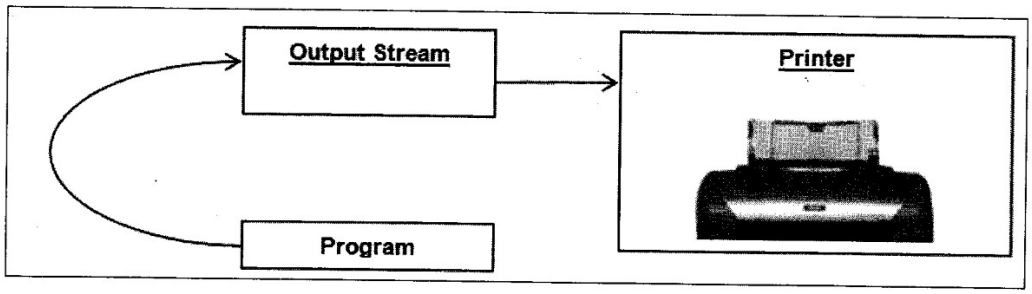

Figure 11.1 (e): Output Stream currently attached to a Printer

C++ library provides prefabricated classes for data streaming activities. In C++, the file stream classes are designed with the idea that a file should simply be viewed as a stream or array or sequence of bytes. Often the array representing a data file is indexed from 0 to len-1, where len is the total number of bytes in the entire file.

A file normally remains in a closed state on a secondary storage device until explicitly opened by a program. A file in open state file has two pointers associated with it:

1. ***Read pointer:*** This pointer always points to the next character in the file which will be read if a read command is issued next. After execution of each read command the read pointer moves to point to the next character in the file being read.
2. ***Write pointer:*** The write pointer indicates the position in the opened file where next character being written will go. After execution of each write command the write pointer moves to the next location in the file being written.

These two file positions are independent, and either one can point anywhere at all in the file. To elucidate I/O streaming in C++ let us write a small program.

```
#include <fstream.h>
int main()
{
    ofstream filename("c:\cppio.dat");
    filename << "This text will be saved in the file named cppio.dat in C:\";
    filename.close();
    return 0;
}
```

When you run this program the text This text will be saved in the file named cppio.dat in C:\ will be saved in a file named cppio.dat in the root directory of the C: drive. Let us run through each line of this program.

```
#include <fstream.h>
```

C++ file stream classes and functions have been defined in this file. Therefore, you must include this header file in the beginning of your C++ program so that these classes may be accessed.

```
ofstream filename ("c:\cppio.dat");
```

This statement creates an object of class ofstream (acronym for output file stream) named filename. This object acts as the output stream to write data in the specified file. Cppio.dat is the name of the

data file in which the program will write its output. If this file does not exists in the specified directory, it is created there.

Note that ofstream is a class. So, ofstream filename("c:\cppio.dat"); creates an object from this class. Here the file name c:\cppio.dat is being passed to the constructor of this class. In short we create an object from class ofstream, and we pass the name of the file we want to create, as an argument to the class' constructor. There are other things, too, that can be passed to the constructor. More about this will be discussed later.

filename << "This text will be saved in the file named cppio.dat in C:\";

The << operator is a predefined operator. This line puts the text in the file. As mentioned before, filename is a handle to the opened file stream. So, we write the handle name, << and after it we write the text in inverted commas. If we want to pass variables instead of text in inverted commas, just pass it as a regular use of the cout << as,

```
filename << variablename;
filename.close();
```

Having finished with writing into the file the stream must be closed. Filename is an object of class ofstream, and this class has a function close( ) that closes the stream. Just write the name of the stream object, dot and close( ), in order to close the file stream. Note that once you close the file, you can't access it anymore, until you reopen it.

Before we take up the subject any further let us quickly go through a program that reads from a data file and presents the same on the monitor. Here is the program listing.

```
#include <fstream.h>
void main() //the program starts here
{
    ifstream filename("c:\cppio.dat");
    char ch;
    while(!filename.eof())
    {
       filename.get(ch);
       cout << ch;
    }
    filename.close();
}
```

Let us run through this program line by line quickly to catch some salient features.

```
ifstream filename("c:\cppio.dat")
```

Similar to ofstream, ifstream is the input stream for a file. The term input and output have been used with respect to the program. What goes to the program from outside is the input while processed data coming out of the program is the output. We here create an input file stream by the name – filename – to handle the input file stream. The parameter passed to the constructor is the file we wish to read into the program.

char ch;

This statement declares a variable of type char to hold one character at a time while reading from the file. In this program, we intend to read one character at a time.

```
while(!filename.eof())
```

The class ifstream has a member function eof( ) that returns a nonzero value if the end of the file has been reached. This value indicates that there are no more characters in the file to be read further. This function is therefore used in the while loop for stopping condition. The file is read a character at a time till the last character has been successfully read into the program when the while loop terminates.

```
filename.get(ch);
```

Another member function of the class ifstream is get( ) which returns the next character to be read from the stream followed by moving the character pointer to the next character in the stream.

```
cout << ch;
```

The character read in the variable ch is streamed to the standard output device designated by cout (for console output, i.e., monitor) in this statement.

```
filename.close();
```

Since we reach at this statement when all the characters have been read and processed in the while loop, we are done with the file and hence it is duly closed.

11.3 OPENING AND CLOSING A FILE

In C++ to open a file, you must-first obtain a stream. There are three types of streams - input, output, and input/output. To create an input stream, you must declare the stream to be of class *ifstream*. To create an output stream, you must declare it as class *ofstream*. Streams that will be performing both input and output operations must be declared as class *fstream*. Once a stream has been created, 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 two ways:

i. Using the constructor function of the stream class.

ii. 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.

11.3.1 Opening Files using Constructors

We know that a constructor of a class initializes an object of its class when it (the object) is being created. Same way, the constructors of stream classes (*ifstream*, *ofstream*, or *fstream*) are used to initialize file stream objects with the *filenames* passed to them. This is carried out as shown below:

To open a file, *Datafile,* as an *input* file *(i.e.,* data will be read 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:

```
ifstream inpt_file ("DataFile") ;
```

The above statement creates an object (inpt_file) of input file stream. The object name is a user-defined name *(i.e.,* any valid identifier name can be given). After creating the *ifstream* object inpt_file, the file *DataFile* is opened and attached *to* the input stream inpt_file. Now both, the data being read from *DataFile* has been channelized through the input stream object.

