int main () {

      clrscr();
      //calling sum function template with integer
paramenters
      int result_1 = sum(1, 3);
      cout << "Sum of 1 and 3 is: " << result_1 <<
endl;

      //calling sum function template with double
paramenters
      double result_2 = sum(1.3, 3.5);
      cout << "Sum of 1.3 and 3.5 is: " << result_2
<< endl;
```

```
      getch();
      return 0;
}
```

Output:

In program 14.1, we have defined a template where we have a sum() function. The sum()function takes any two numerical values and displays their summation result. In the main() function, the sum() function is called twice. For the first time, we have passed integer values and for the second time floating point double values are passed both using the same function sum(). This is possible because we have used a template where the complier fills the missing data type at run time.

We can use more than one data type in the template statement by using a comma separated list. For example, the program 14.2 creates a template function that can have three data types.

Program 14.2: Program showing use of multiple data types in a function template

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

template <class T1, class T2, class T3>
void mprint( T1 a, T2 b, T3 c)
{
      cout << a << endl;
      cout << b << endl;
      cout << c << endl;
}

int main ()
{
      clrscr();
            //calling mprint function using multiple
data types
      mprint(5, 2.3, "KKHSOU");
```

```
      getch();
      return 0;
}
```

Output:

14.5 FUNCTION TEMPLATE OVERLOADING

As we have seen in the programs above, a generic function overloads itself as needed. But we can explicitly also overload it too. This is called explicit specialization. When we overload a generic function, that overloaded function overrides the generic function relative to that specific version.

Program 14.3: Program showing explicit overloading of a template function

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

template <class T>
T sum(T number_1, T number_2){
      cout << "Using Template" << endl;
      return (number_1 + number_2);
}

int sum(int number_1, int number_2) {
      cout << "Using Function" << endl;
      return (number_1 + number_2);
}

int main () {

      clrscr();
      //calling sum function template with integer
paramenters
      int result_1 = sum(1, 3);
      cout << "Sum of 1 and 3 is: " << result_1 <<
endl;

      //calling sum function template with double
paramenters
      double result_2 = sum(1.3, 3.5);
      cout << "Sum of 1.3 and 3.5 is: " << result_2
<< endl;
```

```
      getch();
      return 0;
}
```

Output:

As can be seen from the output of the program 14.3, when sum(1, 3) is called, it invokes the explicitly overloaded version of sum() function defined in the program. So, the compiler does not generate this version of the generic sum() function. However, when sum(1.3, 3.5) is called, then the generic sum() function defined int he template is called.

Just like we can create explicit overloaded generic function, we can also overload the template specification itself too. We can do this by creating another version of the template that differs from any others in its parameter list. An example for overloading of a function template that accepts one or more parameters is shown in program 14.4.

Program 14.4: Program showing overloading of a function template

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

// first version of template
template <class T>
void mprint(T number_1){
      cout << "Using Template version 1" << endl;
}

// second version of template
template <class T, class S>
void mprint(T number_1, S number_2){
      cout << "Using Template version 2" << endl;

}


int main () {

      clrscr();

      //calls mprint(T)
      mprint(10);
```

```
      //calls  mprint(T,S)
      mprint(10,"KKH");

      getch();
      return 0;
}
```

Output:

In the above program, the template for mprint() function is overloaded to accept either one or two parameters.



CHECK YOUR PROGRESS

Q1. What is a template?

.....

.....

.....

Q2. Give the syntax for declaring a function template.

.....

.....

.....

14.6 CLASS TEMPLATES

As we have seen in the earlier sections, just like we have created generic functions we can also create generic class. Generic classes are used when the algorithm used in the class can be generalized. For example, the same algorithm used in a queue of integers will also work for a queue of floating point numbers. When we create a generic class, it can perform the operations we define for any type of data. The compiler will then automatically generate the type of object depending upon the type we specify when the object is created.

A class template is created using the keywords "template" and "class". The general syntax of a class template is given below:

```
template <class T> class class_name {
            // body of class
```

```
}
```

Here, '*T*' is the type name which will be specified when a class is instantiated. Another way to represent the syntax for class template is shown below:

```
template <class T>
class class_name {
        // body of class
};
```

We can create a specific instance of the class by using the following general form:

```
class_name<type> ob;
```

Here, 'type' is the type name of the data that the class will be operating upon. The member functions of a class are automatically generic and we do not need to use template to explicitly specify these member functions.

