UNIT 12 : VIRTUAL FUNCTIONS AND POLYMORPHISM

UNIT STRUCTURE

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***

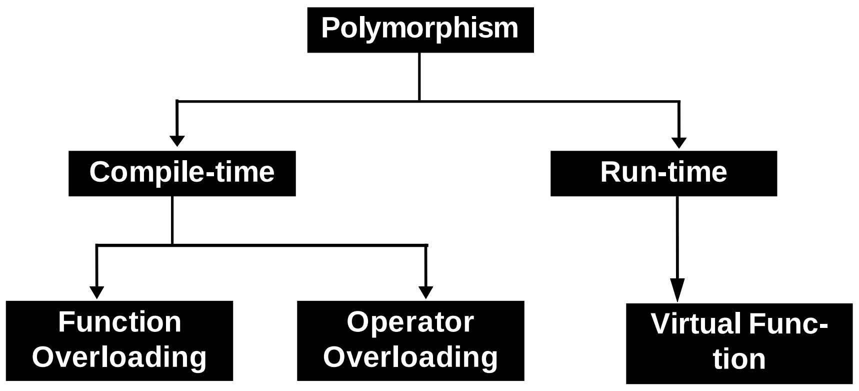


Fig. 12.1: Polymorphism in C++

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 byan 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 func-
tion
    };
    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

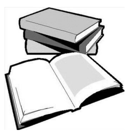
3. Choose the appropriate option for the corect answer:
   (i) Dynamic binding is done using the keyword
   (a) static (b) dynamic
   (c) virtual (d) abstract
   (ii) Virtual function helps us in achieving
   (a) run-time polymorphism
   (b) compile-time polymorphism
   (c) both (a) and (b) (d) none 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

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 ofstream 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. 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
ios :: ate ifstream	This seeks to end-of-file upon opening of the file. I/O operations can still occur anywhere within the file.	ofstream
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.

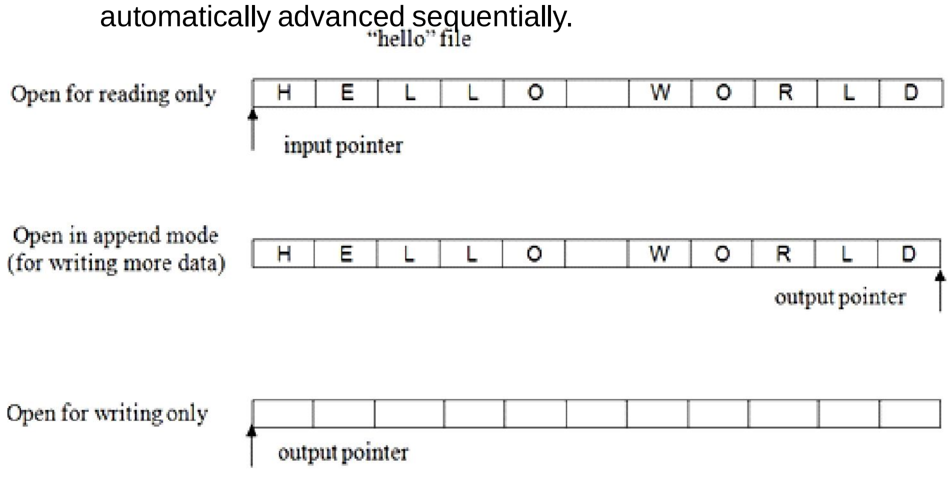


Fig 13.1: Action on file pointers while opening a file

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.

****