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

```
char ch;
Inpt_file >> ch;  //read a character from the file
float amt;
inp_file >> amt;  //read a floating-pt. number from the file
```

Similarly, when you want a program to write a file i.e., to open an *output* file (on which no operation can take place except writing only). This will be accomplished by

(i) creating of stream object to manage the output stream,

(ii) associating that object with a particular file.

For instance,

```
ofstream outpt_file("secret"); // create of stream object named as outpt_file
```

This would create an output stream object named as outpt_file and attach the file *secret* with it. Opening a file for output using stream class constructor creates a new file if there is no file of that name on the disk. However, if the file by that name exists already, the act of opening it for output *scraps it off so that output starts with a fresh file.*

The connections with a file are closed automatically when the input and output stream objects expire *i.e.,* when they go out of scope. (For instance, a global object expires when the program terminates). Also, you can close a connection with a file explicitly by using the close( ) method:

```
inpt_fille.close( ) ;          // close input connection to file
outpt_file.close( );           // close output connection to file  .
```

Closing such a connection does not eliminate the stream; it just disconnects it from the file. The stream still remains there. For instance, after the above statements, the streams inpt_file and outpt_file still exist along with the buffers they manage. You can reconnect the stream to the same file or to another file, if required. Closing a file flushes the buffer which means the data remaining in the buffer (input or output stream) is moved out of it in the direction it is ought to be. For instance, 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. Following example program shows how a single, file can be created by getting input from the user.

Consider the following program that creates a single file and then displays its contents.

```
#include<fstream.h>
#include<conio.h>
int main( )
{
clrscr( );
```

```
ofstream fout("student");
char name[30] ch;
float marks=0.0;
//Loop to get 5 records
for(int i=0; i<5; i++)
{
    cout<<"Student" < < (i+ 1) <<" :\tName: ";
    cin.get(name,30);
    cout << "\t\tMarks : ";
    cin>>marks;
    cin.get(ch)    //to empty input buffer
    //write to the file
    fout<<name<< '\n' <<marks<< '\n';
}
    fout.close( );       //disconnect student file from rout
ifstream fin("student");      //connect student file to Input stream fin
fin.seekg(0);     //To bring file pointer at the file beginning
cout<< "\n";
for(i =0;i <5;i++)       //Display records
{
    fin.get(name,30);    //read name from file student
    fin.get(ch);
    fin>> marks;
    fin.get(ch);         //read marks from file student
    cout<< "Student Name: "<<name;
    cout<<"\tMarks: "<< marks << "\n";
}
    fin.close( ); // disconnect student file from fin stream
    return 0;
}
```

On execution, if the input is given as shown below:

Student 1 : Name : Awad Kishor

Marks : 56

Student 2 : Name : Mamata Sharma

Marks : 67

Student 3 : Name : Srawan Kant

Marks : 98

Student 4 : Name : Navin Gupta

Marks : 46

Student 5 : Name : Sheela Anand

Marks : 67

Then the output produced by the above program will be as shown below:

Student Name : Awad Kishor Marks : 56

Student Name : Mamata Sharma Marks : 67

Student Name : Srawan Kant Marks : 98

Student Name : Navin Gupta Marks : 46

Student Name : Sheela Anand Marks : 67

11.3.2 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 you 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 you can, open a single stream and associate it with each file in turn. To use this approach, declare a stream object without initializing it, then use a second statement to associate the stream with a file. For instance,

```
ifstream filin ;                // create an input stream
filin.open ("Master.dat") ;     // associate filin stream with file Master.dat
                          :     // process Master.dat
filin.close();                  // terminate association with Master.dat
filin.open ("Tran.dat")         // associate filin stream with file Tran.dat
                          :      II process Tran.dat
filin.close( ) ;                // terminate association
```

The above code lets you handle reading two files in succession. Note that the first file is closed before opening the second one. This is necessary because a stream can be connected to only one file at a time. Consider the following example program that prints the contents of two files in succession.

Consider the following program to use input and output files in succession.

```
#include<fstream.h>
#include<conio.h>
int main()
{
clrscr();
ofstream filout;                // create output stream
filout.open("stunames");        //connect stunames to it
filout<< "Amod Kumar \n" << "Suparna Mitra\n"
```

```
<< "Neeraj Gakhar \n";
filout.close();                 //disconnect stunames and
filout.open("stumarks");         //connect stumarks
filout<< "56.78\n" << "67.98 \n" << "98.00\n";
filout.close( );                //disconnect stumarks
char line[80];
ifstream filin;                 //create input stream
filin.open("stunames");         //connect stunames to it
cout<< "The contents of stunames file are given below \n";
filin.getline(line, 80);        //read a line
cout<<line<< "\n";              //display it
filin.getline(line, 80);
cout<<line<< "\n";
filin.getline(line, 80);
cout<<line<<"\n";
filin.close( );                 //disconnect stunames and
filin.open("stumarks");         //connect stumarks
cout<< "\nThe contents of stumarks file are given below \n";
filin.getline(line,80);         //read a line
cout<<line<<"\n";               //display it
filin.getling(line, 80);
cout << line << "\n";
filin.getline(line, 80);
cout << line << "\n';
flin.close ();
return 0;
}
```

The output produced by the above program will be as follows:

The content of stunames file are given below:

Amod Kumar

Suparna Mitra

Neeraj Gakhar

The contents of stumarks file are given below:

56.78

67.98

98.00

Now, if you want to use two or more files simultaneously, you need to create separate streams for handling these files simultaneously. Let us see how this can be achieved as it is shown in the following example program that uses multiple files simultaneously.

```
#include<fstream.h>
#include<conio.h>
int main()
{
clrscr( );
// Assuming that the two files (stunames & stumarks) are already' created
ifstream filin1, filln2;      // create two input streams
filin1.open("Stunames");      // connect stunames to first stream
filln2.open("Stumarks");      // connect stumarks to second stream
char line[80];
cout<< "The contents of Stunames and Stumarks are given below.\n";
filin1.getline(line, 80);
cout<<line<< "\n";
filin2.getline(line, 80);
cout<<line<< "\n";
filin1.getline(line, 80);
cout<<line<< "\n";
filin2.getline(line, 80);
cout<<line<<"\n";
filin1.getline(line, 80);
cout<<line<< "\n";
filin2.getline(line, 80);
cout<<line<< "\n";
filin1.close ( );
filin2.close( );
return 0;
}
```

The output produced by the above program will be as follows:

The contents of stunames file are given below:

Amod Kumar

56.78

Suparna Mitra

67.98

Neeraj Gakhar

98.00

To print the output of a program on printer, open file "PRN" as an output file (DOS names printer as PRN file) and write data to it.