Program 14.5: Program showing class template

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

template <class T>
class PrintNumber {
    private:
        T number;
    public:
        PrintNumber(T num) {
            number = num;
        }

        T GetNumber() {
            return number;
        }
```

```
};

int main () {

      clrscr();
            //creating object with double datatype
      PrintNumber<int>  IntNumber(3);
            //printing number with integer datatype
      cout << "Printing Integer Number: " <<
IntNumber.GetNumber() << endl;

            //creating object with double datatype
      PrintNumber<double>  DoubleNumber(4.7);
            //printing number with double datatype
      cout << "Printing Double Number: " <<
DoubleNumber.GetNumber() << endl;

      getch();
      return 0;
}
```

Output:

From the program 14.5, we can see that the declaration of a generic class is similar to that of a generic function. It is not until an object is created of the class "PrintNumber" is declared that the actual data type is determined. When a specific instance of "PrintNumber" is declared, the complier automatically generates all the functions and variables associated with that class. In this example, two different types of objects are declared of type "integer" and "double".

14.7 DEFAULT TEMPLATE ARGUMENTS

Default arguments can be associated with a template class. For example, in the code below an "integer" type is associated as a default argument with a template class.

```
template <class T = int> class class_name {
            // body of class
      }
```

Here, '*T*' is the type name which will be specified when a class is instantiated and the default value for 'T' is 'int' when no value is specified. The default value is used when no explicit value is specified when the class is instantiated.



CHECK YOUR PROGRESS

Q3. How are class templates created?

.....

.....

.....

Q4. How can we create a specific instance of a class object?

.....

.....

.....



14.8 LET US SUM UP

- Templates are one of the most powerful features of C++.
- It is possible to create generic functions and classes using templates.
- In a generic function or class created using templates, the type of

data upon which the function or the class operates is specified as a parameter.

- By using a template, we define the nature of the algorithm independent of the data. Once defined, the compiler will automatically generate the correct code for the data type that we have used during the execution of the program.
- A function template is created using the keyword “template” and “typeame” or “class”.
- When we overload a generic function, that overloaded function overrides the generic function relative to that specific version.
- Just like we can create explicit overloaded generic function, we can also overload the template specification itself too. We can do this by creating another version of the template that differs from any others in its parameter list.
- Generic classes are used when the algorithm used in the class can be generalized.
- When we create a generic class, it can perform the operations we define for any type of data. The compiler will then automatically generate the type of object depending upon the type we specify when the object is created.
- A class template is created using the keywords “template” and “class”.
- Default arguments can be associated with a template class.



14.10 FURTHER READING

1) Schildt, Herbert. C++: The complete reference. New York: McGraw-Hill, 2003.
2) Stroustrup, Bjarne. The C++ programming language. Pearson Education India, 2000.



14.9 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1: Templates are one of the most powerful features of C++. By using a template, we can define the nature of the algorithm independent of the data. Once defined, the compiler will automatically generate the correct code for the data type that we have used during the execution of the program.

Answer to Q2: The general syntax of a function template is given below:

```
template <class T>
return_type function_name (T parameter 1, T parameter 2, ...)
      {
            // body of function
      }
```

Here, "***class"*** is the keyword and '*T*' is the template argument that accepts different data type to be used by the function. This template argument will be replaced with the actual data type by the compiler automatically when it creates a version of the function.

Answer to Q3: A class template is created using the keywords "template" and "class". The general syntax of a class template is given below:

```
template <class T> class class_name {
            // body of class
      }
```

where, '*T*' is the type name which will be specified when a class is instantiated.

Answer to Q4: We can create a specific instance of the class by using the following general form:

```
class_name<type> ob;
```

where, 'type' is the type name of the data that the class will be operating upon.



14.11 MODEL QUESTIONS

Q1. Define template.

Q2. What are function templates and class templates?

Q3. How are function templates declared? Explain using an example.

Q4. How are class templates declared? Explain using an example.

Q5. Show using a program how function template overloading is done.

Q6. Define a function template that swaps two numeric data types.

Q7. Write a sorting function using generic type so that the function can sort any array of numeric type (int, double, float and long).

Q8. Write a linear search function using generic type so that the function can sort any array of numeric type (int, double, float and long).

****