More about open(): File Modes, File Pointers and their Manipulations

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

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

The second argument of open( ), the *filemode,* is of type *int*, and you can choose one from several constants defined in the *ios* class. Following table lists the *filemodes* available and their meaning.

Constant	Meaning
ios::in	This specifies that the file is capable of input. It opens file for reading.
ios::out	This specifies that the file is capable of output. It opens file for writing. This also opens the file in ios::trunc mode by default. This means an existing file is truncated when opened. That is, its previous contents are discarded.
ios::ate	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.
ios::trunc	This value causes the contents of a preexisting file by the same name to be destroyed and truncates the file to zero length.
ios::nocreate	This causes the open( ) function to fail if the file does not already exist. It will not create a new file with that name.
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.
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 new lines. However, no such character translations occur in files opened in binary mode. Any file can be opened in either text or binary mode. The only difference is the occurrence of character translations in text mode.

The ifstream and ofstream constructors and the open( ) methods provide default values for the second argument (the filemode argument). For instance, the ifstream.open( ) method and constructor use ios::in (open for reading) as the default value for the mode argument, while the ofstream.open( ) method and constructor use ios::out (open for writing) as the default.

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

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

You can combine two or more *filemode constants* using the C++ bitwise OR operator (symbol |). For instance, look at the following statements:

```
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.

11.4 CLOSING A FILE

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

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

The close( ) flushes the buffer before terminating the connection of the file with the stream.

11.5 STREAM STATE MEMBER FUNCTIONS

Many things can go wrong while operating on a data file through a program. The file your program is attempting to open may not exist; the file may be locked by some other program for some operation; for instance.

Fortunately, the I/O system in C++ provides information about the result of every I/O operation performed with files. For this purpose C++ provides an object of enumerated data class called ios::io_state. The object ios::io_state acts as flag to indicate the state of result in the previous operation. It can take the following values:

- ***goodbit:*** This value indicates that the previous operation was successful without any error.
- ***eofbit:*** This value of ios::io_state indicates that end of file has been reached in the previous operation.
- ***failbit:*** This value of ios::io_state indicates that the previous operation resulted in an non-fatal error meaning that the program can still work but the result may not be as desired.
- ***Badbit:*** This value of ios::io_state indicates that the previous operation resulted in a fatal error and further operations are not possible.

Apart from these ios::io_state values there are corresponding functions defined in the file stream object which can be conveniently called to check the status of the file operations. These functions are:

- ***rdstate():*** This function returns the current status of the error-flags. For example, the rdstate() function will return badbit if the last file operation encountered a fatal error.
- ***bad():*** This function returns a boolean value (TRUE or FALSE). It returns true if the last file operation was bad else return false. It actually checks whether badbit is set or not.

- *eof():* This function also returns a boolean value (TRUE or FALSE). It returns true if the last file operation reached the end of the file under operation. It checks eofbit.
- *fail():* This function also returns a boolean value (TRUE or FALSE). It returns true if the last file operation failed due to any reason else returns false. It checks failbit.
- *good():* This function also returns a boolean value (TRUE or FALSE). It returns true if the last file operation succeeded without any error else returns false. It checks goodbit.
- *clear():* This function sets the desired status bit. It takes a flag as parameter and set that flag so that the program may behave according to this state. It is frequently used in cases where the file operation failed and yet the program must proceed. This can be done by clearing the status with passing goodbit parameter to clear() function.

The following program snippet demonstrates the use of status check. The file stream name has been assumed to be myfile.

```
if(myfile.rdstate() == ios::eofbit)
    cout << "End of file has been reached!\n";
if(myfile.rdstate() == ios::badbit)
    cout << "Encountered a fatal I/O error!\n";
if(myfile.rdstate() == ios::failbit)
    cout << "Encountered a non-fatal I/O error!\n";
if(myfile.rdstate() == ios::goodbit)
    cout << "Success without errors!\n";
```

Here is a complete program to show practical usages of these state functions.

```
#include <fstream.h>
void main()
{
    ofstream myfile("data.dat");            //to create data.dat file
    myfile.close();
    ofstream checkState("data.dat",ios::noreplace);
```

//this statement will result into an I/O error because the file data.dat already exists and the statement is trying to //create it without replacement. This will set failbit

```
    if(checkState.rdstate() == ios::failbit)
      cout << "Error : File already exists!\n";
checkState.clear(ios::goodbit);     //set the current status to ios::goodbit
if(checkState.rdstate() == ios::goodbit)  //check if goodbit was set correctly
      cout << "OK! goodbit is et alright!\n";
checkState.clear(ios::eofbit);            //set the state to ios::eofbit
if(checkState.rdstate() == ios::eofbit)   // check again the state
      cout << "EOF!\n";
```

```
checkState.close();
}
```

It is a good programming practice to check the status of data files during file operations failing which the program may crash in incomprehensible manner.

11.6 READING/WRITING A CHARACTER FROM/INTO A FILE

Reading and writing a character in a data file has been dealt with in the previous sections in detail. The general procedure of reading a file one character at a time is listed below:

- Create an input file stream from <fstream.h> header file:

  ifstream name_of_input_stream;

- Open the data file by passing the file name (optionally full name) to this input stream:

  name_of_input_stream.open("data.dat");

  Both the above statements can be combined in the following:

  ```
  ifstream name_of_input_stream("data.dat");
  ```

- Set up a character type variable to hold the read character.

  char ch;

- Read a character from the opened file using get() function:

  name_of_input_stream.get(ch);

  This way you can read the entire file in a loop stopping condition of the loop being the end of file:

  ```
  while(!filename.eof())
  {
        name_of_input_stream.get(ch);
        //process the read character
  }
  ```

- When finished close the file using close() function:

  ```
  name_of_input_stream.close();
  ```

  The if stream class is defined in fstream.h header file. Therefore you must include this file in your program. The complete program is listed below.

  ```
  //reading a file one character at a time
  #include <fstream.h>
  void main() //the program starts here
  {
        ifstream filename("c:\cppio.dat");
        char ch;
        while(!filename.eof())
  ```

```
        {
            filename.get(ch);
            cout << ch;
        }
        filename.close();
}
```

The general procedure of writing one character at atime in a file is listed below:

- Create an output file stream from <fstream.h> header file:

  ofstream name_of_output_stream;

- Open the data file by passing the file name (optionally full name) to this output stream:

  name_of_output_stream.open("data.dat");

  Both the above statements can be combined in the following:

  ofstream name_of_output_stream("data.dat");

- Write a character in the opened file using << operator:

  name_of_output_stream << 'A';

- When finished close the file using close() function:

  name_of_output_stream.close();

  The ofstream class is defined in fstream.h header file. Therefore you must include this file in your program. The complete program is listed below.

```
//Writing a character in a data file
#include <fstream.h>
int main()
{
  ofstream filename("data.dat");
  filename << 'A';
  filename.close();
  return 0;
}
```

Some useful File Operation Functions

In addition to the function discussed thus far, there are many more functions which come handy in writing practical programs on data file processing. An assorted list of them is given below:

tellg()

This function returns an int type value representing the current position of the pointer inside the data file opened in input mode as demonstrated in the following program:

```
//Program demonstrating tellg() function
```

```
#include <fstream.h>
void main()
{
    //Assume that the text stored in data.dat file is "Welcome to C++"
    ifstream myfile("data.dat");
    char ch;
    for(int j=0; j<4;j++)
    myfile.get(ch);
    cout <<myfile.tellg() << endl;
    //this should return 5, as four characters have been read from the file so
the current pointer points at next
    //character, i.e., 5th characterHello is 5 characters long
    myfile.close();
}
```

tellp()

This function does to output files what tellg() does to input files. It returns an int type value representing the current position of the pointer inside the data file opened in output mode as shown in the following code snippet.

```
    ofstream myfile("data.dat");
    myfile<<"India";
    cout << myfile.tellp();
    //this should return 6, as five characters have been written in the file
so the current pointer points at next
    //position, i.e., 6th
    myfile.close();
```

seekg()

While reading data from a file this function is used to shift the pointer to a specified location as shown in the following code snippet.

```
    //Assume that the text stored in data.dat file is "Welcome to C++"
    ifstream myfile("data.dat");
    char ch;
    for(int j=0; j<4;j++)
    myfile.get(ch);
    myfile.seekg(-2);
    //this will take the pointer to 2 characters before the current location.
It will now point to (l)
```

seekp()

This function does to output files what seekp() does to input files. It shifts the pointer to the specified location in the output file. If you want to overwrite the last 5 characters, you will have to go back 5 characters from the current pointer position. This you can do by the following statement:

```
myfile.seekp(-5);
```

ignore()

This function is used when reading a file to ignore certain number of characters. You can use seekg() as well for this purpose just to move the pointer up in the file. However, ignore() function has one advantage over seekg() function. You can specify a delimiter character in ignore() function whence it ignores all the characters up to the first occurrence of the specified delimiter. The prototype is of ignore() function is given below:

```
fstream& ignore( int, char);
```

Where int is the count of characters to be ignored and delimiter is the character up to which you would like to ignore as demonstrated in the following program:

```
//demonstration of ignore() function
#include <fstream.h>
void main()
{
      //Assume that the text contained in data.dat file is "Welcome to C++"
      ifstream myfile("data.dat");
      static char Carray[10];
      //go on ignoring all the characters in the input up to 10th character
unless an 'm' is found
      myfile.ignore(10,'m');
      myfile.read(Carray,10);
      cout << Carray << endl;        //it should display "me to C++"
      myfile.close();
}
```

getline()

This function is used to read one line at a time from an input stream until some specified criterion is met. The prototype is as follows:

```
getline(Array,Array_size,delimiter);
```

The stopping criterion can be either specified number of characters (Array_size) or the first occurrence of a delimiter (delimiter) else the entire line (up to newline character '\n') is read. If you wish to stop reading until one of the following happens:

1. You have read 10 characters
2. You met the letter 'm'
3. There is new line

Then the function will be called as follows:

getline(Carray,10,'m');

The use of getline() function is demonstrated in the following example.

```
//Demonstration of getline() function to read a file line-wise
#include <fstream.h>
void main()
{
    //Assume that the text contained in data.dat file is "Welcome to C++"
    ifstream myfile("data.dat");
    static char Carray[10];
    myfile.getline(Carray,10,'m');
    cout << Carray << endl;    //the output should be "Welco"
    myfile.close();
}
```

peek()

This function returns the ASCII code of the current character from an input file stream very much like get() function, however, without moving the pointer to the next character. Therefore, any number of successive call to peek() function will return the ASCII code of same character each time. To convert the ASCII code (as returned by peek() function use char type cast) as demonstrated in the following code program.

```
//Demonstration of peek() function
#include <fstream.h>
void main()
{
    // Assume that the text contained in data.dat file is "Welcome to C++"
    ifstream myfile("data.dat");
    char ch;
    myfile.get(ch);
    cout << ch << endl;          //the output should be 'W' and the
                                 pointer will move to point 'e'
    cout << char(myfile.peek()) << endl;  //should display "e"
    cout << char(myfile.peek()) << endl;  //should display "e" again
    cout << myfile.peek() << endl;        //should display 101
    myfile.get(ch);
    cout << ch << endl;          //will display "e" again and move the
                                 pointer to 'l'
    myfile.close();
    }
```

putback()

This function returns the last read character, and moves the pointer back. In other words, if you use get() to read a char and move the pointer to next character, then use putback(), it will show you the same character, but it will set the pointer to previous character, so the next time you call get() again, it will again show you the same character as shown in the following program:

```
//Program demonstrating use of putback() function
#include <fstream.h>
void main()
{
    // Assume that the text contained in data.dat file is "Welcome to C++"
    ifstream myfile("data.dat");
    char ch;
    myfile.get(ch);
    cout << ch << endl;          //output will be 'W'
    myfile.putback(ch);
    cout << ch << endl;          //output will again be 'W'
    myfile.get(ch);
    cout << ch << endl;          // output will again be 'W'
    myfile.close();
}
```

flush()

I/O streams are created and maintained in the RAM. Therefore, when dealing with the output file stream, the data is not saved in the file as the program enters them. A buffer in the memory holds the data until the time you close the file or the buffer is full. When you close the file the data is actually saved in the designated file on the disk. Once the data has been written to the disk the buffer becomes empty again.

In case you want to force the data be saved even though the buffer is not full without closing the file you can use the flush() function. A call to flush() function forces the data held in the buffer to be saved in the file on the disk and get the buffer empty.

11.7 EXCEPTION HANDLING DATA FILES OPERATIONS

Sometimes during file operations, errors may also creep in inadvertently. For instance, a file being opened for reading might not exist. Or a file name used for a new file may already exist. Or an attempt could be made to read past the end-of-file. Or such an invalid operation may be performed. There might not be enough space in the disk for storing data.

To check for such errors and to ensure smooth processing, C++ file streams inherit *'stream-state'* members from the ios class that store the information on the status of a file that is being currently used. The current state of the I/O system is held in an integer, in which the following flags are encoded:

Name	Meaning
eofbit	1 when end-of file is encountered, 0 otherwise.
failbit	1 when a non-fatal I/O error has occurred, 0 otherwise
badbit	1 when a fatal I/O error has occurred, 0 otherwise
goodbit	0 value

There are several error-handling functions supported by class ios that help you read and process the status recorded in a file stream. Following table lists these error handling functions and their meaning.

int bad ( ):	Returns non-zero value iran invalid operation is attempted or any unrecoverable error has occurred. However, if it is zero (false value), it may be possible to recover from any other error reported and continue operation.
int eof ( ):	Returns non-zero (true value) if end-of-file is encountered while reading otherwise returns zero (false value).
int fail ( ):	Returns nonzero (true) when an input or output operation has failed.
int good ( ):	Returns nonzero (true) if no error has occurred. This means; all the above functions are *false*. For instance, if fin.good( ) is *true*, everything is okay with the stream named as fin and we can proceed to perform I/O operations. When it returns *zero*, no further operations can be carried out.
clear( ):	Resets the error state so that further operations can be attempted.

These functions may be used in the appropriate places in a program to locate the status of a file stream and thereby take the necessary corrective measures. For example,

```
ifstream fin;
fin.open("Master");
while(!fin.fail())
{
//process the file

if(fin.eof())
{
      //terminate the program
}
else if (fin.bad())
{
      // report fatal error
    }
    else
{
fin.clear ( ) ;
// clear error-state flags
    }
}
```

11.8 BINARY FILES OPERATIONS

A binary file is a file of any length that holds bytes with values in the range 0 to 0xff. (0 to 255). These bytes have no other meaning. In a text file a value of 13 means carriage return, 10 means line feed, 26 means end of file. Software reading or writing text files has to deal with line ends. In Linux these are just separated by line feeds but Windows uses carriage returns and line feeds.

In modern terms we can call a binary file a stream of bytes and more modern languages tend to work with streams rather than files. Use these commands to perform low level file operations. These are useful for reading and writing all data types in both big and little endian formats. When performing read and write access to the memory(x) array in common formats use the Memory Statements. To read and write sequential values using ASCII text files, use the **FINPUT** and **FPRINT** statements.

Statement	Purpose
FOPEN	Opens a file for binary R/W access.
FCLOSE	Closes a binary file.
FSEEK	Sets the file pointer.
FWRITE	Writes data to a file.
FREAD	Reads data from a file.

Example:

```
# Open a file for reading and writing
# Give it an ID of 1 (variable of)
 $myfile = $temppath "\test.bin" of = 1
 fopen of $myfile
# Write text - 29 chars
 $a = "This is a sample text string"
 fwrite of $a 29
# Write a byte
 bytval = 116
 fwrite of bytval u8
# Initialize a table of 1000 square roots and
# write to the file as big endian 64 bit floats
 a = 1
 loop begin 1000
 memory(a) = sqrt(a)
 a = a + 1
 loop end
 fwrite of memory(1) 1000 f64 b
# Change our mind and replace some of
# the file contents at file location 10
```

```
 fseek of 10
 $a = " `changed` "
 fwrite of $a 11
# seek to the end of the file and write
# the square root table as 16 bit integers
 fseek of 8030
 fwrite of memory(1) 1000 u16 b
# Read what we have written
 clear
# Text at file pos 0
 fseek of 0
 fread of $a 29
 print "Text: " $a
# The f64 square root of 398
 fpos = 30 + (397 * 8)
 fseek of fpos
 fread of num f64 b
 print "Number at " fpos " is " num
# The integer square root of 398
 fpos = 30 + (1000 * 8) + (397 * 2)
 fseek of fpos
 fread of num u16
 print "The little endian integer at " fpos " is " num " (wrong)"
 fseek of fpos
 fread of num u16 b
 print "The big endian integer at " fpos " is " num " (correct)"
# Always close the file when done
 fclose of
exit
```

11.9 CLASSES AND FILE OPERATIONS

Classes are also created in the memory just like structures and hence are lost at the termination of the program that created them. In order to store a class' data in a data file the same approach is applied to classes as shown in the following code snippet.

```
class MyAcc
{
public:
 MyAcc(int accNoV, string lastNameV, string firstNameV, double balV)
```

```
 {
  setAccNo(accNoV);
  setLastName(lastNameV);
  setFirstName(firstNameV);
  setBalance(balV);
 }

 int getAccNo() const
 {
  return accNo;
 }
 void setAccNo(int accNoV)
 {
  accNo = accNoV;
 }
 string getLastName() const
 {
  return lastName;
 }
 void setLastName(string lastNameS)
 {
  const char *firstNameV = lastNameS.data();
  strcpy(lastName, lastNameV,10);
  lastName[10] = '\0';
 }
 string getFirstName() const
 {
  return firstName;
 }

 void setFirstName(string firstNameS)
 {
  const char *firstNameV = firstNameS.data();
  strncpy(firstName, firstNameV,10);
  firstName[10] = '\0';
 }
 double getBalance() const
 {
```

```
  return balance;
 }
 void setBalance(double balanceV)
 {
  balance = balanceV;
 }
private:
 int accNo;
 char lastName[20];
 char firstName[20];
 double balance;
};
int main()
{
 fstream putCredit( "data.dat", ios::in | ios::out | ios::binary);
 MyAcc one(100,"Mehta","Vibhor",10000.0);

 // seek position in file of user-specified record
 putCredit.seekp((one.getAccNo() - 1 ) * sizeof(MyAcc));
 // write user-specified information in file
 putCredit.write(reinterpret_cast< const char * >( &one ),sizeof(MyAcc));
putCredit.close();
return 0;
}
```

Classes are also created in the memory just like structures and hence are lost at when you run this program the class data members will be stored in a disk file named data.dat. In a similar manner you can read data from a disk file into the class data members as shown in the code snippet listed below.

```
class MyAcc
{
public:
 MyAcc( )
 {
 }
 MyAcc(int accNoV,string lastNameV,string firstNameV,double balV)
 {
  setAccNo(accNoV);
  setLastName(lastNameV);
  setFirstName(firstNameV);
```

```
 setBalance(balV);
}

int getAccNo() const
{
 return accNo;
}
void setAccNo(int accNoV)
{
 accNo = accNoV;
}
string getLastName() const
{
 return lastName;
}
void setLastName(string lastNameS)
{
 const char *lastNameV = lastNameS.data();
 strncpy(lastName, lastNameV,10);
 lastName[10] = '\0';
}

string getFirstName() const
{
 return firstName;
}
void setFirstName(string firstNameS)
{
 const char *firstNameV = firstNameS.data();
 strncpy(firstName, firstNameV,10);
 firstName[10] = '\0';
}
double getBalance() const
{
 return balance;
}
void setBalance(double balanceV)
{
```

```
  balance = balanceV;
 }

private:
 int accNo;
 char lastName[20];
 char firstName[20];
 double balance;
};

int main()
{
 ifstream getCredit( "data.dat",ios::in);
 MyAcc one;
 getCredit.read( reinterpret_cast< char * >( &one), sizeof( MyAcc));
 getCredit.close();
return 0;
}
```

This program snippet demonstrates how a class stored in a binary file can be read back into the object created into the memory.

11.10 ARRAY OF CLASS OBJECTS AND FILE OPERATIONS

The functions read( ) and write( ) can also be used for reading and writing array of class objects. These functions handle the entire structure of an object as a single unit, using the computer's internal representation of data. For instance, the function write( ) copies a class object from memory byte by byte with no conversion. But one thing that must be remembered is that only data members are written to the disk file and not the member functions.

The length of an object is obtained by **sizeof** operator and it represents the sum total of lengths of all data members of the object. Following program illustrates how class object can be written to and read from the disk files.

Program for reading and writing class objects.

```
#include<fstream.h>
#include<connio.h>          // for clrscr( )
class Student
{ char name[40];
  char grade;
  float marks;
  public:
```

```
    void getdata(void);
    void display(void);
};
void Student::getdata(void)
{  char ch;
   cin.get(ch);
   cout<<"Enter Name:";
   cin.getline(name, 40);
   cout<<"Enter Grade:";
   cin>>grade;
   cout<<"Enter marks:";
   cin>>marks;
   cout<<"\n";
}
void Student::display(void)
{  cout<< "Name:"<<name<< "\t"
     << "Grade:"<<grade<< "\t"
     << "Marks:"<<marks<< "\t" << "\n";
}
int main( )
{  clrscr( );
   Student arts[3];  //declare array of 3 objects
   fstream filin;          //input and output file
   filin.open("Stu.dat", ios::in | ios::out);
   if(!filin)
   {     cout<< "Cannot open file !! \n";
         return 1;
   }
   cout << " Enter details for 3 students \n";
   for(int i=0, i<3; i++)
   {     filin.read((char *) & arts[i], sizeof (arts[i]));
         arts[i].display( );
   }
   filin.close( );
   return 0;
   }
```

If the input is given as follows:

Enter details for 3 students

Enter Name: Disha Aggarwal

Enter Grade: A

Enter Marks : 92

Enter Name: Mohit Dutta

Enter Grade: B

Enter Marks : 64

Enter Name: Vijaya Marwa

Enter Grade: B

Enter Marks : 66

Then the output produced is:

The contents of stu.dat are shown below.

Name : Disha Aggarwal	Grade: A	Marks: 92
Name : Mohit Dutta	Grade: B	Marks: 64
Name : Vijay Modi	Grade: B	Marks: 66

The above program uses for loop for reading and writing objects. This is possible because we know the exact number of objects in the file. In case, we do not know the exact number of objects, we may use while(filin) test approach to decide the end of the file.

11.11 NESTED CLASSES AND FILE OPERATIONS

A *nested class* is declared within the scope of another class. The name of a nested class is local to its enclosing class. Unless you use explicit pointers, references, or object names, declarations in a nested class can only use visible constructs, including type names, static members, and enumerators from the enclosing class and global variables.

Member functions of a nested class follow regular access rules and have no special access privileges to members of their enclosing classes. Member functions of the enclosing class have no special access to members of a nested class. The following example demonstrates this:

```
class A {
 int x;
 class B { };
 class C {
 // The compiler cannot allow the following
 // declaration because A::B is private:
 // B b;
 int y;
```

```
 void f(A* p, int i) {
 // The compiler cannot allow the following
 // statement because A::x is private:
 // p->x = i;
 }
 };
 void g(C* p) {
 // The compiler cannot allow the following
 // statement because C::y is private:
 // int z = p->y;
 }
};
int main() { }
```

The compiler would not allow the declaration of object `b` because class `A::B` is private. The compiler would not allow the statement `p->x = i` because `A::x` is private. The compiler would not allow the statement `int z = p->y` because `C::y` is private.

You can define member functions and static data members of a nested class in namespace scope. For example, in the following code fragment, you can access the static members `x` and `y` and member functions `f()` and `g()` of the nested class `nested` by using a qualified type name. Qualified type names allow you to define a `typedef` to represent a qualified class name. You can then use the `typedef` with the `::` (scope resolution) operator to refer to a nested class or class member, as shown in the following example:

```
class outside
{
public:
  class nested
  {
  public:
   static int x;
   static int y;
   int f();
   int g();
  };
};
int outside::nested::x = 5;
int outside::nested::f() { return 0; };
typedef outside::nested outnest;  // define a typedef
int outnest::y = 10;      // use typedef with ::
int outnest::g() { return 0; };
```

However, using a typedef to represent a nested class name hides information and may make the code harder to understand.

You cannot use a typedef name in an elaborated type specifier. To illustrate, you cannot use the following declaration in the above example:

```
class outnest obj;
```

A nested class may inherit from private members of its enclosing class. The following example demonstrates this:

```
class A {
private:
 class B { };
 B *z;

 class C : private B {
 private:
  B y;
//  A::B y2;
  C *x;
//  A::C *x2;
 };
};
```

The nested class `A::C` inherits from `A::B`. The compiler does not allow the declarations `A::B y2` and `A::C *x2` because both `A::B` and `A::C` are private.

11.12 RANDOM ACCESS FILE PROCESSING

Every file maintains two pointers called *get-pointer* and *put-pointer* which tell the *current position* in the file where writing or reading will take place. (A file pointer in this context is not like a C++ pointer but it works like a *book-mark* in a book.) These pointers help attain random access in file. That means moving directly to any location in the file instead of moving through it sequentially.

There may be situations where random access is the best choice. For example, you have to modify a value in record number 59. Here using random access techniques, you can place the file pointer at the beginning of record 59 and then straightway process the record. If sequential access is used, then you'll have to unnecessarily go through first twenty records in order to reach at record 59.

In C++, random access is achieved by manipulating seekg(), seekp(), tellg() and tellp() functions. The seekg() and tellg() functions allow you to set and examine the *get-pointer,* and the seekp() and tellp() functions perform these operations on the *put-pointer.*

```
seekg()      -      istream & seekg (long);          Form 1
    istream & seekg (long, seek_dir);               Form 2
ofstream & seekp (long);                            Form 1
    ofstream & seekp (long, seek_dir);              Form 2
```

```
(seek_dir takes the definition enum seek_dir{ beg, cur, end};)
tellg( ) -      long tellg();
tellp( ) -      long tellp();
```

The seekg() (or seekp() when used according to *Form* 1, it moves the *get-pointer* or *put-pointer* to an absolute position. For example,

```
ifstream fin;
ofstream fout;
fin.seekg(30)
will move the get-pointer (in ifstream) to byte number 30 in the file.
fout.seekg(30)
```

will move the *put-pointer* (in of stream) to byte number 30 in the file.

When seekg( ) (or seekp( )) function is used according to *Fonn* 2, it moves the *get-pointer* (or *put-pointer)* to a position relative to the current position, following the definition of seek_dir. Seek_dir is an enumeration (defined in *iostream.h)* that has following values:

```
ios::beg //refers to beginning of the file
ios::cur // refers to current position in the file
ios::end // refers to end of the file.
```

For example,

```
fin.seekg(30, ios::beg);
fin.seekg(- 2, ios::cur);
```

The methods tellp() and tellg( ) return the position (in terms of byte number) of *put-pointer* and *get-pointer* respectively in an *output file* and *input file* respectively.

Let us now but all these concepts learnt so far into practice. Following example program accomplishes this task. This program lets you add a record in the file, display a specified record in the file. This program uses a menu and a loop to let you select from the list of actions indefinitely.

```
#include<fstream.h>
#include<conio.h>
#include<string.h>
#include<stdio.h>
int count=0;
class student
{
char name[40];
char grade;
float marks;
public:
void getdata(void);
void display(void);
```

```
void moddata();
};

void student::getdata(void)
{
char ch;
cin.get(ch);
clrscr( );
gotoxy(15,10);
cout<< "Add Student Data\n";
gotoxy(17,12);
cout<< "Record N" << ++ count) <<endl;
gotoxy(1,14);
for(int i=0; i<40; i++)
{
name[i]=" ";
grade=' '
marks = 0.0;
cout<<"Enter name: "; cin.getline(name, 40);
cout << "\nEnter grade: "; cin>> grade;
cout << "\nEnter marks: "; cin>> marks;
cout<< "\n";
}
clrscr( );
}
void student::display(void)
{
            clrscr( );
gotoxy(15,10);
    cout<<"Student Details\n";
    gotoxy(1,12);
    cout     << "Name :" << name<< "\t"
    <<"Grade :"<<grade<< "\t"
    <<"Marks:"<<marks<<"\n";
    }
    student::moddata(void)
    {
      char. nlll[ 40j,gr=' '; float Illk=-T,
```

```
clrscr( );
gotoxy( 15,8 );cout< < "Modify Student Data\n".< < endl;
char ch= cin.get(); cout< <chi
gotoxy( 17, 1 0);
cout< <"Current Details\n ";
gotoxy( 17, 11);
cout< <"    \n\n";
cout < < "Name: "< < name < < "\tGrade : "< < grade < <"\tMarks : "< < marks
< < end!: gotoxy( 17, 16);
cout< <"Enter-New Details\n";
gotoxy( 17, 17); cout< <"gofOXY( 10,24);
, cout< < "If you want to retain old Name or Grade, just Press Enter.";
gotoxy(SO, 17); cout< < end I;
    cout< <" Name: ";    dn.getline(nm,40);
    cout< <" Grade: ";   gr=getche();
    cout< <" \tMarks : ";      dn> > mk;
if (strlen(nm}1-.-O) strcpy(name,nm);
if «gr!= 13) && (gr!=32» grade=gr;
II ASCII values for Enter and Space are 13 and 32 respectively. if(mk> =0)
marks=mk;
gotoxy( 1 0,24);
    clreol( );     lito clear uptil end .of the line
}
Int main( )
( clrscr( );
Student stud;
fstream flnout;
flnout.open("Stumast.dat",ios::ln I ios::out I ios::binary);
If(!flnou_) .
{ cout< < "Cannot open file !!\n";
return 1;
}
lot choice, mrec=O,offset=O;
char ans;
do
{ clrscr();
cout< < "\n\n\n\t\tMaln Menu\n";
    cout< < "\t\t  \n"< <endl;
cout< <" 1. Add Record." < <endl;
```

```
cout< <" 2. Modify Rec:ord."< <endl;
cout< <" 3. Display Record."< < end I;
cout«" 4. Exit." < <endl;
cout< < "Enter Your choice.. ( 1-4)..";
cin> > choice;
switch(cholce)

\n\n";

I
j
case I: stud.getdata();
mrec = count;
    offset= «mrec-l)*sizeof(Student));
fmout.seekp(offset,ios::beg); / /place the file pointer
finout. write« char *) &stud, sizeof(Student));
break;
case 2: if(!count)
    {cout< <"No Record has been added yet.\n";
    cout< < "Please run option I first of.all\n\n";
gotoxy( I 0,23); cout< < _'Press a key to continue... \n";
getch( );
    break;
}
cout< < "\n\nModlfy Which Record? #";
dn> > mrec;
if(mrec > count)
{cout< < "\n\n\nOnly "< <count< <It records have been added.\n."'; cout< <
"Invalid Record Number..\n";
gotoxy( 15,23);
cout< <"Press a key to continue..."< <endl;
getch( );
    break;
}
else
{ offset = (mrec-I) * sizeof(Student);
finout.seekg(offset,ios::beg); / /place the file pointer
finout.read«char *) &stud, sizeof(Student));
```

```
stud.display( );
cout<.<"Modify this record? (y/n)..";  .
dn> >ans;
if(ans=='y'llans=='Y')
{ cout<-< "Enter New Detalls\n";
stud.mod_data();
    finout.seekp(offset,los::beg);   / /place the file pointer
finout.write«char *) &stud, sizeof(Student));
cout< < "\n Record Modified\n";
gotoxY(30,24);
cout< < "Press a key to continue..." < <endl;;
    getch( );
}
break;
}
case 3: If(!count)
    { cout< < "\n\n\n\nNo Record has beeR--adMd yet\n" ;
    cout< < "Please run option I first of all\o\n";

gotoxy(10,23);
cout< <"Pre_s a key to continue..";
getch( );
break;
,
cout< < "\n\n\DisplayWhich Record? U";
cin> >mrec; cout< < end I;
if(mrec > count)   .
{COl:lt< < "\n\n\nOnly "< <count< <" records have been added.\n"; cout< <:
"Invalid Record"Number..\n";
gotoxy( 1 5,P);
cout < < "_ress a l<ey to continue..." < < end_
_etch( );
reak;
)
else{ offset = (mrec- 1) * sizeof(Student); fmout.seekg(offset,ios::beg);
finout.read( (char *) &stud, sizeof(Student); stud.display( );
gotoxy( 10,23);
cout< < "Press a key to continue._\n";
```

```
    getch( )i       "
    }
break; case 4: break; default: cout < < "Wrong choice!! Valid choices are 1-
4.\n";
    break; !i'.
}
} while (choice> = 1 && choice < =3);
t"
flnout.cIose( );

retUrn 0; }
```

When this program is executed, fol1owing menu appears.

Main Menu

1. Add Record. 2. Modify Record. 3. Display Record. 4. Exit.

Enter Your choice..(1:;4)..

Check Your Progress

1. Fill in the blanks:
   i. The data file itself can exist in many .....
   ii. The connections with a file are ..... automatically when the input and output stream objects expire.
   iii. ..... a file flushes the buffer which means the data remaining in the buffer is moved out of it in the direction it is ought to be.
   iv. The close() flushes the buffer before ..... the connection of the file with the stream.
2. Give Short Answers
   i. A file in open state file has two pointers associated with it. What are they?
   ii. Please quote two methods of opening a data file in a program.

11.13 LET US SUM UP

A data of a file is stored in the form of readable and printable characters then the file is known as text file. A file contains non-readable characters in binary code then the file is called binary file. The function get( ) read and write data respectively. The read( ) and write( ) function read and write block of binary data. The close( ) function close the stream. The eof( ) functions determines end-of-file by returning true otherwise false. In C++ to open a file, you must-first obtain a stream. There are three types of streams - input, output, and input/output. A file is closed by disconnecting it with the stream it is associated with. Many things can go wrong while operating on a data file through a program. The file your program is attempting to open may not exist; the file may be locked by some other program

for some operation; for instance. Reading and writing a character in a data file has been dealt with in the previous sections in detail. Sometimes during file operations, errors may also creep in inadvertently. For instance, a file being opened for reading might not exist.C++ treats each source of input and output uniformly. The abstraction of a data source and data sink is what is termed as stream. A stream is a data abstraction for input/output of data to and fro the program.C++ library provides prefabricated classes for data streaming activities. In C++, the file stream classes are designed with the idea that a file should simply be viewed as a stream or array or sequence of bytes. A file normally remains in a closed state on a secondary storage device until explicitly opened by a program. The << operator is a predefined operator. This line puts the text in the file or an output stream.C++ offers a host of different opening modes for the input file each offering different types of reading control over the opened file. The file opening modes have been implemented in C++ as enumerated type called ios. During a program execution structures are not created on a disk. Rather they are created in the memory. The structures stored in the memory are lost when the program terminates. Binary files provide a better way of storing structures into a data file on the disk using read() and write() functions. Classes are also created in the memory just like structures and hence are lost at the termination of the program that created them.

11.14 KEYWORDS

File: A storage unit that contains data. A file can be stored either on tape or disk.

Stream: A stream is a general name given to a flow of data.

EOF: End of file.

EOL: End of line.

I/O Stream: An abstraction that views every input/output device as a stream or array or sequence of bytes.

Read Pointer: This pointer always points to the next character in the file which will be read if a read command is issued next. After execution of each read command the read pointer moves to point to the next character in the file being read.

Write Pointer: The write pointer indicates the position in the opened file where next character being written will go. After execution of each write command the write pointer moves to the next location in the file being written.

File Opening Mode: An enumerated data type that determines what operations can be applied on an opened file.

Goodbit: A file operation status flag that indicates that the previous operation was successful without any error.

Eofbit: A file operation status flag that indicates that end of file has been reached in the previous operation.

Failbit: A file operation status flag that indicates that the previous operation resulted in an non-fatal error meaning that the program can still work but the result may not be as desired.

Badbit: A file operation status flag that indicates that the previous operation resulted in a fatal error and further operations are not possible.

11.15 QUESTIONS FOR DISCUSSION

1. How will you open and close a file in C++?
2. What are stream state member functions?
3. How will you read/write a character from a file?
4. How is C++ able to treat all the input and output operation uniformly?
5. Develop a simple C++ class that provides rudimentary functionalities of a Database management system including:
   (a) Creating a table
   (b) Record insertion
   (c) Record updation
   (d). Record deletion
   (e) Report generation

Check Your Progress: Model Answer

1. i. Forms
   ii. Closed
   iii. Closing
   iv. Terminating
2. i. Read Pointer and Write Pointer
   ii. These methods are described below.
   ifstream filename("filename <with path>"); Or ofstream filename("filename <with path>");

11.16 SUGGESTED READINGS

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

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

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